TRACE-BASED LEARNING FOR AGILE HARDWARE DESIGN AND DESIGN AUTOMATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yuan Zhou

May 2021

TRACE-BASED LEARNING FOR AGILE HARDWARE DESIGN AND DESIGN AUTOMATION

Yuan Zhou

Cornell University, May 2021

Modern computational platforms are becoming increasingly complex to meet the stringent constraints on performance and power. With the larger design spaces and new design trade-offs brought by the complexity of modern hardware platforms, the productivity of designing high-performance hardware is facing significant challenges. The recent advances in machine learning provide us with powerful tools for modeling and design automation, but current machine learning models require a large amount of training data. In the digital design flow, simulation traces are a rich source of information that contains a lot of details about the design such as state transitions and signal values. The analysis of traces is usually manual, but it is difficult for humans to effectively learn from traces that are often millions of cycles long. With state-of-the-art machine learning techniques, we have a great opportunity to collect information from the abundant simulation traces that are generated during evaluation and verification, build accurate estimation models, and assist hardware designers by automating some of the critical design optimization steps.

In this dissertation, we propose three trace-based learning techniques for digital design and design automation. These techniques automatically learn from simulation traces and provide assistance to designers at early stages of the design flow. We first introduce **PRIMAL**, a machine-learning-based power estimation technique that enables fast, accurate, and fine-grained power modeling

of IP cores at both register-transfer level and cycle-level. Compared with gate-level power analysis, PRIMAL achieves an average error within 5% while offering an average speedup of over 50x. Secondly, we present **Circuit Distillation**, a machine-learning-based methodology that automatically derives combinational logic modules from cycle-level simulation for applications with stringent constraints on latency and area. In our case study on network-on-chip packet arbitration, the learned arbitration logic is able to achieve performance close to an oracle policy under the training traffic, improving the average packet latency by 64x over the baselines while only consuming area comparable to three eight-bit adders. Finally, we discuss **TraceBanking**, a graph-based learning algorithm that leverages functional-level simulation traces to search for efficient memory partitioning solutions for software-programmable FPGAs. TraceBanking is used to partition an image buffer of a face detection accelerator, and the generated banking solution significantly improves the resource utilization and frequency of the accelerator.

BIOGRAPHICAL SKETCH

Yuan Zhou received his bachelor's degree in Electronics Engineering from Tsinghua University, Beijing, China in 2015. He then joined the School of Electrical and Computer Engineering (ECE) at Cornell University as a Ph.D. student. Since then, Yuan studied under the supervision of Prof. Zhiru Zhang at the Computer Systems Laboratory, where he passed his Ph.D. candidacy exam and received a Master of Science degree in ECE in March 2019. During his graduate study, Yuan has worked on a variety of research areas, including high-level synthesis, deep learning acceleration, benchmarking, and application of machine learning in digital design. He interned at NVIDIA research and Google in summer 2018 and summer 2020, respectively.

For everyone who has faith in me.

ACKNOWLEDGEMENTS

I sincerely appreciate the support from my family, my friends, my mentors, and all other people who have helped and encouraged me during my graduate study. This dissertation would not have been possible without their support.

First and foremost, I would like to thank my advisor, Prof. Zhiru Zhang, for his support and supervision during my Ph.D. Zhiru saw my potential of becoming a researcher and brought me into CSL. In the past five-and-a-half years, Zhiru advised me patiently and consistently provided precious suggestions to my research. Zhiru always encouraged me and had faith in me, even when I was questioning myself.

I would also like to thank my dissertation committee members: Prof. David Albonesi, Prof. Adrian Sampson, and Dr. Mark Haoxing Ren. I thank Dave for detailed suggestions on my dissertation and concrete feedback on my power estimation work from a circuit expert's perspective. His comments during our discussions made me realize the value of high-level modeling and pointed me in the direction of research that is actually important to hardware designers. I thank Adrian for being a very supportive mentor and an inspiring source of research ideas. Adrian also taught me a lot about programming languages and compilers, which will be very useful for the first job in my career. Last but not least, I would like to thank Mark for being my mentor during my internship at NVIDIA in 2018 and serving on my committee afterwards. It was from a discussion with Mark and other collaborators where we got the idea of accelerating power estimation with machine learning. Mark pointed me to the correct direction during my internship and continued to provide suggestions to my research after I left NVIDIA.

I am grateful to all members of the Zhang research group and the CSL com-

munity for building such a lively and friendly environment for my research and daily life. Specifically, I really appreciate the help from the Zhang group alumni: Dr. Steve Haihang Dai, Dr. Gai Liu, Dr. Ritchie Zhao, Dr. Nitish Srivastava, Dr. Zhenghong Jiang, and Prof. Cunxi Yu. They got me familiar with the research infrastructure in our group, provided valuable suggestions on how to be successful during my Ph.D., and served as a great source of research ideas. I acknowledge all the hard work from my collaborators within and outside of Cornell, and I thank Weizhe Hua, Yu Gan, and Chenhui Deng for inspiring research discussions. I would also like to thank my friends at CSL and all members of the Zhang group for the fun activities, including badminton, table tennis, card games, and picnic. Special thanks to Bo Yuan at Google and Dr. Yanqing Zhang at NVIDIA for being outstanding mentors during my internships and providing valuable suggestions to my future career.

Finally, I would like to express my deepest gratitude to my parents, Fugen Zhou and Wenyan Liu, for raising me to become who I am today and always being supportive throughout my graduate study. Also, I would like to deeply thank my girlfriend Mengyao Xu, for being loving, caring, and supportive in the past three years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

With the end of Dennard scaling, the performance improvement of single-core microprocessors has significantly slowed down in the past decade [51]. In order to satisfy the stringent performance and power requirements under current and future application scenarios, modern computational platforms are increasingly relying on parallel and/or heterogeneous processing to achieve high performance under a tight power budget. Multi-core CPUs are now prevalent in embedded systems [13], desktop- and server-grade computers [72, 74], as well as supercomputers [70]. Hardware acceleration using specialized accelerators is also becoming popular in both cloud and embedded computing platforms. In fact, FPGA and ASIC accelerators are now empowering a number of major products and services provided by some of the industry giants in their datacenters [11, 36, 81], while also being integrated into smartphones [55] and self-driving cars [113].

Unfortunately, high performance and power efficiency often come at the cost of the scale and complexity of modern hardware platforms. Notably, the latest Apple M1 system-on-chip (SoC) contains sixteen billion transistors, with an eight-core CPU, an eight-core GPU, and a neural engine for machine learning (ML) workloads [13]. The NVIDIA GPUs are massively parallel with thousands of CUDA cores and contain specialized accelerators for ray tracing and tensor computation [114, 115]. Hardened AI engines are also introduced into modern FPGAs which have already incorporated dedicated DSP units and memory modules [165]. Furthermore, the hardware industry is rapidly redesigning and updating their products to adapt to the emerging applications. The increase

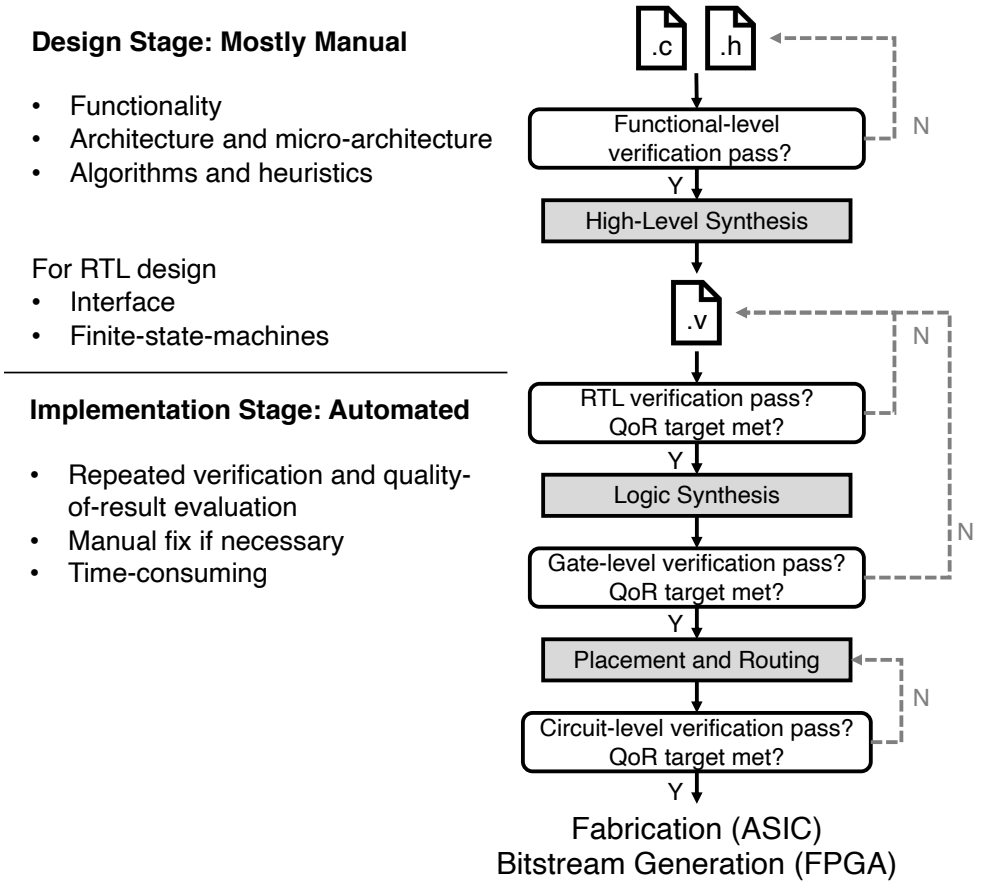


Figure 1.1: Digital design flow with HLS — The design stage is still mostly manual even with the assistance of HLS. The implementation stage, including synthesis, placement, and routing, is mostly automated by EDA tools. Hardware designs must be iteratively modified to pass verification and meet QoR targets.

in complexity and the urge for fast development are posing significant challenges to both hardware designers and the existing electronic design automation (EDA) tools.

1.1 Challenges in Digital Design: Productivity and Quality

Figure 1.1 shows a typical hardware design flow based on high-level synthesis (HLS)[1]. While the implementation of digital circuits is highly automated with the assistance of EDA tools, the hardware is usually designed manually by experienced engineers. At the design stage, developers must first provide a functional description of the hardware. In cases where the more traditional register-transfer-level (RTL) design methodology is applied, developers then further specify the RTL description including the connection of sub-modules, interfaces, and the logic and state transition of each sub-module. HLS raises the level of abstraction from RTL to the behavioral level and automates the generation of hardware microarchitecture. However, the design of hardware architecture is still not fully automated. Furthermore, modern hardware relies on carefully-designed heuristics to make decisions at run time, and the design of these heuristics is mostly manual.

Hardware designers rely on their intuition and experience to search for good design points in the vast design space. For modern hardware platforms, this process is becoming more and more challenging because of the even larger design space and the potentially different design trade-offs brought by new applications. Under this scenario, experience from past projects may not lead to successful design decisions, and the heuristics that were effective may no longer be satisfying for the new hardware. As a result, developers have to spend many more iterations to reach a good design point, and for every iteration the EDA tools must be rerun for functional verification and QoR evaluation. In some cases, it might be out of the designers' capability to design an effective heuristic

[1]More traditional hardware design flows start from the RTL design step.

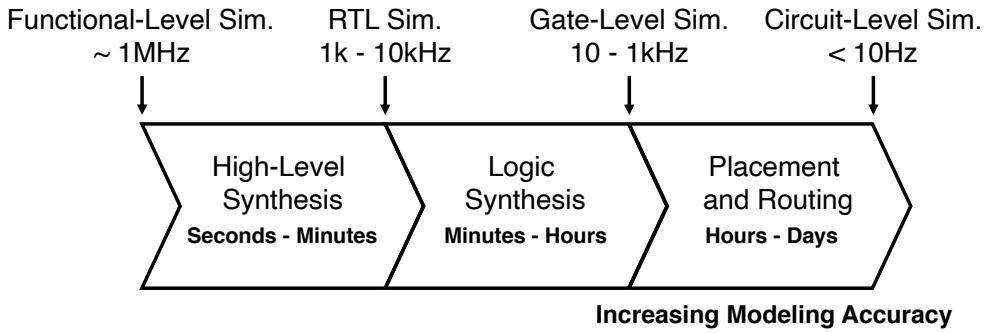


Figure 1.2: Time consumption of EDA steps and simulation throughput at different abstraction levels — Modeling accuracy increases at lower levels of abstraction. The speed of cycle-level simulation supported by SystemC [1] or GEM5 [99] is between functional-level simulation and RTL simulation.

algorithm or find a satisfying design point in a limited amount of time, and the final hardware implementation will be sub-optimal. The growing complexity of hardware systems will make manual hardware design more challenging and time-consuming.

Even without the increased number of design iterations, optimizing and implementing a hardware design is a time-consuming process. Figure 1.2 shows a rough breakdown of the time required at each step of the digital design flow. For large designs, the implementation stage itself may take hours or days, and the simulation at lower levels of abstraction can be as slow as ten cycles per second. Despite the low speed/throughput of these steps, they are necessary for designers to obtain meaningful evaluation results in the current hardware design flow, because modeling at earlier steps or higher levels of abstraction is not accurate enough. Combined with the increased number of design iterations, the turn-around time of hardware design will quickly become intolerable for fast-paced development.

1.2 Trace-Based Design-Specific Learning for Agile Hardware Design

There is an emerging trend of applying ML to EDA [69]. Existing works have explored a variety of topics including high-level modeling [44, 150, 175], design-space exploration [97, 109, 149, 158], automated architectural and micro-architectural design [30, 60, 133, 146, 171, 182], verification [68, 78], placement [107, 143, 164, 173], and layout pattern generation [170, 177]. These techniques have been shown to achieve superior performance on a variety of tasks in the EDA toolchain, and are able to significantly accelerate the hardware design flow by providing assistance to developers.

While ML techniques are shown to be effective for EDA problems in existing work, learning a design-agnostic model that performs well for any arbitrary design is very difficult. A design-agnostic ML model must learn the underlying characteristics of the target technology library and the optimization algorithms in the EDA tool flow. Due to the complexity of the digital design flow, many algorithms used in EDA are highly sophisticated and stochastic. Modern ML models would require a massive amount of training data from many different designs to effectively learn the behavior of these algorithms. In addition, hardware designs from different application domains have distinct characteristics. Without a collection of comprehensive datasets, new designs are likely to be out of the training distribution, causing the ML models to make inaccurate predictions.

Unfortunately, due to the lack of open-source designs and the long execution time of EDA tools, collecting and constructing large and useful datasets for

Table 1.1: Summary of techniques presented in this dissertation.

Technique	PRIMAL	Circuit Distillation	TraceBanking
Traces Used	Gate-level/RTL/Cycle-level	Cycle-level	Functional-level
Learning Technique	ML	ML & RL	Graph-based data mining
Target Designs	Hardened IP cores	Partially-reconfigurable modules	Specialized accelerators
Design Metric	Power	Performance & Area	Performance

EDA problems remains a daunting task. An alternative is to use ML for design-specific learning. In this setting, the data collection effort is greatly reduced since the training data can be easily acquired from the given design. Furthermore, with design-specific learning, ML models only need to learn about the target design. The behavior of hardware designs is usually deterministic. In addition, while the ML models still need to learn the behavior of the EDA tools on the target design, this learning task is much easier than learning complete EDA algorithms. As a result, design-specific models can usually provide more detailed and more accurate predictions than their design-agnostic counterparts. In this dissertation we focus on using design-specific learning to improve the quality of one single design.

We argue that simulation traces generated by various stages of the hardware design flow are a good source of information for design-specific learning. Hardware developers often run millions or even billions of cycles of tests to guarantee the correctness of their designs and discover performance bottlenecks. Furthermore, depending on the level of abstraction, simulation traces contain different levels of details about the design: (1) Functional-level simulation verifies the functional correctness of a software-specified design and provides a sequence of transactions that is useful for understanding the high-level functional behavior of the design; (2) Cycle-level simulation evaluates the latency and throughput of the design while exposing the value of critical signals in each cycle for analysis and verification; (3) Simulation at RTL or lower ab-

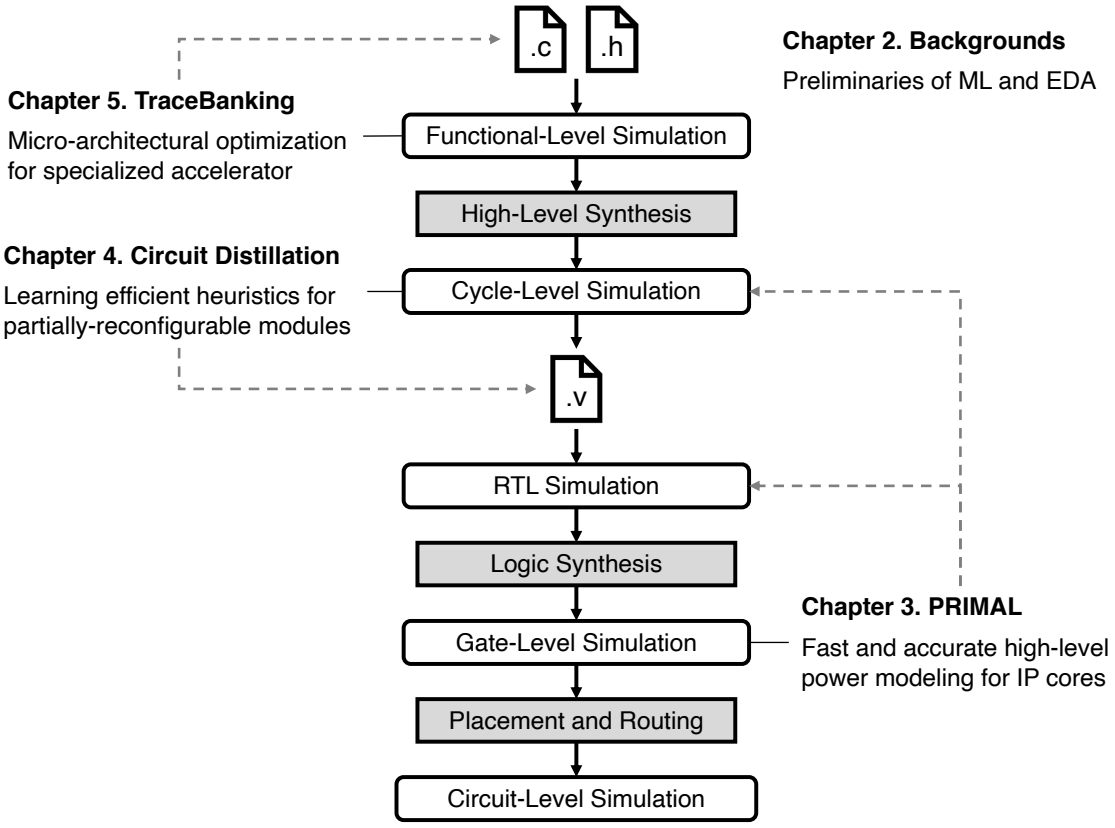


Figure 1.3: Dissertation outline.

straction levels exposes more internal details of the design, better reflects its timing behavior, and can be used to calculate the power and thermal characteristics of the underlying circuit. These details are necessary for accurate and fine-grained analysis of hardware designs. By collecting and analyzing these details, trace-based learning with ML can effectively assist the hardware design process.

1.3 Dissertation Overview

In this dissertation we introduce three trace-based, design-specific learning techniques for agile hardware design and design automation. As shown in

Figure 1.3, our research aims to improve both the optimization and modeling aspects of the hardware design flow. Moreover, our proposed techniques address the challenges that arise from optimizing different hardware design metrics, including performance, area, and power. Table 1.1 shows the coverage of this dissertation, which spans multiple levels of hardware design abstraction and explores a rich set of learning techniques. A more detailed overview of this dissertation is as follows:

- Because of the broad coverage of this dissertation, Chapter 2 is dedicated to introducing the necessary background for readers to get a better understanding of the techniques discussed in this dissertation. We will introduce the digital design flow and also broadly discuss several widely-used ML techniques, including linear and tree-based ML models, deep learning (DL), and reinforcement learning (RL).
- Chapter 3 presents PRIMAL, a ML-based methodology that provides fine-grained RTL and cycle-level power estimation for IP cores. In this work we explore the potential of using DL models to characterize large circuit modules such as RISC-V processor cores and accelerators designed using HLS. Our experiments show that for RTL power estimation, convolutional neural networks can effectively model the power of large hardware modules, even if the test workloads are completely independent from the training sets. PRIMAL can achieve less than 5% error on cycle-by-cycle RTL power estimation while providing around 50× speedup compared with gate-level power analysis. This work was published in DAC'19 [186]. For cycle-level power estimation, we leverage recurrent neural networks to tolerate the inaccuracies in cycle-level simulation traces. Thanks to the faster simulation throughput at cycle level, we were able to achieve an-

other 3.5× speedup over our RTL power estimation technique on a collection of HLS accelerators, with marginal degradation in estimation accuracy.

- Chapter 4 proposes a fully automated methodology to directly learn combinational logic from simulation traces and presents a case study on a network-on-chip (NoC) router arbitration task. Specifically, we leverage deep RL to learn a neural network (NN) agent which implements an optimized arbitration policy from simulation. The proposed Circuit Distillation flow then uses tree-based ML models as a bridge between the learned NN policy and a compact, combinational logic implementation. Experiments show that the learned arbitration logic is able to achieve a 64× reduction in average packet latency and a 5% improvement in network throughput over the baseline FIFO policy under the training traffic. This work has been accepted to DAC'2021.
- Chapter 5 presents TraceBanking, a data-driven approach to automatically generating on-chip memory banking solutions for software-programmable FPGAs. Our approach takes a memory trace of the target application as input and uses a graph-coloring-based algorithm to generate an efficient memory banking with no conflicts. Compared with existing compile-time memory partition techniques, our approach can handle arbitrary memory access patterns that are fixed at run time. This work was published in FPGA'17 [184].
- Chapter 6 summarizes the contributions of this dissertation and discusses future research directions.

1.4 Collaborations, Funding, and Previous Publications

This dissertation would not be possible without the contributions of my colleagues within the Zhang Research Group and Batten Research Group in the Computer Systems Laboratory at Cornell University, as well as collaborators from University of California Los Angeles (UCLA), Lehigh University, and the VLSI research group at NVIDIA. My advisor and committee chair, Prof. Zhiru Zhang, provided valuable suggestions and assistance to all the projects mentioned in this dissertation. The ideas of the high-level power modeling techniques presented in Chapter 3 originated from a discussion with Dr. Mark Ren, Dr. Yanqing Zhang, Dr. Ben Keller, and Dr. Brucek Khailany at NVIDIA research during my summer internship in 2018, and we collaborated on the RTL power estimation part of the PRIMAL project. The HLS power estimation part of PRIMAL is a collaboration with Dr. Young-kyu Choi and Prof. Jason Cong at UCLA. Dr. Choi and Prof. Cong provided valuable support on cycle-level simulation of HLS designs. The Circuit Distillation project in Chapter 4 was done in collaboration with Prof. Jieming Yin at Lehigh University and Hanyu Wang from Shanghai Jiao Tong University[2]. The initial idea of tackling this problem arose from Prof. Yin's paper on learning NoC arbitration policy using reinforcement learning [171]. Prof. Yin also provided detailed suggestions on the usage of the GEM5 simulator [99], the hyperparameter settings during training, and the writing of the submission. Khalid Al-Hawaj from Prof. Christopher Batten's group contributed significantly to the TraceBanking project presented in Chapter 5.

This dissertation was supported in part by a DARPA Young Faculty Award,

[2]Hanyu was a (remote) research intern at Cornell when participating in this project.

NSF Awards #1337240, #1453378, #1512937, #1909661, the Intel ISRA Program, the Semiconductor Research Corporation (SRC), and research gifts from Xilinx, Inc and NVIDIA Corporation. A complete list of my publications during my Ph.D. in reverse chronological order is as follows:

1. Yuan Zhou, Hanyu Wang, Jieming Yin, and Zhiru Zhang, **Distilling Arbitration Logic from Traces using Machine Learning: A Case Study on NoC**, *to appear in Design Automation Conference (DAC)*, December 2021.
2. Eshan Singh, Florian Lonsing, Saranyu Chattopadhyay, Maxwell Strange, Peng Wei, Xiaofan Zhang, Yuan Zhou, Jason Cong, Deming Chen, Zhiru Zhang, Priyanka Raina, Clark Barrett, and Subhasish Mitra, **A-QED Verification of Hardware Accelerators**, *Design Automation Conference (DAC)*, July 2020.
3. Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G. Edward Suh, **Channel Gating Neural Networks**, *Thirty-third Conference on Neural Information Processing Systems (NeurIPS)*, December 2019.
4. Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G. Edward Suh, **Boosting the Performance of CNN Accelerators with Dynamic Fine-Grained Channel Gating**, *International Symposium on Microarchitecture (MICRO)*, October 2019.
5. Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang, **PRIMAL: Power Inference using Machine Learning**, *Design Automation Conference (DAC)*, June 2019.
6. Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang, **HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing**,

International Symposium on Field-Programmable Gate Arrays (FPGA), February 2019.

7. Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F.Y. Young, and Zhiru Zhang, **Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning**, *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April/-May 2018.
8. Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Velasquez, Wenping Wang, and Zhiru Zhang, **Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs**, *International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2018.
9. Yuan Zhou, Khalid Al-Hawaj, and Zhiru Zhang, **A New Approach to Automatic Memory Banking using Trace-Based Address Mining**, *International Symposium on Field-Programmable Gate Arrays (FPGA)*, February 2017.

CHAPTER 2

PRELIMINARIES ON EDA AND MACHINE LEARNING

In this chapter we provide background knowledge for the techniques presented in this dissertation. The rest of this chapter will introduce the electronic design automation (EDA) flow, basic concepts and important techniques of machine learning (ML), as well as the application of learning-based methods in EDA and computer architecture.

2.1 Overview of the EDA Flow

Figure 2.1 shows the digital design flow with more details. As mentioned in Chapter 1, designing a digital circuit is a time-consuming process that involves many design iterations. The process of finding a satisfactory design point is usually referred to as design space exploration (DSE). Each promising design point must be repeatedly verified and evaluated at different levels of abstraction, and manual optimizations are necessary if the design fails to meet the constraints. When evaluating the design, some metrics such as area and frequency can be directly provided by the EDA tools, while other metrics like latency and power can only be obtained by running simulation using a comprehensive set of workloads. The extensive use of simulation further increases the design time because detailed simulation is very time-consuming, especially at lower levels of abstraction such as gate-level and circuit-level.

As shown in Figure 2.1, accurate estimations of the quality-of-result (QoR) metrics can only be obtained at later stages of a typical EDA flow. To facilitate DSE, ideally designers would like such estimations to be available earlier in

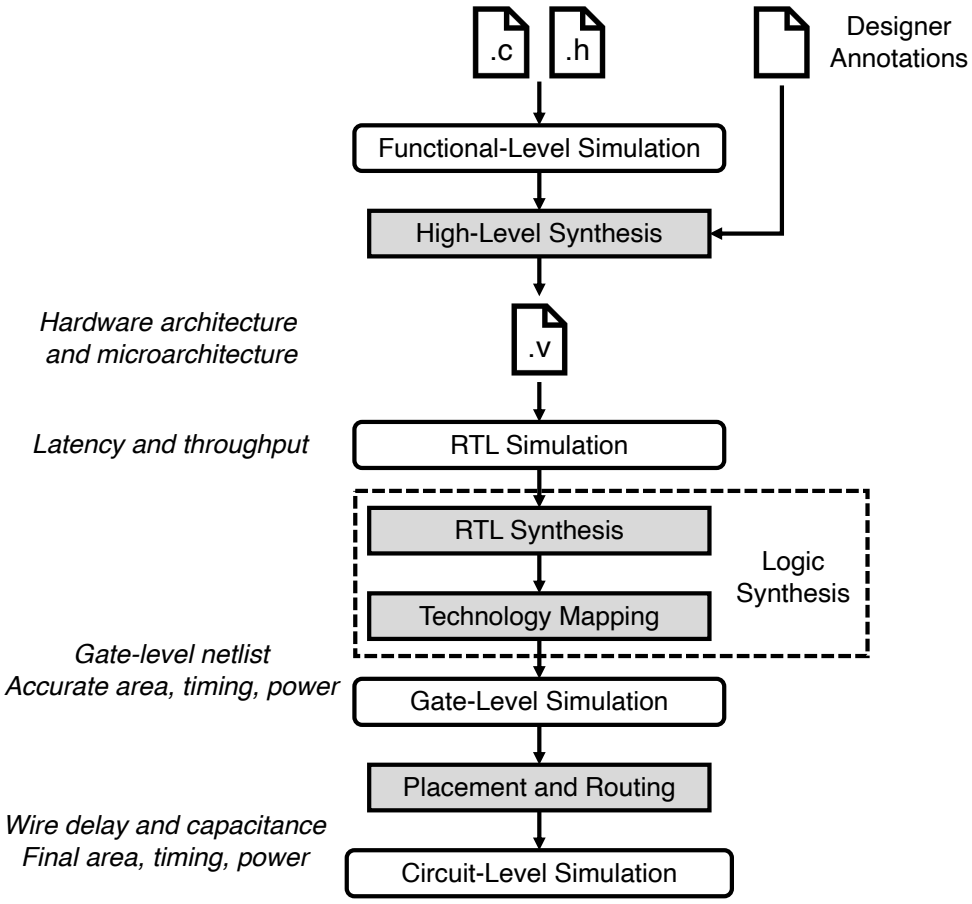


Figure 2.1: Digital design flow with HLS — Accurate estimations of many QoR metrics are only available at lower levels of abstraction because each step in the design flow introduces additional details about the design.

the design flow, where the level of abstraction is higher and exploring different design points is easier. Unfortunately, high-level modeling of digital circuits is inherently challenging, because each step in the EDA flow introduces many additional details about the design which are hard to predict at higher levels of abstraction. Chapter 3 focuses on the power estimation aspect of this problem. The rest of this section briefly introduces each step in the EDA flow.

High-Level Synthesis The high-level synthesis (HLS) step converts a behavioral-level description of the design to an register-transfer-level (RTL) de-

scription. The behavioral-level description can be specified in C/C++ [71, 168], OpenCL [73, 167], or Python [88, 166]. With HLS, the designer no longer has to design the interface between hardware modules or the finite-state-machine (FSM) of each module. Modern HLS tools usually provide a set of predefined interface protocols. In addition, the scheduling and binding algorithms in the HLS tools are able to directly generate optimized datapath and FSM from the behavioral-level description. The designers are given the flexibility to specify additional optimizations to the generated hardware, such as parallelizing the computation, customizing the datatype, or partitioning the data memory for higher memory bandwidth. While the decisions of where and how to add these optimizations need to be made manually in modern HLS tools, automatically finding good combinations of the optimizations and effective optimization solutions are active lines of research.

The RTL source code generated by the HLS tools accurately describe the architecture and microarchitecture of the hardware. The reports from HLS tools contain a lot of useful information such as the type and count of operators, the size of memories, and the latency of fixed-latency hardware modules. For hardware blocks with variable latency, accurate performance measurements can be obtained from RTL simulation. However, the low-level implementation details of the design are yet to be determined by the downstream flow. Even with the information from the generated RTL representation, the HLS tools have limited knowledge on the selection of operators or the sizing of gates, and can only provide very crude estimates of wire delay and capacitance. All these missing details make accurate modeling at RTL and higher abstraction levels extremely challenging.

Logic Synthesis Logic synthesis refers to the step which transforms the RTL description written in hardware description languages (HDLs) into a technology-specific gate-level netlist. As shown in Figure 2.1, logic synthesis can be further divided into two steps: RTL synthesis and technology mapping. RTL synthesis converts the HDL into a technology-independent logic representation. This process involves several key steps, including RTL elaboration, datapath synthesis, and logic minimization. RTL elaboration converts the HDL code into an abstract, intermediate format which is suitable for downstream processing of the synthesis tool. The common operators such as addition and multiplication are directly converted to logic by datapath synthesis, while the rest of the design are synthesized to logic and further optimized by the logic minimization step.

Technology mapping takes the optimized technology-independent logic representation as input and maps it to components provided by the technology library. For ASICs, the technology library contains basic elements of digital circuits, such as flip-flops and combinational logic gates with different drive strengths, memory modules, and register files. When mapping to FPGAs, the mapper is limited to a small set of elements provided by the FPGA architecture, including look-up tables (LUTs), flip-flops, digital-signal-processing (DSP) units, and block RAMs (BRAMs).

After technology mapping, the gate-level implementation of the design is precisely defined. The synthesis tool now has complete information about the size, capacitance and delay of each gate in the design. As a result, the tool will be able to provide more accurate estimations of area, frequency, and power, which are often used as indicators of the design quality in practice. However, the gate-

level netlist still does not contain any topological information, so the details of the wires and interconnects are unavailable. The wire delay and capacitance significantly affect the frequency and power consumption of the design. Without these details, the modeling accuracy at gate-level is still limited.

Placement and Routing The placement and routing steps are often referred to as physical design. For ASIC design, these steps arrange the components of the gate-level netlist on the chip area and connect them together without violating the constraints specified by the fabrication technology. As the names suggest, the placement step places the standard cells and macros onto the chip area, and the routing step connects the components together using metal wires. For FPGAs, the positions of the logic elements are fixed for each device. Therefore, rather than performing placement and routing on an empty chip area, the placer for FPGAs allocate the elements in the synthesized netlist onto the predefined positions, while the routing algorithm for FPGAs configures the programmable interconnects on the FPGA to connect the logic elements. Since the solution spaces of the placement and routing problems are large and hard-to-predict, stochastic algorithms are often used to search for good solutions within a limited amount of time. In modern EDA toolchains, the placer and router often work together to generate higher-quality layouts: the placer tries to predict the final routing quality and generate easy-to-route placements, while the router may slightly modify the placement to improve the routing quality. Because of the complexity and randomness of these steps, it is challenging to accurately estimate all the circuit-level details even from gate-level.

2.2 Machine Learning Overview

In the past decade, ML techniques have been applied to a wide variety of fields including autonomous driving [56], gaming AI [134, 135], image classification and captioning [62, 172], machine translation [142, 151], protein structure prediction [131], and trading [14, 176]. Most of these successes leverage deep learning (DL), a branch of ML that uses deep neural networks (DNNs) with a large amount of parameters to approximate arbitrary target functions. Instead of presenting a rigorous, mathematical introduction to ML, the purpose of this section is to help the readers build an intuitive understanding of the basic concepts and important techniques in the field of ML. For a more comprehensive introduction, interested readers can refer to the abundant online resources [4, 5, 6, 117] or textbooks [54, 111].

At a high level, ML techniques learn from data and try to model the probability distribution of data. The process of "learning from data" is referred to as **training**, and the process of evaluating the learned model using another portion of available data is called **testing** or **evaluation**. During training, a training dataset is provided to the ML model, and the training algorithm tries to optimize a carefully-designed objective function. Since the model only has access to a sampled subset of the input space, it will never observe the complete data distribution, and the distribution in the training set often slightly differs from the actual data distribution because of the randomness of sampling or the bias in the data collection process. As a result, the ML model will **overfit** the training set if it only focuses on perfectly fitting the training distribution but overlooks the generalization to the whole input space. This often happens when the model is too complicated for the task. On the contrary, if the model does not have

enough complexity to fit the training distribution to a satisfying degree, we say the model is **underfitting**. To balance the complexity and generalizability of the model, the objective function during training often has the form shown in Equation 2.1:

$$J(\mathbf{w}, \mathcal{D}) = L(\mathbf{w}, \mathcal{D}) + \lambda R(\mathbf{w}) \tag{2.1}$$

where $\mathbf{w}$ refers to the learnable parameters of the model and $\mathcal{D}$ refers to the training set. The first term on the right hand side, $L(\mathbf{w}, \mathcal{D})$, is called the **loss** term and represents how well the model performs on the training set. The second term, $\lambda R(\mathbf{w})$, is a **regularization** term to constrain the model's complexity and avoid overfitting, where λ and $R(\mathbf{w})$ are often called the regularization factor and the regularization function, respectively. Notice that the form in Equation 2.1 is consistent with constrained optimization problems, and the goal of ML techniques is just to optimize an objective function. As a result, by a broad definition, traditional optimization techniques such as combinatorial optimization methods can also be considered as a form of ML.

Depending on the specific use case, ML techniques can be roughly categorized as **supervised learning**, **unsupervised learning**, and **reinforcement learning**. In supervised learning, the ML models learn from labeled training data. Specifically, each training data sample is associated with a categorical label for classification tasks, or a numerical label for regression tasks. As a result, supervised learning techniques actually model the probability distribution of labels given the input. In contrast, unsupervised learning techniques learn from unlabeled data and try to directly model the distribution of the input data. For reinforcement learning, no training dataset is directly provided to the ma-

chine learning model. Instead, the model (also called an "agent" in the reinforcement learning setting) is given an environment that can respond to the decisions made by the model, as well as a user-defined reward function which evaluates the model's performance based on the state transition in the environment. Under this setup, the goal is to train an agent which can make optimal decisions according to information provided by the environment. During training, the agent repeatedly makes decisions, and new training data is generated on-the-fly every time the agent makes a new decision.

The rest of this section will briefly discuss both supervised learning and reinforcement learning techniques that are used in the approaches introduced in this dissertation. Section 2.2.1 outlines linear models and tree-based models. Section 2.2.2 introduces the fundamentals of deep learning. We also discuss the basics of reinforcement learning in Section 2.2.3, with a focus on deep reinforcement learning.

2.2.1 Linear and Tree-Based Models

Linear models and tree-based models are widely-used, traditional ML models. In contrast to deep learning models that can approximate arbitrary target functions, these two types of models only target a specific type of functions. While this constraint limits the expressiveness of these models, it also allows them to be extremely efficient when the function to be approximated falls into the category that can be accurately modeled by them. Without special mentioning, the following discussions on linear and tree-based models assume a single-target regression scenario.

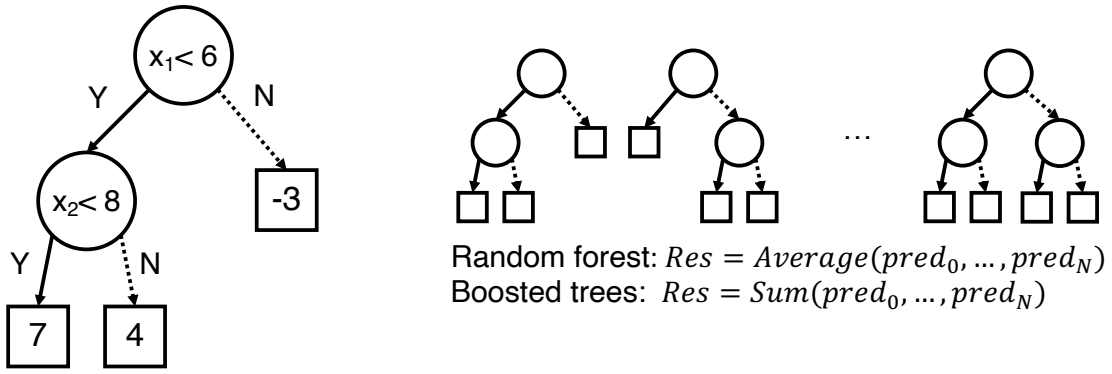
Linear Models As one of the simplest ML models, linear models assume the target function has a linear relationship with the input features. In a single-target regression setting, a linear model can be represented in the form shown in Equation 2.2:

$$f(\mathbf{x}) = \sum_{i=1}^{M} w_i x_i + b \tag{2.2}$$

where $\mathbf{x}$ is an M-dimensional input feature vector of real numbers $\{x_1, ..., x_M\}$, $\mathbf{w} = \{w_1, ..., w_N\}$ represents the learnable linear coefficients, and b is a learnable bias value. Suppose the training set is represented as $\mathcal{D} = \{(\mathbf{x}^1, y^1), ..., (\mathbf{x}^N, y^N)\}$, for regression tasks common objective functions to optimize during training are shown in Equation 2.3 [65, 147].

$$\begin{aligned} J(\mathbf{w}, \mathcal{D}) &= \frac{1}{N}\sum_{i=1}^{N}(f(\mathbf{x}^i) - y^i)^2 && \textbf{Ordinary Least Squares} \\ &\frac{1}{N}\sum_{i=1}^{N}(f(\mathbf{x}^i) - y^i)^2 + \lambda\|\mathbf{w}\|_1 && \textbf{Lasso Regression} \\ &\frac{1}{N}\sum_{i=1}^{N}(f(\mathbf{x}^i) - y^i)^2 + \lambda\|\mathbf{w}\|_2 && \textbf{Ridge Regression} \end{aligned} \tag{2.3}$$

If trained with different loss functions, linear models are also very effective on classification tasks when samples from different classes are linearly-separable. For binary classification tasks, linear models can be used to predict class labels by testing $f(\mathbf{x}^i) > 0$ or by estimating the probability of the data sample belonging to class one with a logistic function ($p(y^i = 1|\mathbf{x}^i) = \frac{1}{1+e^{-f(\mathbf{x}^i)}}$). The latter approach is often referred to as logistic regression [66]. Linear support vector machines (SVMs) improve the robustness of linear classifiers by enforc-



(a) Decision tree.

(b) Random forest and boosted trees.

Figure 2.2: Examples of tree-based models for regression tasks.

ing a minimum margin between any training data sample and the learned decision boundary [41].

Tree-Based Models Tree-based models approximate the target function by repeatedly partitioning the input space and learning a separate function in each partition. As the simplest form of tree-based models, a decision tree recursively partitions the input space using axis-parallel splits and learns a constant value in each partition. Figure 2.2a shows a decision tree for regression. The leaf nodes encode the final predicted values, which are constants for decision trees. Multiple decision trees can be assembled together using bagging or boosting, resulting in random forests [63] or boosted trees [31] as shown in Figure 2.2b. When more complicated models are used at the leaf nodes, such trees are referred to as model trees. For example, if linear regression models are used at the leaves, the resulting linear model tree will be able to learn piecewise-linear functions.

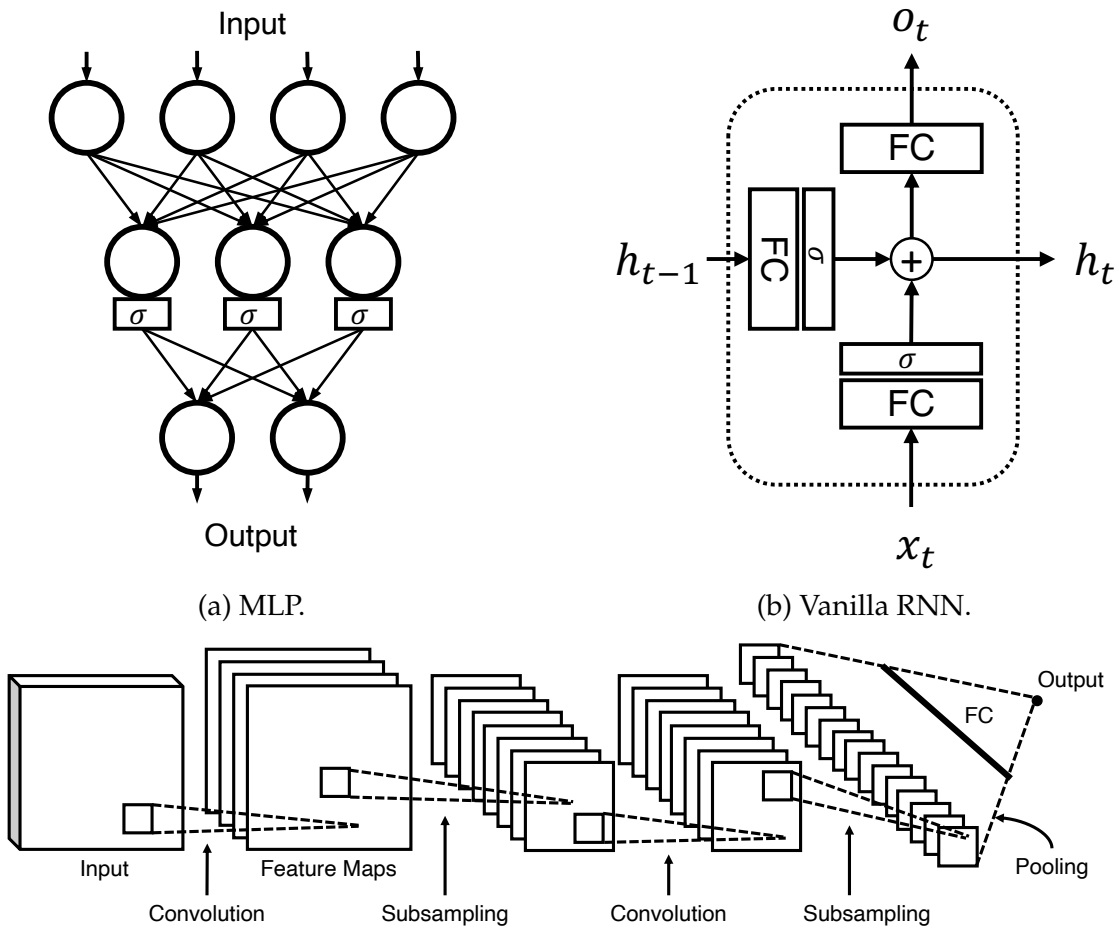
During training, a decision tree or model tree is gradually "grown" by repeatedly partitioning the training data. At the root node, all the training data

is analyzed and a certain gain function is computed to find the best split that maximizes the gain. For classification problems, the difference of gini impurity or mutual information are common gain functions, while for regression problems the mean-squared error between the predictions and ground-truth labels can be used [23]. After the training data is split into two partitions, the same process is repeated at the children of the root node, where each child node only considers one partition of the data. This procedure is applied recursively until the splitting condition is not met. In this case, the node where the partitioning process terminates is a leaf node, and a function is used to fit all training data that arrives at this node.

A decision tree or model tree without any regularization is very prone to overfitting, because it can keep partitioning the input space until there are only a few samples at each leaf. Such a tree is unlikely to generalize well to unseen inputs. Common regularization methods for tree models include constraining the maximum depth, the minimum samples at each leaf, the minimum number of samples to make a split, and the minimum gain to make a split. Bagging is also an effective method to avoid overfitting for tree-based models.

2.2.2 Deep Learning

The fundamental building blocks of deep learning (DL) models are linear "layers" followed by non-linear activation functions. While a single layer has limited expressiveness, the flexibility of combining multiple layers in various ways enables DL models to accurately approximate a wide range of target functions. The non-linear activation functions are crucial for DL models to represent com-



(a) MLP.

(b) Vanilla RNN.

(c) CNN.

Figure 2.3: Examples of the most basic versions of different DL models — **MLP**: Only contains FC layers and activation functions. **Vanilla RNN**: The output o_t at each time step depends on the input at the current time step x_t and the hidden signal from the previous time step h_{t-1}. **CNN**: Performs repeated convolution and subsampling to the input image, where subsampling can be performed using either pooling layers or convolution layers with non-unit stride. One or more FC layers are used at the end of the network to generate outputs for regression or classification.

plicated nonlinear functions. Depending on the specific use case, common activation functions include sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU) [112] and its extensions [38, 61, 102].

Figure 2.3 shows the most basic versions of the DL models used in this

dissertation: multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). As the simplest type of DL model, MLPs only contain fully-connected (FC) linear layers and activation functions. RNNs are designed for sequence processing where the output at the current time step may depend on inputs from previous time steps. As shown in Figure 2.3b, the outputs from the previous time step are used to compute outputs at the current time step. While this recurrent connection enables RNNs to leverage information from the past, it also limits the parallelizability of the RNN models. Popular RNN models such as long short-term memories (LSTMs) [64] and gated recurrent units (GRUs) [34] have more sophisticated mechanisms to control the importance of historical information. CNNs specialize for image processing by replacing the fully-connected layers with convolution layers, which can be considered as large fully-connected layers with most elements being zeros. The convolution layers exploit local information from image patches, while global information is gradually collected through subsampling using pooling layers or convolution layers with non-unit stride. Figure 2.3c shows a basic CNN architecture for regression. Modern CNN models often feature small convolution filters [136], residual connections [62], and batch-normalization [75]. Computationally efficient CNN architectures use group convolution [87, 163] or depthwise-separable convolution [35, 67] to replace normal convolution layers, and apply channel shuffling [100, 178] to avoid significant accuracy degradation.

Training State-of-the-art DL models often contain many layers and millions of parameters in the weight matrices, enabling them to approximate extremely complicated functions. As a result, a large amount of training data is required

to properly optimize all parameters in the model without overfitting. For example, the smaller datasets for image classification contain tens of thousands of images [86, 91], the popular ImageNet dataset has several million images [130], while industrial datasets may contain tens of millions of training samples. It is impractical to directly optimize all parameters in the models with so many training samples using traditional solvers.

DL models are usually trained using stochastic gradient descent (SGD), where the whole training set is divided into small batches containing only tens to hundreds of training samples. For each batch, the model being trained first makes its predictions using its current version of parameters, and the loss function (the L term in Equation 2.1) on this batch is computed. If regularization is applied and the regularization function can be explicitly expressed as a regularization term (the R term in Equation 2.1), the regularization function is also computed based on the current version of parameters. The goal of training is to minimize the objective function (J in Equation 2.1), and the gradient of each parameter in the model can be computed from the objective function using the chain rule. For vanilla SGD, the parameters are then updated using the computed gradient values according to Equation 2.4:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha \bigtriangledown \mathbf{w} \tag{2.4}$$

where $\mathbf{w}_n$ refers to the parameter values at batch n. The hyper-parameter α is called the learning rate and is set by the user to control the step size of each update. To guarantee convergence, the learning rate is usually set to a small value and the model must be trained using a large number of batches. More advanced variants of SGD introduce additional mechanisms and hyper-parameters to es-

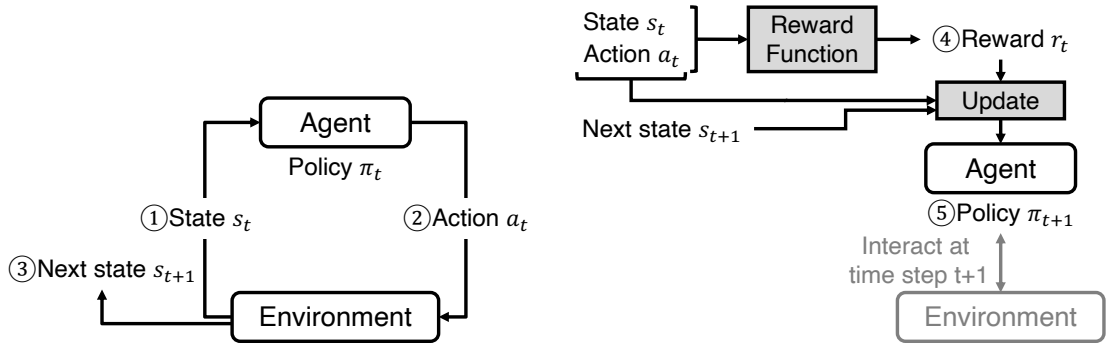
cape bad local minima [123, 141] and automatically adjust the learning rate for individual parameters [46, 83]. Training with SGD allows DL models to effectively exploit a large amount of training samples and be continuously updated with new data.

Regularization Due to the large capacity, DL models can easily overfit to training sets. The most basic regularization method for DL models is weight decay, which uses the L2 norm of all the weights as the regularization term[1]. Weight decay encourages the weights of the model to have small magnitudes so that a small change in the input would not cause drastic changes in the outputs. Dropout [137] randomly drops part of the output neurons in a layer with probability p by setting the output values of these neurons to zeros. Because a different set of neurons are present for each training batch, the neurons learn more robust representations by themselves instead of trying to correct other neurons' mistakes. During evaluation, no neuron is dropped and all the weights in the layer are multiplied by p to maintain the magnitude of the outputs. In practice, reducing the batch size during training also helps avoid overfitting. With smaller training batches, the sampling noise in each batch prevents the model from quickly converging to the optimal solution on the training set and results in a more generalizable model.

2.2.3 Reinforcement Learning

Different from supervised learning and unsupervised learning where a training dataset must be provided in advance, reinforcement learning (RL) techniques

[1]Similar to the regularization term of Ridge regression in Equation 2.3.



(a) Agent interacts with environment. (b) Update the policy.

Figure 2.4: Overview of RL model training.

generate data during the training process and use this data to update a model that is used to make decisions. This model is often referred to as an "agent" in the RL setting. Figure 2.4 presents a high-level illustration of the RL training process. The goal of training is for the agent to learn an effective policy π which can make optimal decisions according to the information from the environment.

The agent is trained by repeatedly making decisions and interacting with the provided environment. At each time step t, the agent is provided with the state of the environment, s_t, and selects an action a_t according to its current policy π_t. The agent then interacts with the environment by performing action a_t and changes the environment's state from s_t to s_{t+1}. A reward value at the current time step, r_t, is computed based on s_t, s_{t+1}, and a_t using a human-designed reward function. The internal parameters of the agent can then be updated using the tuple (s_t, s_{t+1}, a_t, r_t) from the current time step as well as tuples from earlier time steps. The updated policy, π_{t+1}, will be used in the next time step for selecting the most promising action.

Rather than taking a greedy approach and focusing on the immediate reward, the goal of the RL agent is to maximize the long-term cumulative reward.

A common way of computing the long-term cumulative reward R from the current time step T to the future is shown in Equation 2.5, where a user-specified parameter γ determines whether to focus on near-term reward or long-term reward.

$$R = \sum_{t=T}^{\infty} \gamma^{t-T} r_t, 0 < \gamma < 1 \tag{2.5}$$

Q-Learning One important branch of RL is Q-learning [160], where the agent directly tries to predict the long-term cumulative reward of each state-action pair (s, a). The predicted rewards, denoted as $Q(s, a)$, are called Q-values, and the function that computes the Q-value of a given state-action pair is called the Q-function. When making a decision at time step t, the learned policy π_t simply chooses the action with the maximum Q-value. During training, the Q-values are updated following the famous Bellman equation:

$$Q^{\pi_{t+1}}(s_t, a_t) = r_t + \gamma max_{a'}(Q^{\pi_t}(s_{t+1}, a')) \tag{2.6}$$

An intuitive explanation of Equation 2.6 is that the updated Q-value of the current state-action pair $Q^{\pi_{t+1}}(s_t, a_t)$ equals to the reward at the current time step plus the maximum cumulative reward obtainable from the next state times a discount factor. Proof of convergence is out of the scope of our discussion, but in practice the Q-values usually converge well because of the exponential decay of long-term rewards.

Deep Q-Learning Traditional Q-learning techniques often use table-based approaches to store the Q-values. Table-based Q-learning is effective when the state space and action space are both small. When applied to computer architecture, the Q-value tables can be easily updated in hardware during execution [76, 110]. However, table-based Q-learning does not scale well to more complicated problems where the state space and action space are large. With the advance of DL, Mnih et al. propose deep Q-learning (DQN), which represents the Q-function using a DNN [108]. While losing the ability to be efficiently updated in hardware, DQN presents a unified and scalable approach to complicated, large-scale decision-making problems.

During training, the DQN agent is trained with SGD, and the loss function can be as simple as an L2 loss[2]. To encourage the agent to explore the environment, at the beginning of training, the agent has a high probability of choosing a random action instead of following the prediction of the neural network. This probability gradually decreases during the training process so that the agent can learn a stable policy towards the end of training. In addition, since the earlier decisions made by the agent affect the state of the environment, the training samples would be highly dependent on each other if the neural network is only trained using the latest state-action pairs. A "replay memory" mechanism is designed to alleviate the dependency problem and encourage the agent to sufficiently learn from its past experiences. The replay memory caches the state-action pairs and rewards during training. At each time step, the training data to the neural network is randomly sampled from the replay memory.

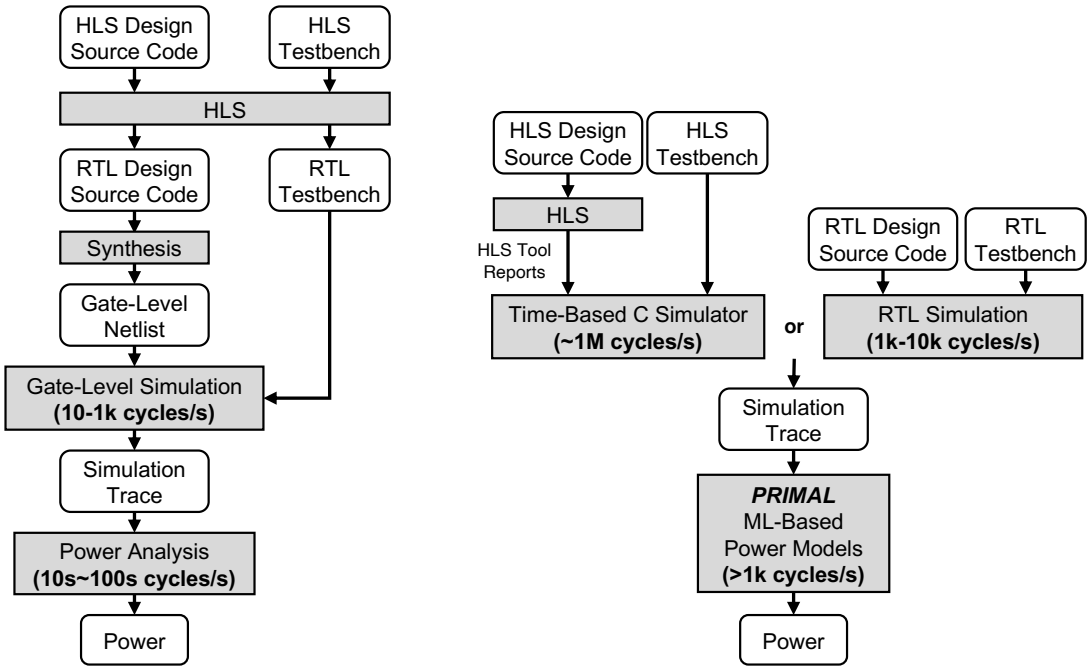
[2] $L = (Q^{\pi_t}(s_t, a_t) - r_t - \gamma max_{a'}(Q^{\pi_t}(s_{t+1}, a'))^2$

CHAPTER 3

PRIMAL: POWER INFERENCE USING MACHINE LEARNING

Modern VLSI design requires extensive optimization and exploration in a large design space to meet the ever stringent requirements with respect to performance, area, and power. Existing electronic design automation (EDA) tools can provide reasonably accurate area and performance estimates at register-transfer level (RTL) or even behavioral level with the aid of high-level synthesis (HLS) tools. However, in order to achieve power closure, designers must obtain detailed power profiles for a diverse range of workloads from different application use cases or even from different levels of design hierarchy. Currently, the common practice is to feed the gate-level netlist and simulation results to power analysis tools such as Synopsys PrimeTime PX (PTPX) to generate cycle-level power traces. Figure 3.1a depicts a typical ASIC power analysis flow, which offers accurate estimates but runs at a very low speed. The throughput of power analysis is in the order of 10-100s of cycles per second, while the gate-level simulation step for generating simulation traces runs at less than one thousand cycles per second. Given the high complexity of present-day ASIC designs, it can take hours or days to perform gate-level power analysis for one intellectual property (IP) core under desired workloads. Furthermore, power-directed optimization is an iterative process, which means designers have to repeat this time-consuming power estimation process after every optimization step. As a result, power analysis has become a critical bottleneck which prevents rapid design-space exploration.

An alternative is to analyze power above the gate level. There exists a rich body of research on power analysis at RTL or higher abstraction levels



(a) Traditional flow. (b) PRIMAL flow.

Figure 3.1: Conventional ASIC power estimation flow vs. PRIMAL — **(a)** With existing EDA tools, designers must rely on the time-consuming gate-level simulation and power analysis for accurate power profiles. **(b)** PRIMAL trains ML-based power models for reusable IPs. Using the trained models, detailed power traces are obtained by running ML model inference on RTL or timed C simulation traces.

[9, 12, 21, 29, 92, 126, 132, 140, 169]. These efforts typically make use of measured constants or simple curve fitting techniques such as linear regression to characterize the power of a given circuit, improving the speed of power analysis at the expense of estimation accuracy. For accurate power characterization, many low-level details of the circuit need to be modeled, including standard cell parameters, sizing of the gates, and clock gating status of the registers. Gate-level power analysis uses them to estimate the switching capacitance and activity factor of each circuit node. However, these low-level details are unavailable at (or above) RTL by design. It is also very difficult for simple analytical models or linear regression models to capture the complex nonlinear relationship between

the register toggles and the total switching capacitance.

In this chapter we introduce PRIMAL, a methodology based on machine learning (ML) for fast and accurate high-level power estimation. PRIMAL leverages gate-level power analysis to train ML models on a set of training workloads. These trained models can then be used to infer power profiles of the same IP core under a different set of user-specified workloads. Figure 3.1b illustrates the inference flow of PRIMAL, which only requires inputs from RTL simulation or C simulation with timing information to generate accurate power estimates at a much higher speed (>1k cycles per second). By greatly reducing the required number of gate-level simulation cycles, PRIMAL allows designers to perform power-directed design space exploration in a much more productive manner. The major technical contributions of this work are five-fold:

1. We present PRIMAL, a novel ML-based methodology for rapid power estimation with RTL or timed C simulation traces. The trained ML models can provide accurate, cycle-by-cycle power estimation for user workloads even when they differ significantly from those used for training.
2. We investigate several established ML models for RTL power estimation, and report trade-offs between accuracy, training effort, and inference speed. Our study suggests that nonlinear models, especially convolutional neural networks (CNNs), can effectively learn power-related design characteristics for large circuits.
3. We propose to use long short-term memory (LSTM) [64] for HLS power estimation. Because LSTMs are designed for sequence processing, they are able to tolerate the inaccuracies in the simulation traces generated by C-based simulators by leveraging history information before the current

cycle.

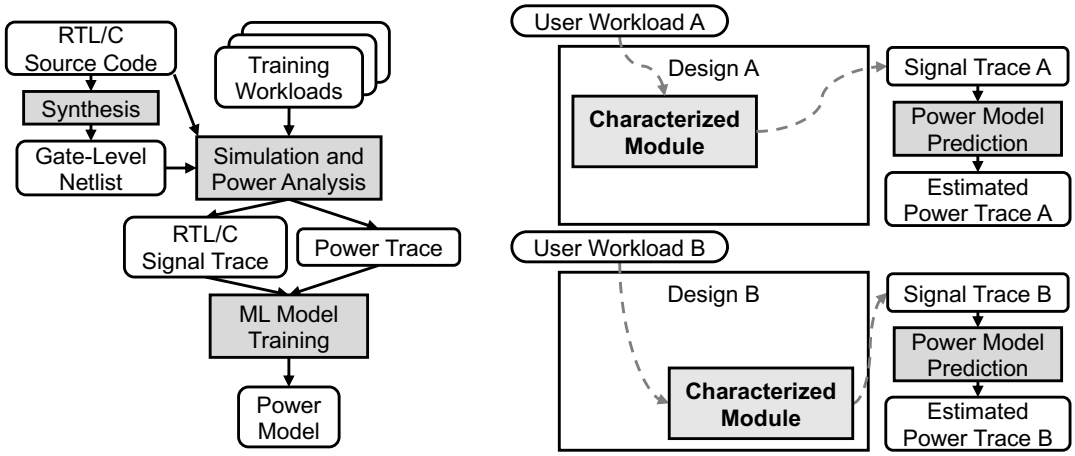
4. For RTL power estimation, we demonstrate that PRIMAL is at least 50x faster on average than PTPX for cycle-accurate power estimation with a small error. Notably, our CNN-based approach is 35× faster than PTPX with a 5.2% error for estimating the power of a RISC-V core. PRIMAL also achieves a 15× speedup over a commercial RTL power analysis tool for average power estimation.
5. For HLS power estimation, our LSTM-based approach offers an additional 3.5× speedup over the CNN-based RTL power estimation approach while achieving comparable estimation accuracy.

The RTL power estimation part of this chapter was published in DAC'19 [186]. The remainder of this chapter is organized as follows: Section 3.1 presents the overall methodology and intended use cases of PRIMAL. Section 3.2 introduces our feature construction methods. Experimental results for RTL and HLS power estimation are reported in Sections 3.3 and 3.4, respectively. Section 3.5 presents related works with additional discussions.

3.1 Methodology

Unlike previous works, PRIMAL uses state-of-the-art ML models for fast and accurate high-level power estimation. Figure 3.2 shows the two phases of the PRIMAL workflow[1]. The characterization phase (Figure 3.2a) requires an RTL/C model of the module, the gate-level netlist, and a set of training workloads. RTL register or C variable traces are used as the input features, while

[1]We assume C-based HLS design flows for HLS power estimation.



(a) Characterization phase. (b) Estimation phase.

Figure 3.2: Two phases of the PRIMAL workflow — Power models are trained once per module. Models can then be used across different workloads, as well as in different designs that instantiate the module.

ground-truth power numbers for the training workloads are obtained from gate-level power analysis. The characterization process only needs to be performed once per IP block. The trained power models can then be used to estimate power for different user workloads in the estimation phase as illustrated in Figure 3.2b.

It is important to note that the training workloads may be very different from the actual user workloads. For example, designers can use functional verification tests to train the power models, which then generalize to realistic workloads. By using state-of-the-art ML models, our approach accommodates diverse workloads and can model large, complex circuit blocks. The ML models are trained for cycle-by-cycle power estimation, which provides a more detailed power profile than average power and enables more effective design optimization.

3.1.1 RTL Power Estimation Methodology

For RTL power estimation, the RTL register traces are used as input features during the characterization phase. More specifically, we use register switching activities in the simulation traces as input features. Compared with using all signals from the RTL simulation trace, only using register switching activities significantly reduces input feature size, prevents overfitting, and still captures the complete current state of the circuit. In addition, the mapping between RTL signals and gate-level registers can be retrieved from the logic synthesis tool report. Because we use cycle-accurate power traces from gate-level simulation as ground truth, the ML models are essentially learning the complex relationship between the switching power of all gate-level cells and register switching activities.

In this work we explore a set of established ML models for RTL power estimation. The classical ridge linear regression model is used as a baseline. We also experiment with gradient tree boosting, a promising non-linear regression technique [104]. For linear models and gradient tree boosting models, we apply principal component analysis (PCA) [80] to the input data to reduce model complexity and avoid overfitting. We also study the efficacy of deep learning (DL) models, which are capable of approximating more complex nonlinear functions. Specifically, we experiment with multi-layer perceptron (MLP) and CNN for RTL power estimation. MLP contains only fully-connected network layers and is more compute-efficient than CNN. However, the parameter count of MLP grows quickly with respect to the feature size of the design, resulting in overfitting and training convergence issues. CNNs have shown impressive performance in image classification tasks. Thanks to the structure of convolu-

tional layers, the parameter count of CNNs does not increase significantly as input image size grows. As a result, CNN is a more scalable choice than MLP for large designs.

3.1.2 HLS Power Estimation Methodology

For HLS power estimation, since we rely on timed software simulation at cycle level, the RTL register information is no longer available. As a result, we use C variable traces as the input features to the ML models. We leverage the FLASH simulator [33] to generate C/C++ source code annotated with timing information. Using the timing information from the HLS synthesis report, FLASH accurately estimates the execution time and simulates FIFO communication cycle-accurately. Because FLASH abstracts the binding and allocation information of the computational statements, it is several orders of magnitude faster than the RTL co-simulation provided by current HLS tools. The software simulation trace is obtained by compiling the annotated source code together with the testbench and running the generated executable.

FLASH abstracts the binding and allocation information away to achieve speedup over RTL simulation. However, this abstraction also causes FLASH to simulate the computational statements in a cycle-approximate manner. In addition, the C variables do not have a one-to-one correspondence with RTL or gate-level signals, adding to the difficulty of accurately estimating power consumption. As a result, the values of the C variables in each cycle is only an incomplete and shifted approximation of the current state of the circuit, and may have poor correlation with the power of the circuit.

```
int mac(int A[N], int B[N]) {
  int result = 0;
  for (int i = 0; i < N; i ++ ) {
#pragma HLS pipeline II=1
    result += (A[i] * B[i]);
  }
  return result;
}
```

(a) C++ HLS code.

Cycle	A[i]	B[i]	result	i
0	—	—	—	—
1	3	5	15	1
2	0	3	15	2
3	-1	7	8	3
4	-4	-6	32	4
...	...	...	...	...

(b) FLASH simulation trace.

```
module mult(clk, ce, a, b, p);
  input clk;
  input ce;
  input[31:0] a;
  input[31:0] b;
  output[31:0] p;
  reg [31:0] a_reg0;
  reg [31:0] b_reg0;
  reg [31:0] buff0;
  wire [31:0] tmp;
  assign p = buff0;
  assign tmp = a_reg0 * b_reg0;
  always @ (posedge clk) begin
    if (ce) begin
      a_reg0 <= a;
      b_reg0 <= b;
      buff0 <= tmp;
    end
  end
endmodule
```

(c) Verilog code of the multiplier.

Figure 3.3: Pipelined integer MAC unit example — The HLS-generated multiplier has a two-cycle latency, so the `result` column of the FLASH-generated trace is not perfectly aligned with the output of the multiplier.

We illustrate this phenomenon using a motivational example. Consider the pipelined integer MAC unit shown in Figure 3.3a. HLS generates an integer multiplier for this design, which consumes a significant part of the total power. Figure 3.3c shows the Verilog code of the integer multiplier generated by Vivado HLS. It is clear that the multiplier has a two-cycle latency, and multiplication is performed in the middle of the pipeline. Therefore, the power of this multiplier not only depends on the inputs in the current cycle, but also depends on the inputs of the previous cycle. Figure 3.3b shows a FLASH simulation trace for this example. We can only observe the inputs and outputs of the multiplier but not the internal register values. Furthermore, notice that in Figure 3.3b the `result` variable changes in the same cycle as `A[i]` and `B[i]`, without the two-cycle latency. If we only allow an ML model to take the signals from the current cycle as input, any model that cannot leverage history information is unable to make accurate power estimations. The orange dashed curve in Figure 3.4 shows

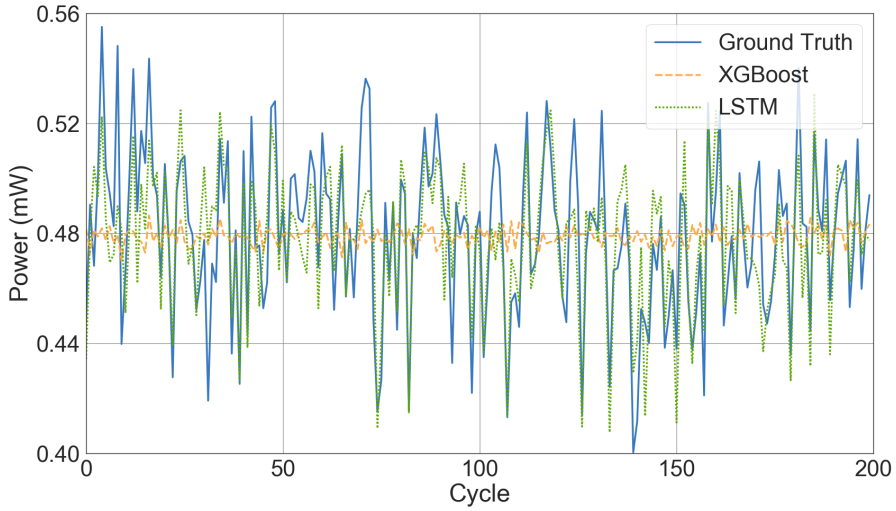


Figure 3.4: Ground truth and predicted power traces for the MAC unit — The LSTM prediction follows the trend of the ground truth, while XGBoost predicts average power.

the prediction of XGBoost [31], a tree-based regression model. Since the signals from the current cycle have poor correlation with the power consumption, the XGBoost model struggles to follow the ground truth and only predicts the average power.

To accurately estimate per-cycle power using C variable traces generated by FLASH, the ML model must be able to rectify the "mistakes" in the traces and use the C variable values from the history to estimate power of the current cycle. We propose to use LSTM [64] to achieve this goal. LSTM is a variant of RNN which has been successfully applied to linguistic tasks such as speech recognition, machine translation, and image captioning. The design of LSTM allows it to effectively capture long-term dependencies and relationships within a sequence, thus enabling LSTMs to make accurate power predictions from the C variable traces. As shown in Figure 3.4, the LSTM makes better predictions for the pipelined MAC unit because it can leverage history information. When updating its hidden state, the LSTM can compensate for the shifting behavior

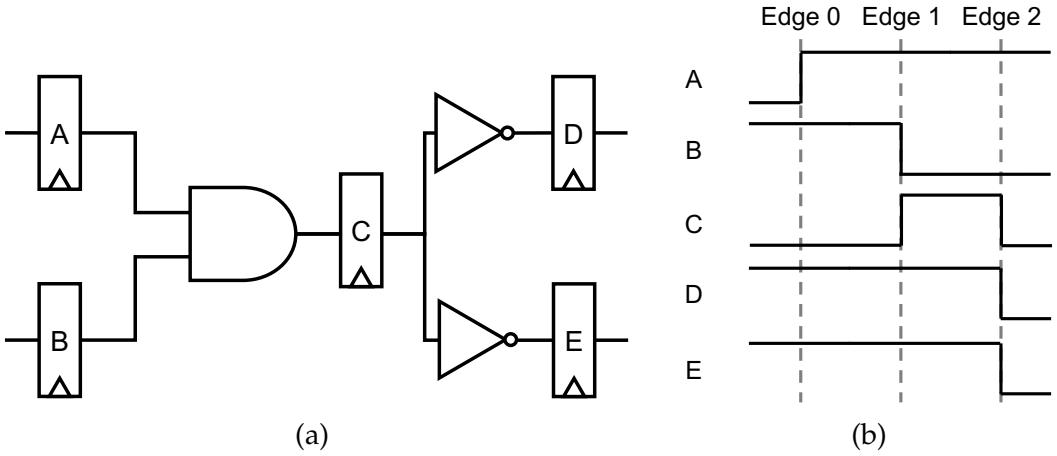


Figure 3.5: Example circuit and waveform for illustrating feature construction methods — **(a)** Sequential logic with five registers and three gates; **(b)** Waveform of register output values.

in the FLASH traces and infer the internal state of the multiplier.

3.2 Feature Construction

This section describes the feature construction procedure using the circuit in Figure 3.5a as an example. An example waveform is shown in Figure 3.5b. The discussion in Sections 3.2.1 and 3.2.2 is under the RTL power estimation context. Section 3.2.3 describes how we extend the feature construction methods to HLS power estimation.

3.2.1 Feature Encoding for Cycle-by-Cycle Power Estimation

For cycle-by-cycle power estimation, we use RTL register traces as input features without any manual feature selection. Both internal register traces and I/O signal traces are required to capture all circuit states. A good feature encod-

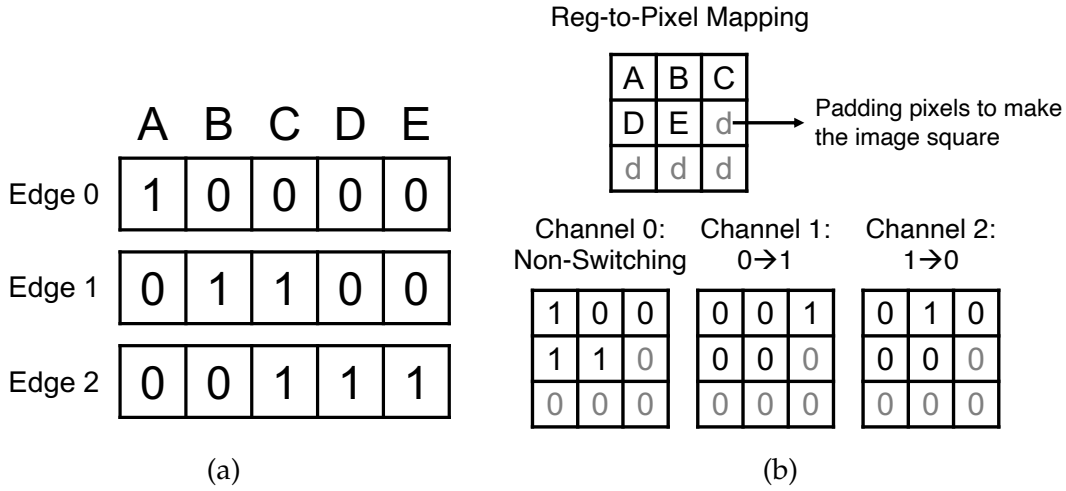


Figure 3.6: Default 1D and 2D feature encoding for the circuit and waveform in Figure 3.5 — **(a)** 1D switching encoding for three clock cycles; **(b)** Default 2D encoding for edge 1.

ing of the simulation traces should capture the switching activities and differentiate between switching and non-switching events. A concise encoding, which we refer to as *switching encoding*, is to represent each register switching event as a 1, and non-switching event as a 0. For an RTL module with n registers, each cycle in the RTL simulation trace is represented as a $1 \times n$ vector. Figure 3.6a shows the corresponding encoding for the waveform in Figure 3.5b. Each vector in Figure 3.6a represents one clock rising edge in the waveform. We use this 1D switching encoding for all but the CNN models. The same feature encoding is used in [169].

In order to leverage well-studied two-dimensional (2D) CNN models, we create a three-channel 2D image representation for every clock rising edge in the register trace. For an RTL module with n registers, we use a $\lceil \sqrt{n} \rceil \times \lceil \sqrt{n} \rceil \times 3$ image to encode one clock rising edge in the RTL simulation trace. We use one-hot encoding in the channel dimension to represent the switching activities of each register: non-switching is represented as $[1, 0, 0]$, switching from zero to

one is represented as $[0, 1, 0]$, and switching from one to zero is $[0, 0, 1]$. We refer to this encoding as *default 2D encoding*. Figure 3.6b shows how we encode **edge 1** of the waveform shown in Figure 3.5b. Since the total number of pixels in the image can be greater than n, the pixels shown as d's are paddings which do not represent any register. In our implementation, the padding pixels have zero values in all three channels. Every other pixel corresponds to one register in the module. For this default 2D encoding, the registers are mapped by their sequence in the training traces. For example, since in Figure 3.5b the order of registers is A, B, C, D, and E, in each channel the top-left pixel in Figure 3.6b corresponds to A, the top-right pixel is mapped to C, and the center pixel refers to E.

3.2.2 Mapping Registers and Signals to Pixels

In the default 2D encoding described above, the mapping between registers and pixel locations are determined by the way the registers are arranged in the trace file. This mapping method does not guarantee any meaningful local structure in the constructed images: Registers that are mapped to adjacent pixels may not be correlated or physically connected.

CNNs are most effective when there are spatial relationships in their 2D inputs. As a result, it is natural to exploit graph topology information in the gate-level netlist so that the register-to-pixel mapping can reflect the connectivity or even physical placement of the registers. Since the gate-level netlist of the design is available during the characterization phase, we use the outputs of logic synthesis tools to map RTL signals to netlist nodes and construct the

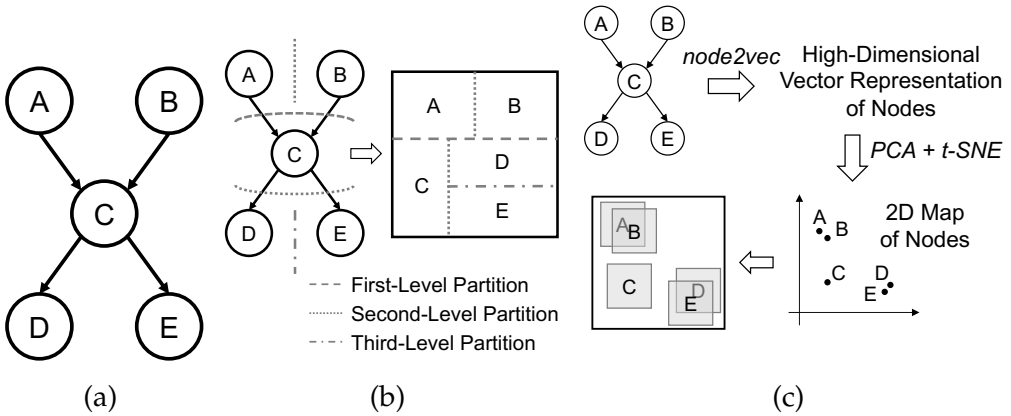


Figure 3.7: Graph-based register-to-pixel mapping methods — **(a)** The register connection graph generated from the circuit in Figure 3.5a; **(b)** Register mapping based on graph partitioning. The register connection graph is recursively partitioned into two parts. Each partition also divides the map into two non-overlapping parts; **(c)** Register mapping based on graph node embedding. The coordinates of each mapped register are generated by node2vec followed by dimensionality reduction techniques. In the generated mapping each register occupies a unit square.

graph. Since we only use register traces as our features, we ignore all combinational components in the gate-level netlist and only extract register connection graphs when processing the gate-level netlist. The CNN models are expected to "learn" the information of combinational gates between each pair of registers. The graph constructed for the example circuit in Figure 3.5a is shown in Figure 3.7a. Each node in the graph corresponds to one register in the design.

We propose two graph-based methods for generating register-to-pixel mappings. The first method is based on graph partitioning, in which the graph is recursively divided into two partitions of similar sizes, and the partitions are mapped to corresponding regions in the image (see Figure 3.7b). The area allocated for each partition is computed according to the number of nodes in the partition. The second method is based on graph node embedding as shown in Figure 3.7c. We apply node2vec [57], a popular graph node embedding

technique, to map nodes in the register connection graph into a vector space. PCA [80] and t-SNE [103] are used to further reduce the dimensionality of the vector representation to two. The resulting 2D vector representations are scaled according to the image size, and indicate the mapping locations of the registers.

These two methods introduce local structures into the images according to the structural similarities between nodes. We still use the channel-wise one-hot encoding for every register when we apply the graph-based mapping methods. However, with the two graph-based methods, the area of each pixel can overlap with the area occupied by multiple registers. In such cases, for every channel, each register's contribution to the pixels is proportional to the overlapping area of the register's occupied space and the pixel.

3.2.3 Feature Construction for HLS Power Estimation

Feature Encoding The same feature encoding methods described in Sections 3.2.1 and 3.2.2 can be directly applied to HLS power estimation if we decompose each signal recorded in the cycle-level simulation trace into single bits. In our experiments we use switching encoding for the LSTM models. As a result, given a simulation trace that is T cycles long and all signals sum up to N bits, the encoded trace is a $(T-1)\times N$ matrix where each row is a $1\times N$ binary vector. Despite increasing the input dimensionality, this encoding method incurs minimal information loss compared to more compact encoding methods.

Handling Unobserved Variables in the Simulation Trace Software simulators cannot guarantee to track the value of every variable in every cycle because

the variables can only be tracked when their corresponding instructions are executed. For instance, the variables inside a not-taken branch will not be observed. In the absence of a loop, when the program has finished certain instructions the outputs of those instructions will no longer be tracked. In this work, whenever the value of a variable is unobserved from the trace, we assume it *maintains the previously observed value*. We believe this is a reasonable assumption, because the logic corresponding to the unobserved variables is most likely idle in this case.

3.3 Experimental Results for RTL Power Estimation

The proposed RTL power estimation framework is implemented in Python 3.6, leveraging networkx [58], metis [82], and a node2vec package [57]. MLP and CNN models are implemented using Keras [3]. Other ML models are realized in scikit-learn [121] and XGBoost [31]. We conduct our experiments on a server with an Intel Xeon E5-2630 v4 CPU and a 128GB RAM. We run neural network training and inference on a NVIDIA 1080Ti GPU. We use Synopsys Design Compiler for RTL and logic synthesis, targeting a 16nm FinFET standard cell library. The RTL register traces and gate-level power traces are obtained from Synopsys VCS and PTPX, respectively. Gate-level power analysis is performed on another server with an Intel Xeon CPU and 64GB RAM using a maximum of 30 threads.

3.3.1 Benchmarks

Table 3.1 lists the benchmarks used to evaluate the efficacy of PRIMAL for RTL power estimation. Our benchmarks include a number of fixed- and floating-

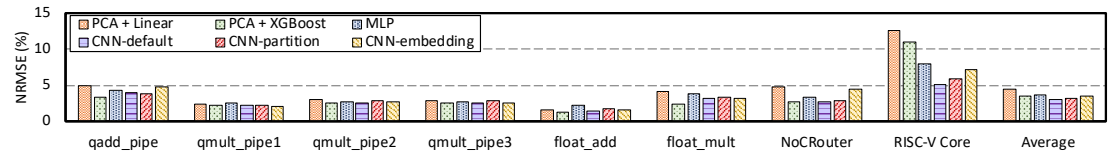
Table 3.1: Benchmarks for RTL power estimation — We evaluate PRIMAL with a diverse set of benchmark designs. For NoC router and RISC-V core, the test sets are realistic workloads which are potentially different from the corresponding training set.

Design	Description	Register + I/O signal count	Gate count	PTPX throughput (cycles/s)	Training set (# cycles)	Test set (# cycles)
qadd_pipe	32-bit fixed-point adder	160	838	1250	Random stimulus (480k)	Random stimulus (120k)
qmult_pipe {1, 2, 3}	32-bit fixed-point multiplier with 1, 2, or 3 pipeline stages	{384, 405, 438}	{1721, 1718, 1749}	{144.9, 135.1, 156.3}	Random stimulus (480k)	Random stimulus (120k)
float_adder	32-bit floating-point adder	381	1239	714.3	Random stimulus (480k)	Random stimulus (120k)
float_mult	32-bit floating-point multiplier	372	2274	454.5	Random stimulus (480k)	Random stimulus (120k)
NoCRouter	Network-on-chip router for a CNN accelerator	5651	15076	44.7	Unit-level testbenches (910k)	Convolution tests (244k)
RISC-V Core	RISC-V Rocket Core (`SmallCore`)	24531	80206	45	RISC-V ISA tests (2.2M)	RISC-V benchmarks (1.7M)

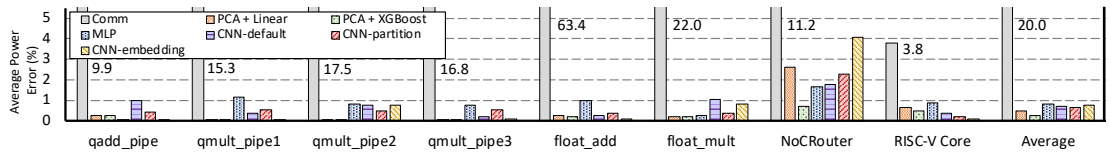
point arithmetic units from OpenCores [118]. We also test our approach against two complex designs — a NoC router used in a CNN accelerator and a RISC-V processor core. The NoC router block is written in SystemC and synthesized to RTL by an HLS tool. The RISC-V core is an RV64IMAC implementation of the open-source Rocket Chip Generator [15] similar to the `SmallCore` instance. We use different portions of random stimulus traces as training and test sets for the arithmetic units. For the NoC router and the RISC-V core, we select functional verification testbenches for training and use realistic workloads for testing. For the NoC router, we test on actual traces of mesh network traffic from a CNN accelerator SoC. In the RISC-V experiment, `dhrystone`, `median`, `multiply`, `qsort`, `towers`, and `vvadd` form the set of test workloads.

3.3.2 Results

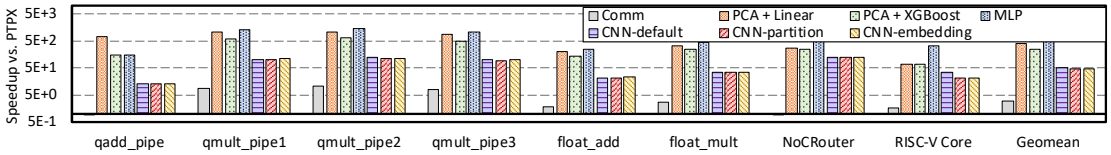
Figure 3.8 summarizes the results for RTL power estimation. Here we use RTL register traces as the raw input, and apply the feature construction techniques



(a) Cycle-by-cycle estimation error.



(b) Average power estimation error.



(c) Speedup vs. Synopsys PTPX.

Figure 3.8: Performance of different machine learning models on test sets — The ML models used by PRIMAL achieve high accuracy for both cycle-by-cycle and average power estimation, while offering significant speedup against both Synopsys PTPX and a commercial RTL power analysis tool (`Comm`). PRIMAL is also significantly more accurate than `Comm` in average power estimation.

Table 3.2: Training time of different ML models.

Design	**PCA**	**Ridge Regression**	**XGBoost**	**MLP**	**CNN**
arithmetic units	~10 min	~1 min	~15 min	~25 min	~3 h
NoCRouter	~7 h	~15 min	~1 h	~1.5 h	~10 h
RISC-V Core	~20 h	~30 min	~1.5 h	~7 h*	~20 h*

* A random 50% of training data is used per training epoch.

described in Section 3.2. Two percent of the training data is used as a validation set for hyper-parameter tuning of the ML models. They are also used for early stopping when training the deep neural networks. All models except CNNs use the 1D switching encoding, while CNNs use the 2D image encoding methods introduced in Section 3.2. For ridge regression and gradient tree boosting, we apply PCA to reduce the size of input features to 256, except for `qadd_pipe` which has only 160 features with 1D switching encoding. We use three-layer MLP models for the arithmetic unit and four-layer MLP models for the NoC router

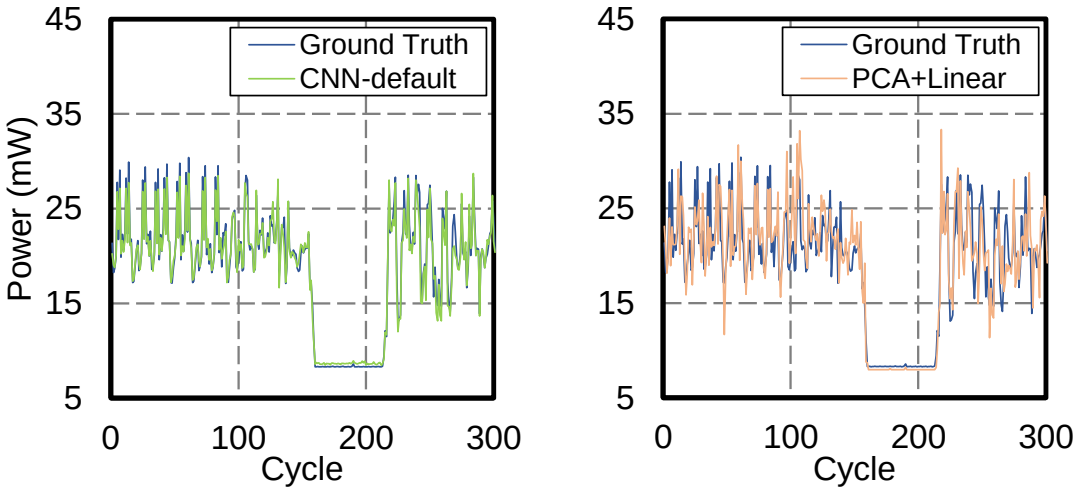


Figure 3.9: Ground truth vs. CNN-default and PCA+Linear for RISC-V — Showing 300 cycles from the `dhrystone` benchmark.

and the RISC-V core. We use an open-source implementation [2] of ShuffleNet V2 [100] for CNN-based power estimation because of its parameter-efficient architecture and fast inference speed. The *v0.5* configuration in [100] is used for the arithmetic units, while the *v1.5* configuration is used for the NoC router and the RISC-V core. `CNN-default`, `CNN-partition`, and `CNN-embedding` in Figure 3.8 refer to the default 2D encoding, graph-partition-based register mapping, and node-embedding-based register mapping methods introduced in Section 3.2, respectively.

Cycle-by-Cycle Power Estimation Results We use normalized root-mean-squared-error (NRMSE) as our evaluation metric. Suppose the ground-truth power trace is represented as a n-dimensional vector $\mathbf{y}$, and the estimated power trace is a vector $\hat{\mathbf{y}}$ of the same shape. Then

$$NRMSE = \frac{1}{\bar{\mathbf{y}}} \sqrt{\frac{\sum_{i=1}^{n} (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{n}}$$

As shown in Figure 3.8a, all ML models can achieve an average estimation error of less than 5% across our benchmarks. The training time for each ML model is

summarized in Table 3.2. For small designs, XGBoost offers competitive accuracy with much less training effort. CNN models show significant advantages over other ML models for larger designs like the RISC-V core. Notably, our CNN model with default 2D encoding achieves an impressive 5.2% error on the test set, while MLP, XGBoost and Linear models achieve around 8%, 11% and 13% error, respectively. Figure 3.9 compares the estimation of `CNN-default` and `PCA+Linear` with the ground truth power trace of the RISC-V core. The CNN estimation fits the ground truth curve more closely. These results demonstrate the superior capability of deep neural networks in approximating complex nonlinear functions.

We observe that the graph-based register mapping methods do not provide much benefit over default 2D encoding. In fact, the advanced encodings result in a lower accuracy on some benchmarks. One possible reason is that with the graph-based mapping methods, the information of multiple registers can be aggregated into one single pixel, making the feature representation more convoluted and hard for the CNN models to decipher. Our experiments show that CNN models already have enough complexity to learn a favorable internal representation with the default encoding scheme. However, a graph-based image representation that is relatively stable with respect to small circuit changes might be useful if we want the trained models to quickly adapt to the changes with a small amount of training data.

Average Power Estimation Results The average power consumption for a workload can be easily obtained from a cycle-accurate power trace. We compare the ML-based techniques with a commercial RTL power analysis tool (`Comm`). According to Figure 3.8b, all of the ML techniques achieve less than 1% aver-

age error across the eight benchmarks, while the commercial tool has an average error of 20%. The worst case is NoC router, where all models except for `PCA+XGBoost` have more than 1.5% error, and CNN with graph-embedding-based feature encoding has around 4% error. The reason is that different operational modes of the NoC router have drastically different power characteristics, causing our training set to have very large power variations. On the other hand, the test set contains a lot of low-power, idle cycles which are not common in the training set. As a result, for our test set the average power error is highly dependent on how well the models estimate idle power. The linear model does not have enough complexity to estimate both active and idle power with high accuracy, while the DL models are affected by the limited number of idle samples in the training set. The ML models require a significant amount of time to be trained for complex designs as shown in Table 3.2. Therefore, the commercial RTL power analysis tool is still favorable for power estimation of non-reusable modules.

Speedup Figure 3.8c presents the speedup of `Comm` and the PRIMAL techniques against Synopsys PTPX. Notice that for PRIMAL, the reported speedup is for model inference only, which is the typical use case. While `Comm` is only around 3× faster than PTPX on average, all ML models achieve much higher estimation speed. Even the most compute-intensive CNN models provide around 50× speedup over PTPX. The linear model, XGBoost and MLP has an additional 8×, 5× and 10× speedup over CNNs, respectively. Note that the linear and XGBoost models are executed on a multi-core CPU, while MLP and CNN inference is performed on a single GPU. As a result, if all models can be efficiently migrated to GPUs, or if more advanced CPU and GPU platforms are available,

Table 3.3: Accuracy and speedup of PRIMAL-RTL , FLASH-CNN, and PRIMAL-HLS against gate-level power analysis on test sets — We use `mmul` and `nw` from the FLASH paper [33], and `3d rendering`, `spam filtering`, and `digit recognition` from the Rosetta benchmark suite [185]. Both the simulation time and the time used for power estimation are included when computing the throughput. Simulation and gate-level power analysis are performed with a single thread.

Benchmark	Gate-Level Power Analysis Throughput (Cycles/s)	PRIMAL-RTL				FLASH-CNN				PRIMAL-HLS			
		Training Time	NRMSE	R^2	Speedup	Training Time	NRMSE	R^2	Speedup	Training Time	NRMSE	R^2	Speedup
mmul	149.4	103 min	1.72%	0.99	54.3×	108 min	3.64%	0.96	81.8×	24 min	3.81%	0.96	126.2×
nw	124.2	45 min	0.47%	0.99	69.2×	65 min	0.42%	0.99	42.6×	20 min	0.41%	0.99	131.3×
3d rendering	35.4	147 min	7.73%	0.88	66.6×	61 min	20.2%	0.18	246.1×	30 min	4.39%	0.96	443.6×
spam filtering	26.6	610 min	2.00%	0.99	65.0×	199 min	29.8%	0.87	360.9×	41 min	2.55%	0.99	527.1×
digit recognition	19.0	304 min	1.16%	0.99	50.4×	110 min	20.8%	0.58	88.1×	36 min	12.3%	0.85	116.4×
Average	50.6	242 min	2.64%	—	60.7×	109 min	15.0%	—	122.2×	30 min	4.70%	—	214.2×

1. CNN models are trained for 20 epochs with the Adam optimizer [83], batch size of 64, learning rate of 1e-3, and weight decay of 1e-4. 5% of training data is used for validation.
2. LSTMs are trained for 50 epochs with Adam, batch size of 64, learning rate of 1e-3, weight decay of 1e-4, and 20% dropout [137]. 10% of training data is used for validation. An additional 20% dropout is added to the input layer for `digit recognition` to avoid overfitting.

more speedup can be expected with a modest hardware cost. For small designs, the linear model and XGBoost are almost always more favorable choices, since the neural network models do not provide significant accuracy improvement but require much more compute and training effort. For complex designs such as the RISC-V core, CNN provides the best accuracy with around 35× speedup, while other models are faster but less accurate.

3.4 Experimental Results for HLS Power Estimation

The proposed HLS power estimation approach (referred to as **PRIMAL-HLS** in the rest of this section) is evaluated using a set of realistic benchmarks from the FLASH paper [33] and the Rosetta benchmark suite [185]. For each benchmark, the training and test set are complete execution runs of the accelerators with different inputs. We implement the LSTM power model in PyTorch [119]. We use

Vivado HLS 2018.2 for high-level synthesis, targeting a Kintex UltraScale device (`xcku060-ffva1156-2-e`) and 250MHz clock frequency. We then synthesize the generated RTL designs into gate-level netlists with Synopsys Design Compiler, targeting a TSMC 28nm standard cell library, an ARM SRAM compiler, and 250MHz clock frequency. Synopsys VCS and PTPX are used to run gate-level simulation and generate per-cycle power traces. It is important to note that we have to use ASIC CAD tools after FPGA HLS to obtain cycle-accurate power analysis because similar capabilities are currently not offered by the FPGA vendors. We do not intend to use this proof-of-concept tool flow for actual ASIC implementation.

To evaluate the accuracy degradation against RTL power estimation, we compare against the CNN-based RTL power estimation approach described in Section 3.1.1, which is referred to as **PRIMAL-RTL** in the rest of this section. To illustrate the effectiveness of LSTM models over other ML models that cannot leverage history information, we also compare with a baseline approach referred to as **FLASH-CNN**. The only difference between FLASH-CNN and PRIMAL-HLS is that FLASH-CNN uses CNNs and the default 2D encoding described in Section 3.2.1. The CNN models we use in our experiments are similar to ResNet-18 [62]. Simulation and gate-level power analysis are performed on a machine with a 2.40GHz Intel Xeon CPU and 256GB memory. CNN/LSTM training and inference are performed on a machine with a 2.20GHz Intel Xeon CPU, 256GB memory, and one NVIDIA RTX 2080Ti GPU.

Estimation Accuracy Table 3.3 shows the estimation accuracy and throughput of different approaches. We use two metrics to evaluate the estimation accu-

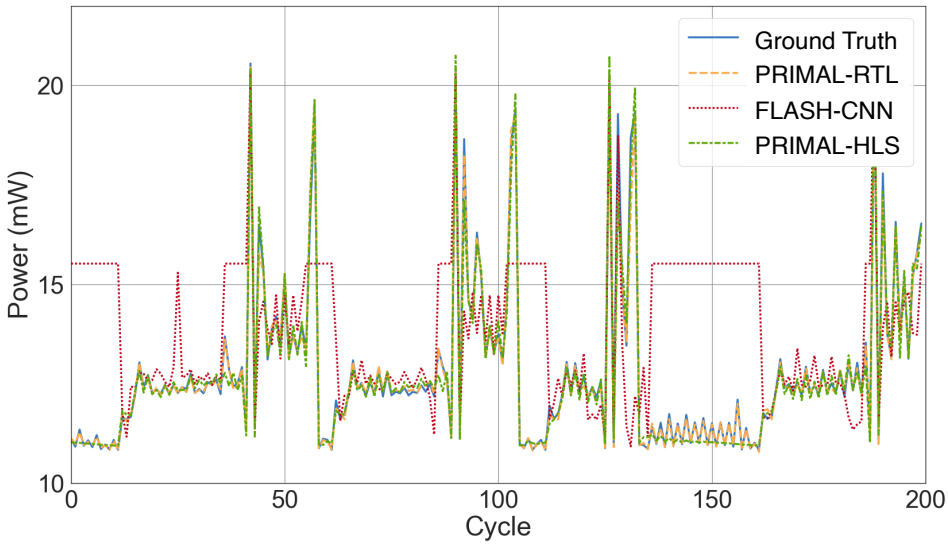


Figure 3.10: 3D rendering trace comparison.

racy: NRMSE and R^2 score[2]. NRMSE shows the normalized average difference between the estimated power trace and the ground-truth power trace, while the R^2 score illustrates the correlation between the estimation and the ground truth.

While all three techniques achieve similar accuracy for `mmul` and `nw`, PRIMAL-HLS is significantly more accurate than the FLASH-CNN baseline on `3d rendering`, `spam filtering`, and `digit recognition`. On `3d rendering` and `spam filtering`, the predictions from PRIMAL-HLS have similar accuracy as the ones generated by PRIMAL-RTL and are very close to the ground truth power traces from gate-level power analysis. Figure 3.10 compares the predictions of different approaches against the ground-truth power trace for `3d rendering`. The predictions of PRIMAL-RTL and PRIMAL-HLS can closely follow the ground truth, while FLASH-CNN often fails to follow the trend and predicts average power.

As mentioned in Section 3.1.2, PRIMAL-HLS should be more accurate than

[2]Assume the ground-truth power trace is represented as an n-dimensional vector $\mathbf{y}$ and the estimated power trace is a vector $\hat{\mathbf{y}}$ of the same length, then $R^2 = 1 - \frac{\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{\sum_{i=1}^{n}(\mathbf{y}_i - \bar{\mathbf{y}}_i)^2}$.

FLASH-CNN because CNNs cannot leverage history information that is important for HLS power estimation. Interestingly, FLASH-CNN achieves good accuracy on `mmul` and `nw`. By examining these two designs, we found that both designs have very regular control flow. The power of these designs are significantly affected by the state they are in rather than the exact changes of the signals. In addition, due to the lack of diversity in the control flow, the control pattern of the test data is very close to the training data. Since the CNN models could overfit and memorize the control pattern during training, they are able to offer accurate predictions on the test set as well. On the contrary, the other three benchmarks have more complicated control flow and deeper pipelines, causing the schedule assumed by FLASH to significantly differ from the actual schedule. With the more complex control patterns, FLASH-CNN cannot correctly identify the state transitions of the designs according to the information from the current cycle, while PRIMAL-HLS is capable of recovering the behavior of the original schedule.

Speedup Table 3.3 also shows the throughput of gate-level power analysis for every benchmark, as well as the speedup of each technique over gate-level power analysis. On average, PRIMAL-HLS offers 214.2× speedup over gate-level power analysis, and 3.5× speedup over PRIMAL-RTL. The speedup over PRIMAL-RTL results from both faster simulation and faster ML inference. Our results show that FLASH simulation is on average 10.3× faster than RTL simulation. In addition, LSTM inference in PRIMAL-HLS is on average 3.8× faster than the CNN inference in PRIMAL-RTL. Despite being harder to parallelize, the LSTM models require much less computation during inference because of the model architecture and the reduced input size. The training time of PRIMAL-

HLS is also much shorter than FLASH-CNN and PRIMAL-RTL.

3.5 Related Work and Discussions

Related Work on Power Estimation Behavioral-level power estimation provides optimization guidance early in the design flow. In an earlier work, Chen et al. [29] combine profiling and simple analytical models to estimate FPGA power consumption. Ahuja et al. [9] propose an HLS-based methodology, which relies on the HLS tool to provide mappings between system-level variables and RTL signals, and uses RTL power estimation tools to predict average power. Aladdin [132] characterizes the power of different sub-components through micro-benchmarking, performs sub-component scheduling to introduce timing information into untimed C descriptions, and runs gate-level simulation to estimate switching activities and average power. In a more recent work [92], Lee et al. back-annotate micro-architectural information into the LLVM [90] intermediate representation using information from the HLS tool. By running the annotated program, timed variable traces can be obtained through software simulation alone. The variable traces are then used to train a separate ML model for each control step for predicting cycle-by-cycle power.

While behavioral-level power estimation techniques can provide fast estimations at a high abstraction level, their results can only be used for early architectural-level explorations due to the lack of low-level details. Most existing techniques can only provide average power estimation. More implementation details are available at RTL. Earlier works in RTL power estimation use simple regression models, such as linear regression and regression trees, to char-

acterize small circuit design blocks [12, 21, 126]. The regression models are trained with gate-level power analysis results. Average power, or even cycle-by-cycle power of the whole design, can be obtained by summing up the outputs from multiple models. More recent works in RTL power estimation try to characterize larger modules or even the complete IP block. PrEsto [140] uses linear regression models for this purpose, and applies heavy feature engineering and feature selection to reduce the complexity of power models. A more recent work by Yang et al. [169] uses a single linear model to characterize the whole design. A feature selection technique based on singular value decomposition (SVD) is applied to reduce model complexity so that the regression model can be efficiently mapped onto an FPGA. Both PrEsto and [169] can provide cycle-by-cycle power estimates.

Power Estimation for SystemC SystemC [1] is a powerful framework based on C++. It is flexible enough to support hardware design at different abstraction levels. Designers can either write behavioral descriptions of the hardware modules and let SystemC-based HLS tools generate the RTL, or specify the cycle-level behavior of the hardware in detail. Depending on the coding style and the simulation tool, traces generated from SystemC simulation can be either cycle-accurate or transaction-accurate. The techniques for HLS power estimation presented in Section 3.1.2 can be directly applied to SystemC power estimation and generate cycle-by-cycle power traces, because the LSTM model is able to tolerate the shifting behavior in the traces.

We explored an alternative in our DAC'19 submission for handling inaccuracies in SystemC traces. The main idea is to train the ML models to estimate the average power inside fixed-size time windows instead of per-cycle power. We

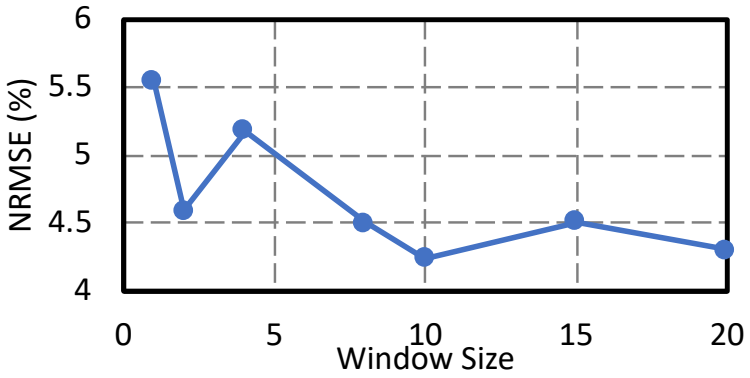


Figure 3.11: SystemC power estimation accuracy of NoCRouter using a VGG16 CNN model.

perform an element-wise sum of the encoded features inside the time window and use the results for ML model training and inference.

We use the same NoC router design shown in Section 3.3 as our case study. For this design, the SystemC variable trace can differ from the RTL trace by up to seven cycles. We continue to employ functional verification testbenches as our training set, and test on 3.5k cycles of chip-level convolution testbenches. A VGG16 model [136] pre-trained on ImageNet [130] is fine-tuned for window-by-window estimation. The default 2D encoding is used for this experiment. The estimation accuracy of the VGG16 model with different window sizes is shown in Figure 3.11. While the CNN model performs reasonably well for small window sizes, there is a clear error decrease when the window size is larger than seven as the effect of trace inaccuracy is mitigated. When the window size is larger than 8, the CNN model is able to achieve less than 4.5% error, which is satisfactory in most cases. Nevertheless, the error of SystemC power estimation remains higher than that of RTL power estimation because of the trace inaccuracy and the information loss in the feature construction process.

CHAPTER 4

CIRCUIT DISTILLATION: DISTILLING ARBITRATION LOGIC FROM TRACES USING MACHINE LEARNING

Resource sharing is common in modern computer systems to balance performance and hardware cost. In order to maximize utilization and achieve a high performance, the hardware needs to frequently make arbitration decisions to allocate access to shared resources on the fly. The design of an effective arbitration unit often involves intricate trade-offs amongst performance, area, and power. Traditionally, such arbitration policies and circuits are almost exclusively designed by humans. However, with the current and future generations of computer architectures becoming increasingly complex and heterogeneous, it is now much more difficult for humans to devise efficient heuristics that can account for information from various parts of the system.

The recent advances in machine learning (ML) provide an opportunity to overcome this challenge. Using ML techniques, an effective heuristic can be learned by the model from a sufficient amount of data. Early attempts along this line investigated perceptron branch predictors [53, 79] and memory controllers using table-based reinforcement learning (RL) algorithms [76, 110]. More recently, there is an emerging trend of applying deep learning (DL) to tackle the decision making problems in computer architecture, such as cache replacement [133], prefetching [60, 174], network-on-chip (NoC) packet arbitration [171], and NoC dynamic frequency-voltage scaling [182]; in many cases, DL techniques have been shown to achieve a superior performance in simulation. However in most cases, it is not feasible to directly use a DL accelerator as an arbitration unit due to its high overhead in both latency and area. As a

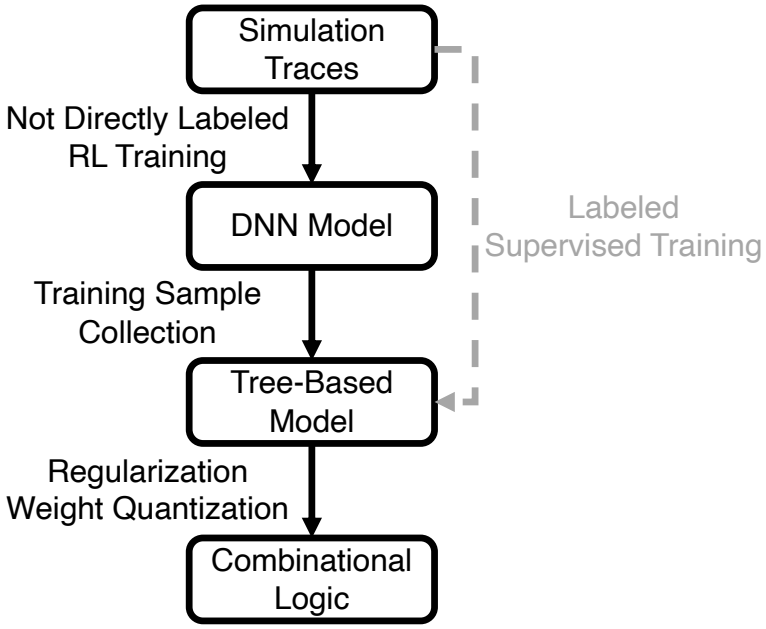


Figure 4.1: Proposed flow of distilling logic from traces — Tree models are used to bridge the implementation gap between neural network models and logic. In case of a supervised task, tree models can be directly trained from simulation traces. The discussion of this paper focuses on converting DL models to circuits.

result, feature engineering and manual analysis of neural network models are necessary to convert DL models to affordable arbitration logic implementations. While such manual conversion is effective for small neural networks, it quickly becomes intractable when the model becomes complicated and hard to interpret. A fully automated conversion step is needed to fill the missing link of applying DL to arbitration problems in computer architecture.

In this chapter, we propose Circuit Distillation, which tackles this challenge by leveraging tree-based models as a bridge between neural network models and circuit implementations. Figure 4.1 outlines the proposed approach, where tree-based models are trained using the outputs of a pre-trained DL model. Since tree models can be easily converted to circuits, the arbitration logic can be directly "distilled" from simulation traces. This flow is very suitable for learn-

ing arbitration logic because arbitration policies can be effectively learned using deep RL. Under this scenario, labels are not available during the training process, and the DL model learns to predict the potential reward (or priority score) of each legal action. In such cases, converting DL models into tree models improves the interpretability of the learned policy, because designers can examine the tree models and check whether the policy complies with their experience. Depending on the exact problem setup, CART trees [23], random forests [63], or model trees [89] can be used to approximate the output of the DL model.

We believe our approach can potentially be applied to many decision-making problems in computer systems. In this work, we specifically focus on on-chip networks and present a detailed case study on the NoC arbitration problem. NoC arbitration is a well-defined problem, and a good arbitration policy is critical for fairness, bandwidth utilization, and performance [93, 124]. In addition, the arbitration logic in a NoC router is subject to stringent area and latency constraints, so it is necessary to generate efficient and high-performance arbitration logic. Our major technical contributions are fourfold:

1. We are the first to propose a methodology for automatically generating compact, application-specific arbitration logic from simulation traces.
2. We present a case study on NoC packet arbitration and comprehensively analyze the learned arbitration policy. Specifically, we found that linear model trees are very suitable for this task and can be converted to compact arbitration logic.
3. The learned arbitration policy achieves up to 64× reduction in average packet latency and 4.9% increase in network throughput over the FIFO arbitration policy on the training traffic, and is able to generalize to different

injection rates. Compared with the DL agent, the generated arbitration logic achieves comparable performance with up to 282× area reduction.

4. We investigate how well the learned arbitration policy generalizes to other traffic patterns. Our experiments demonstrate the need for automatically generating reconfigurable arbitration units.

The rest of this chapter is organized as follows: Section 4.1 introduces some background on the NoC arbitration problem. Details of the Circuit Distillation approach are introduced in Section 4.2. In Section 4.3, we show our case study on NoC arbitration and demonstrate the efficacy of Circuit Distillation on this problem. Section 4.4 discusses related work on the application of ML in computer systems, learning Boolean logic using ML techniques, and efficient implementations of DNN models.

4.1 NoC Arbitration Background

In a NoC, routers are interconnected through links. A NoC router consists of multiple input and output ports. Within each input port, one or more virtual channels (VCs) are used to store incoming packets. NoC arbitration occurs when packets from multiple input VCs compete for the same output port. The arbitration logic determines which VC is given the priority to use the output port in cases of contention. NoC arbitration policy is critical for the network's performance—a good policy provides better fairness and achieves low average packet latency as well as high network throughput, while a sub-optimal policy could buffer an old packet in the network for a long time, resulting in poor network performance. Round-robin arbitration is a commonly used pol-

icy that guarantees fairness in scheduling by treating each input port and input VC equally. However, it only considers local fairness for individual routers, and therefore provides insufficient equality of service (i.e., link bandwidth allocation becomes more unfair the longer the routes are). Global-age-based arbitration prioritizes the packet with the oldest age, thereby providing global fairness and reducing the variance in packet transit time. Although global-age policy is considered one of the best policies, its hardware cost is largely impractical for use in on-chip routers [171].

NoC arbitration is a well-defined problem suitable for RL. Yin et al. present a case study on learning NoC arbitration policy with RL, where the RL agent predicts a priority score for each packet based on information including the packet's local age, payload size, and traversed hop count [171]. Since the input space of the RL agent is concise, it is easier to analyze and understand why the agent makes a certain decision. In this paper, we use a similar setup to evaluate and analyze the arbitration logic generated by our approach.

4.2 Distilling Arbitration Logic from Data

In this section, we introduce the details of our logic distillation process. As shown in Figure 4.1, starting from running simulation, an RL agent is trained to perform arbitration. After the agent learns an effective policy, the corresponding tree model is trained using the agent's outputs as labels, and the trained tree model is then converted to combinational logic. The area, power, and timing of the generated arbitration logic can be evaluated by ASIC tools, while the performance is evaluated through software or RTL simulation.

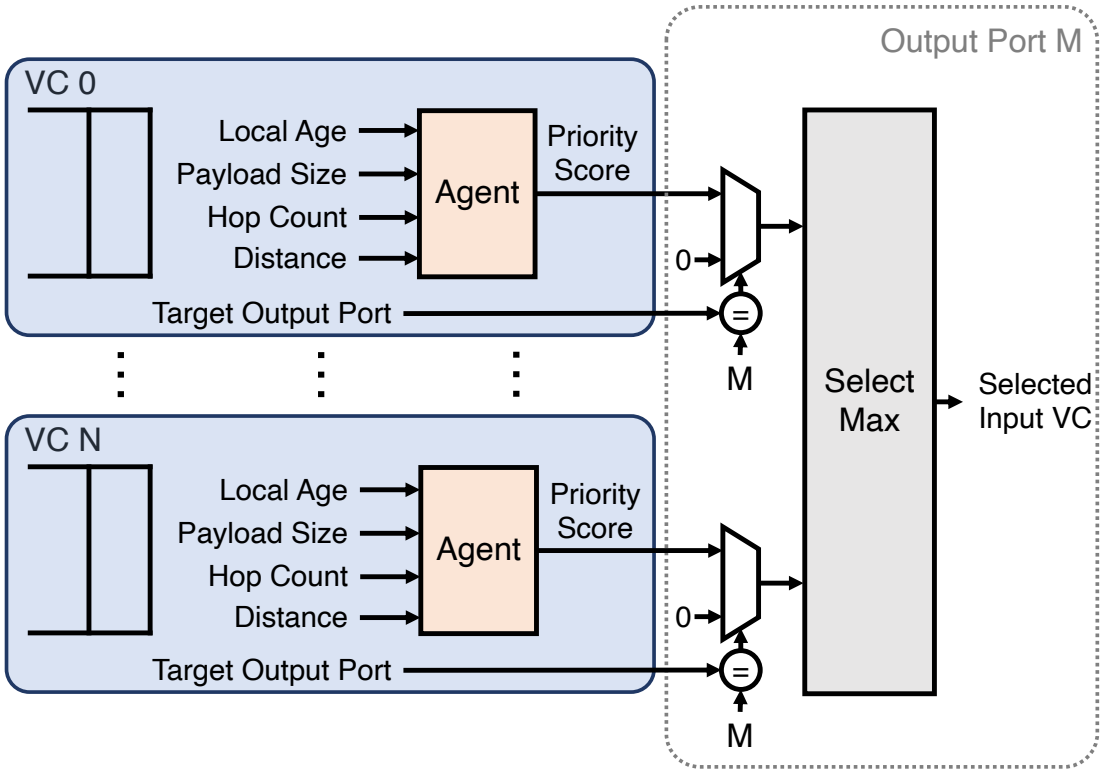


Figure 4.2: Architecture diagram for router arbitration — All routers in the network and all VCs in each router use identical copies of the same RL agent. At every cycle, the agent takes the packet information and computes a priority score for each VC. A set of masking and select-max logic chooses the VC with the highest priority at each output port.

4.2.1 Step 1: Learning an Arbitration Policy

Arbitration policies in computer systems can generally be learned using RL techniques. In this section we use NoC arbitration as a concrete example to introduce the key steps of feature construction and training. We believe similar approaches should apply to other problems in computer systems.

We use deep Q-learning (DQN) [108] to learn an arbitration policy for NoC routers. Figure 4.2 shows the architecture of routers in our simulation framework, where the same agent is shared by all VCs and all routers. Upon arbi-

tration, each candidate VC queries the agent and the agent returns a priority score. Similar to [171], the agent uses four features of the packet to make predictions: 1) local age, i.e., the amount of time the packet has spent at the local router where arbitration takes place; 2) payload size, i.e., size of the packet in bytes; 3) hop count, i.e., number of hops the packet has traversed so far; and 4) distance, i.e., number of hops between the current and destination nodes. These features are integers of fixed bit widths, and are normalized to the range of zero to one when training the neural network agent. At every output port, the priority scores of the VCs that are not requesting this port will be masked with zeros, and the output port is granted to the VC with the highest priority score [1]. The agent is given a reward of one if it correctly selects the globally oldest packet that is requesting a specific output port, otherwise a reward of zero is given. While our design of the reward function optimizes for network latency, designers can emphasize other quality-of-service (QoS) metrics by tuning the reward function. For example, assigning rewards based on router buffer occupancy emphasizes resource utilization. Through RL, the agent will learn different policies depending on the reward functions, which will result in different circuit implementations. The collected simulation trace contains tuples of ⟨*current state, action, next state, reward*⟩, and is added to a large replay memory. The weights of the agent are periodically updated by training on randomly sampled data from the replay memory. Please refer to Section 4.3 for more details on the hyperparameters and training dynamics.

[1]For multi-priority traffic, the arbiter selects the "best" packet within each priority level. With additional logic to enforce proper priority ranking, our methodology can accommodate this scenario without any major changes.

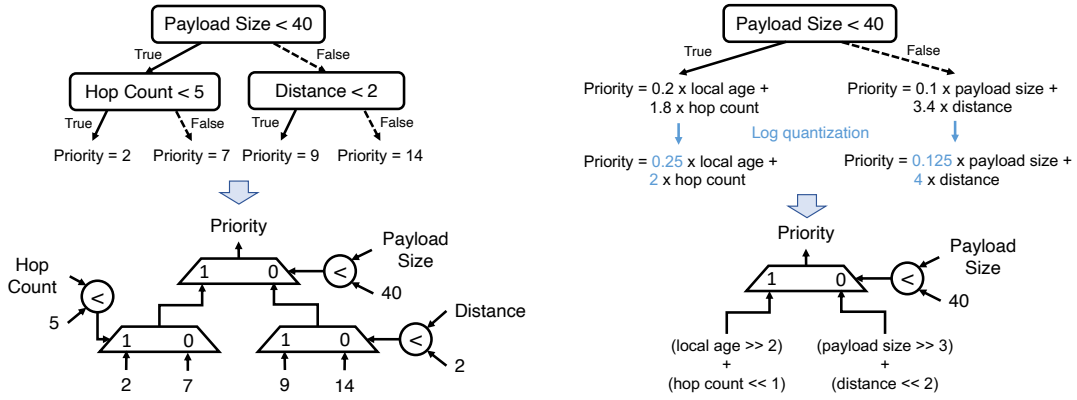
4.2.2 Step 2: Selecting the Tree Model

Among the various tree-based models, decision trees and linear model trees are of particular interest in our approach because they can be easily converted to logic. However, these two types of models intrinsically learn different types of functions: decision trees learn step functions, while linear model trees learn piecewise linear functions. Therefore, linear model trees are naturally more suitable if the target function is linear or piecewise linear, while decision trees are more suitable for very nonlinear target functions. Modern neural networks with ReLU activation functions approximate any arbitrary target function using piecewise linear functions, so linear model trees might be a better fit if we want to approximate the outputs of these neural networks.

Because linear model trees learn a linear function at each leaf node, they can represent more complicated functions than decision trees at equal depth. As a result, when implementing the same piecewise linear function, linear model trees will be shallower and can be implemented in hardware with potentially smaller area budget (more details in Section 4.3.1).

4.2.3 Step 3: Generating Implementable Logic

The logic generation process starts by training a tree model that approximates the output of the neural network agent. Decision trees and linear model trees must be trained in a supervised manner. As a result, a set of inputs must be collected, and the predicted scores from the neural network agent are used as labels to train the tree models. If the number of input features is small, the inputs can be collected by exhaustively including all possible input combinations.



(a) Convert decision tree to logic. (b) Convert linear model tree to logic.

Figure 4.3: Convert tree models to combinational logic — For linear model trees, log quantization is applied to convert multiplications to shift operations.

Otherwise, inputs can also be collected from simulation, where the collected data would cover the common cases but probably not every corner case. For NoC arbitration we choose the first approach, because the inputs are all within limited ranges, resulting in around 4,000 possible combinations.

To minimize the hardware overhead, the tree model should take unnormalized integer features, and output integer priority scores. In our experiments, we rescale the predicted priority scores from the neural network agent to the range of [0, 64), and use the rescaled scores as labels when training the tree models. The outputs of the tree models are quantized to six-bit integers using linear quantization.

Figure 4.3 shows how the trained tree models are converted to combinational arbitration logic. Each non-leaf node is implemented using a two-input multiplexer and a comparator. For decision trees, the leaf nodes are constant values. For linear model trees, the linear models at the leaves require multiplication and addition[2]. To further simplify the logic, we perform log quantization

[2]The predicted values of linear model trees might be negative. In this case, we can either use

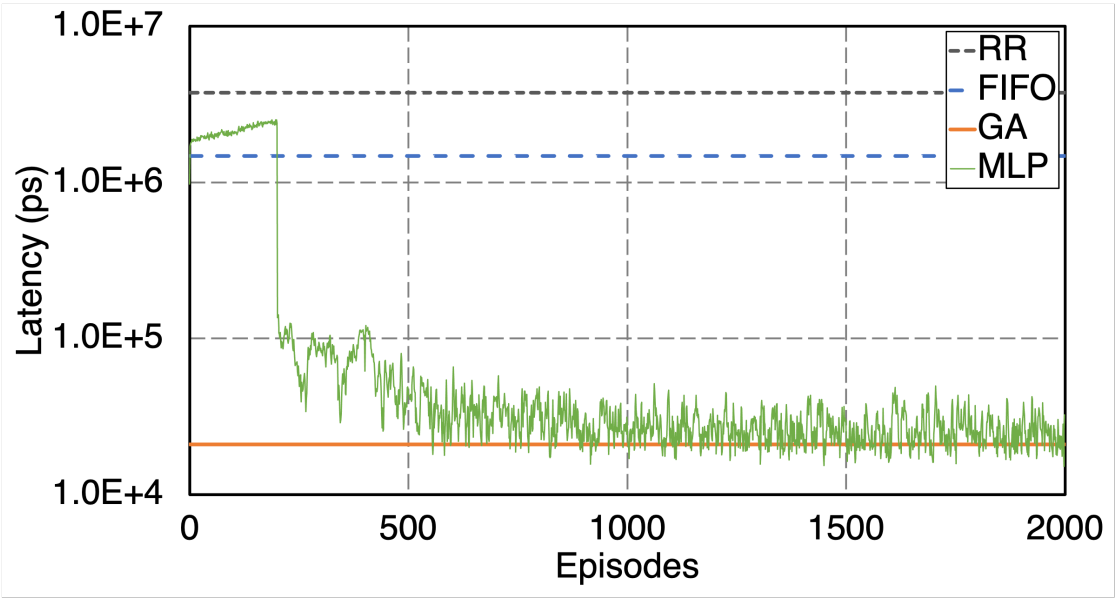


Figure 4.4: Training dynamics of the MLP agent and comparisons with different policies — `FIFO`: local-age-based, `RR`: round-robin, `GA`: global-age-based, `MLP`: MLP agent.

to the weights of the linear models and quantize them to powers of two. With this simplification, multiplications can be completely replaced by shift operations. Log quantization also gives the logic synthesis tool more opportunity to optimize the addition logic, because in case of right shifts the valid bit width of the operands will be reduced.

4.3 Case Study: NoC Arbitration Policy

In this section we present a detailed case study on NoC arbitration policy. We use the Garnet [8] network model in GEM5 [99] as our simulation platform[3]. The training of RL agents, tree models, and the conversion to logic

signed comparison at the select-max unit, or add a constant to all leaves such that the prediction is always non-negative.

[3]We modified the Garnet model in GEM5 to collect data and train the RL agent.

are implemented in Python leveraging PyTorch [119], sci-kit learn [120], and an open-source implementation of model trees [161]. To evaluate the area and latency, we synthesize the generated arbitration logic modules using Synopsys Design Compiler, targeting a TSMC 28nm technology library and 1GHz clock frequency. All experiments are performed on a server with a 64-core 2.80GHz Intel Xeon CPU and 384GB memory.

Without loss of generality, we constrain the routers in the network to have one virtual channel per virtual network to speed up training and facilitate our analysis. As mentioned in Section 4.2.1, the RL agent is given a scalar reward of one if it selects the globally oldest packet. This is equivalent to guiding the agent with a global-age-based oracle policy, which is unrealistic to implement in hardware. The RL agent is trained on a 4×4 mesh network with an injection rate of 0.32 packets/node/cycle under uniform random traffic. We choose this particular injection rate because it is the network saturation point (the point at which packet latency starts to increase dramatically) for the global-age-based policy. As shown in Figure 4.2, all VCs of all routers in the network share the same neural network agent, which is a multi-layer perceptron (MLP) with one hidden layer and sixteen neurons in this hidden layer.

During training, we launch GEM5 ten times, where at each launch we warm up the NoC for two million cycles and train for one million cycles. The agent is trained and updated every 5,000 cycles. In the rest of this section we will refer to 5,000 cycles as one episode. An exponential decay of the agent's exploration rate is used to encourage the agent to explore different actions at the beginning of the training, and exploit the learned policy towards the end[4]. Figure 4.4

[4]We use a replay memory with 80,000 entries, and every episode the agent is trained with 200 random batches of 32 entries sampled from the replay memory. The agent is trained using the Adam optimizer [83] with an initial learning rate of 0.001. Exploration rate $\epsilon = 0.9e^{-\tau/500}$,

Table 4.1: Performance and area comparison of different arbitration policies — Performance is measured under Uniform Random traffic with an injection rate of 0.32. Only the area of the per-VC arbitration logic is measured, i.e. the "agent" logic in Figure 4.2. `DT`: decision tree; `LMT`: linear model tree.

Model	**Avg. Packet Latency (ps)**	**Arb. Logic Area (μm^2)**
MLP	25659 ± 346	11446
DT (no regularization)	27015 ± 1174	725.8
DT (max depth = 12)	27241 ± 263	684.1
DT (max depth = 8)	25655 ± 597	247.7
DT (max depth = 4)	77833 ± 23931	28.2
LMT (max depth = 4)	23146 ± 222	214.0
LMT (max depth = 3)	24029 ± 171	112.9
LMT (max depth = 2)	24537 ± 494	73.7
LMT (max depth = 1)	23354 ± 373	45.9
LMT (max depth = 0)	24256 ± 518	19.7
FIFO	2113339 ± 50046	0.0
Manually Constructed [171]	2056407 ± 54344	8.6
Oracle (global-age)	21492 ± 272	N/A

1. Each agent logic is evaluated for five times. All circuits meet the 1ns timing constraint. The area of one instance of the agent logic is shown.
2. With no regularization, DT is equivalent to a hard-coded, quantized version of the MLP agent. LMT with a depth of zero is equivalent to a linear model.
3. The "agent" logic is just a set of wires for FIFO. The global-age-based policy is not realistic to be implemented in hardware.

shows the performance of the agent during training, as well as the comparison with FIFO (prioritizes packets based on their arrival time to the local router), round-robin (RR), and the oracle global-age-based (GA) policy. With this specific traffic, the agent steadily converges to the oracle policy and achieves two orders of magnitude lower average packet latency than RR and FIFO policies.

4.3.1 Area and Performance of the Distilled Arbitration Logic

Table 4.1 shows the performance and area comparison of different arbitration policies. We also include the manually constructed arbitration logic from [171][5]. While being very area-efficient, the manually constructed logic results in poor performance in our setup. Since the training traffic is at the saturation point of the global-age-based oracle policy, this manual policy would quickly degenerate to the FIFO policy with a large number of buffered packets. A straightforward way to implement an MLP as combinational logic is to use an array of multipliers and adders. The "MLP" row of Table 4.1 shows the area of this implementation, where the inputs to the multipliers and adders are quantized to eight bits. Compared with this implementation, decision trees can achieve up to 53× area reduction with slight performance degradation, while linear model trees achieve up to 282× area reduction with marginal performance improvement. When no regularization is applied to the decision tree, the generated logic can be considered as a hard-coded, quantized version of the neural network agent. Compared with this version, the circuits generated from regularized decision trees and linear model trees can achieve up to 15× area reduction without significant performance degradation. The circuit distilled from the linear model tree with a max depth of one achieves competitive performance with only $45.9\mu m^2$ area. As a reference, an eight-bit adder consumes around $17\mu m^2$ area under the same technology node and target frequency. While heavily regularized decision tree models can be converted to more area-efficient logic, the performance loss is non-negligible.

The linear model trees generally achieve better average packet latency than

where τ is the number of trained episodes.

[5]Priority = (local age << 1) + (hop count >> 1)

Algorithm 1 LMT Arbitration Policy

Input : local age, payload size, hop count, distance
Output : Priority
if *hop count* ≤ 5 **then**
    Priority = (local age >> 3) + (payload size >> 3) + (hop count << 1) + (distance >> 1) + 9
else
    Priority = (local age >> 2) + (payload size >> 1) + (hop count << 2) + distance - 20

Figure 4.5: Arbitration policy learned by linear model tree with a max depth of one, with all weights quantized to powers of two and biases adjusted.

their decision tree counterparts, which indicates that a piecewise linear priority function is more suitable for the NoC arbitration problem. Interestingly, the model trees can achieve lower latency than the original neural network agent. Our conjecture is that the neural network agent might have overfit during training, while the conversion to linear model trees applied a proper regularization effect.

4.3.2 Analysis of the Distilled Arbitration Logic

Figure 4.5 shows the arbitration logic learned by the linear model tree of depth one. The policy uses hop count as a critical feature so that packets are treated differently based on their travel distance. The learned policy generally favors larger packets that have been buffered locally for a longer time while also considering the topology information, which is consistent with human intuition. Compared with the manual policy from [171], our learned policy emphasizes less on local age and places higher weights on packet size and topology information. This prevents our policy from quickly degenerating to a FIFO policy under heavy traffic.

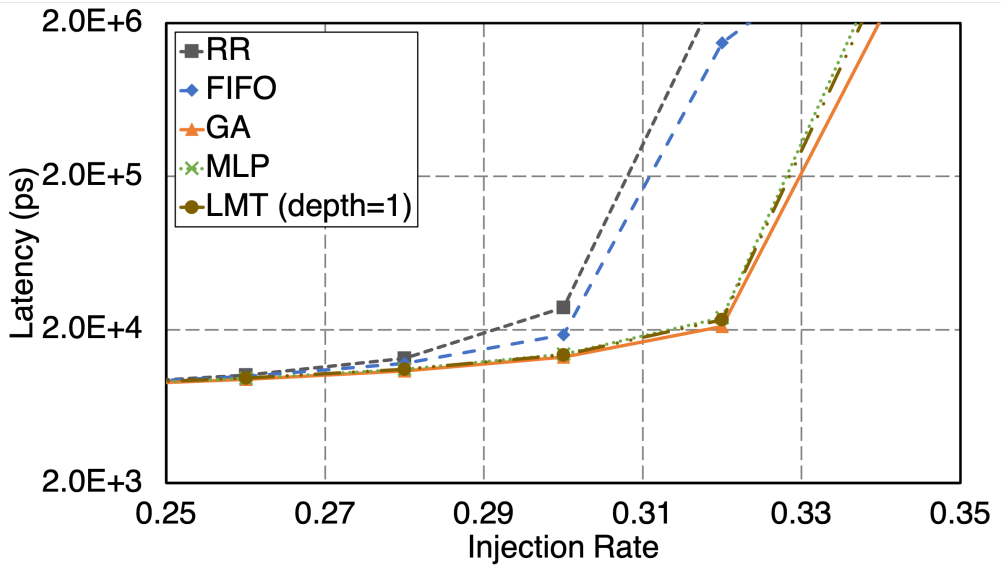
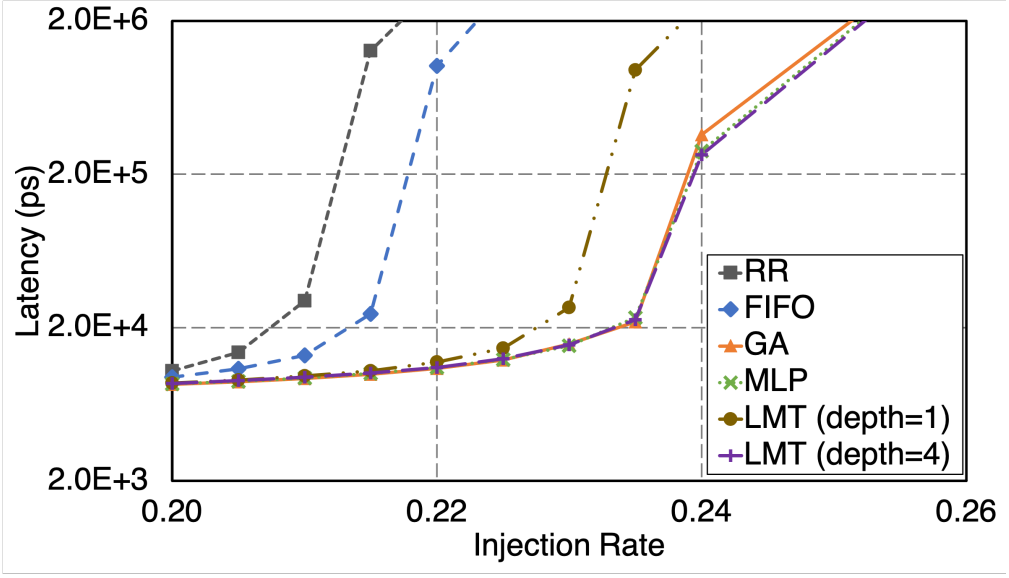


Figure 4.6: Performance of different policies under Uniform Random traffic — `LMT (depth=1)`: distilled arbitration logic from linear model tree with depth one.

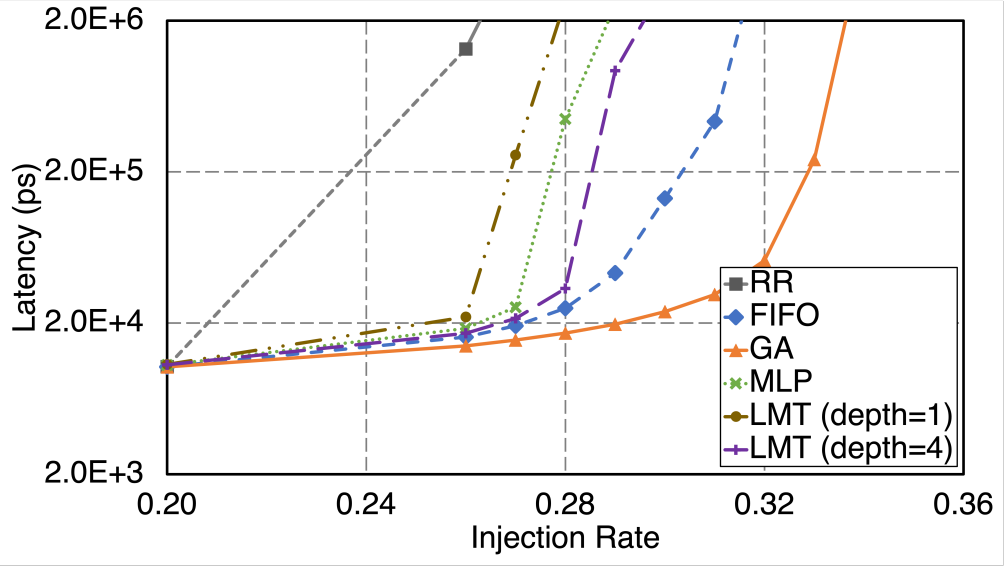
4.3.3 Generalization to Different Injection Rates

Figure 4.6 shows the performance comparison between the baselines, the oracle policy, the MLP agent, and distilled arbitration logic across different injection rates under uniform random traffic. The figure only shows the region around the network saturation point, as arbitration policy has little impact on NoC performance under low injection rates. While not shown in the figure, the manual policy from [171] performs marginally better than FIFO.

When the network is close to saturation, the MLP agent consistently outperforms FIFO and RR while being close to the oracle GA policy. The performance of the distilled arbitration logic is slightly better than the MLP agent, which is consistent with our findings in Section 4.3.1. Under the same traffic pattern, our agent and the distilled arbitration logic are able to generalize in situations of both light and heavy traffic and consistently outperform the baselines. Com-



(a) Transpose traffic.



(b) Bit-complement traffic.

Figure 4.7: Performance of different policies under traffic patterns that are unseen during training — MLP and LMT agents are trained under uniform random traffic.

pared to FIFO, the distilled policy improves the NoC throughput by 4.9%.

4.3.4 Generalization to Different Traffic Patterns

Figure 4.7 shows the performance comparison between different arbitration policies under the Transpose and Bit-complement traffic patterns[6]. Again, the manual policy from [171] (not shown in Figure 4.7) achieves similar performance with FIFO. The policies learned from uniform random traffic perform very differently on these two traffic patterns: the MLP and tree-based policies generalize quite well to the Transpose traffic, but perform worse than the FIFO policy on the Bit-complement traffic. Using shallower trees hurts performance under both traffic patterns. The reason is that the distilled arbitration logic is only an approximation of the MLP agent. While this inaccuracy results in a slight performance improvement under uniform random traffic, these two traffic patterns have longer average hop counts and require more sophisticated priority functions.

Under the Bit-complement traffic, again we observe the interesting behavior that the distilled LMT policy performs better than the original MLP agent. Apparently, the MLP agent overfit to uniform random traffic during training and cannot generalize well to the Bit-complement traffic. The conversion to linear model trees happens to apply a proper regularization effect to the agent. However, it is not guaranteed that such conversion will improve performance in all cases. It is important to note that the arbitration policies learned using our approach are still **traffic-specific** and cannot guarantee to generalize well to arbitrary traffic patterns.

[6]For the Transpose traffic pattern, node (x, y) only communicates with node (y, x), while for the Bit-complement pattern node (x, y) only communicates with node $(\bar{x}, \bar{y})$ where $\bar{x}$ and $\bar{y}$ are the bitwise inverse of x and y, respectively.

4.4 Related Work and Discussions

Related Work on Applying ML to Computer Systems Many critical problems in computer architecture can be effectively solved using ML. One of the most well-known applications of ML in computer architecture is probably the perceptron branch predictor, where a set of perceptrons are continuously updated during CPU execution to make accurate predictions [79]. Similar ideas are later applied to cache replacement [146] and prefetching [19]. Another line of research focuses on using RL to solve typical arbitration problems, including memory request scheduling [76], NoC routing policy [47], and cache prefetching [122]. These works implement their RL agents using Q-tables stored in memories, and the content of the memories are updated at run time to adapt to different workloads. The sizes of the Q-tables are subject to area and power constraints, which limits the complexity of the policies these approaches can learn.

With the development of DL, recent works explore the opportunity of applying DL techniques to computer architecture. Hashemi et al. performed a pure theoretical study of applying long-short term memory (LSTM) to cache prefetching [60]. Zeng et al. explored a similar idea, but embedded a small LSTM accelerator into the prefetcher to perform online training and inference [174]. Shi et al. proposed an LSTM-inspired cache replacement policy implemented as a support vector machine, where the hardware implementation and feature representation are designed after carefully examining the attention coefficients of the LSTM [133]. Zheng et al. proposed to use deep Q-Networks (DQN) for dynamic frequency-voltage scaling (DVFS) in a NoC [182]. The latency of the neural network accelerator is tolerable because the DVFS decisions

are made infrequently. Yin et al. presented a detailed case study on using DQN to learn a NoC arbitration policy [171]. By analyzing the weights of the trained neural network and incorporating domain knowledge, the authors were able to implement effective arbitration policies with small hardware overheads.

While our approach distills circuits from data and deep learning models, another relevant line of research focuses on efficient hardware implementation of deep neural networks. LUTNet [154] provides an efficient way of implementing binarized neural networks [42] on FPGAs by heavy pruning, fine-tuning, and directly mapping the XNOR gates in the network to look-up tables (LUTs). LogicNets go one step further by implementing the accumulation and activation functions also as LUTs [148]. These techniques are designed for mapping low-precision networks onto FPGAs, while our approach maps full-precision networks to ASICs.

Learning Combinational Logic from Labeled Training Data The discussions in this chapter have been focusing on learning an arbitration policy using RL and converting the DL model into combinational logic. If labeled data can be directly collected from simulation traces, a combinational logic implementation can be derived by directly training a tree-based model on the labeled dataset and converting the model to logic. In cases where the target function is complicated and cannot be directly learned by tree-based models, an alternative is to first train a DL model on the dataset and distill its knowledge to tree-based models as described in Section 4.2.3. In general, Circuit Distillation provides a realistic way of implementing the functionality of a DL model using combinational logic. While the conversion from the DL model to combinational logic is lossy, the trade-off between conversion accuracy and hardware cost can be controlled by

adjusting the tree depth and quantization mechanism.

As a special case of learning logic from labeled data, the problem of learning Boolean functions has been investigated for decades. Learning Boolean functions using decision trees can be dated back to the 1990s [116]. ESPRESSO [129] and Boolean optimization techniques based on binary decision diagrams [24] can also be viewed as learning Boolean functions from partial truth tables. Cartesian genetic programming (CGP) [106] searches for optimal logic implementations using an evolutionary algorithm. A more recent work by Chatterjee [26] approaches Boolean function learning as a ML classification problem and learns Boolean functions using DL models and LUT networks.

In summer 2020, a programming contest on learning Boolean functions was organized at the 2020 International Workshop in Logic Synthesis. One hundred benchmarks from arithmetic, random logic, and machine learning domains were provided to the contest participants, where only 6400 randomly-selected lines of the truth table were given for each benchmark. The generated logic for each benchmark must be represented as an And-Inverter graph (AIG) [20] and use no more than five thousand AND gates. I cooperated with several colleagues in our group and won third place in the contest [7]. Our technique features an ensemble of multiple ML models including decision trees, random forests, and MLPs. Tree-based models achieve competitive performance on benchmarks with a small number of input bits, while MLPs excel on benchmarks with more inputs, including the machine learning tasks derived from the MNIST [91] and CIFAR-10 [86] image classification datasets. Other participants leverage different ML models, CGP, and LUT networks to learn the provided benchmarks. More details can be found from the IWLS 2020 technical

[7] In collaboration with Jordan Dotzel, Yichi Zhang, and Hanyu Wang.

report [125].

Distilling Tree Models from Deep Neural Networks The ML community approaches the problem of training tree-based models from DL models from an interpretability perspective. Earlier attempts try to improve the robustness of the tree-based models by efficiently sampling the input space. The TREPAN algorithm queries the MLP oracle model at the split boundaries of the student decision tree model to generate more faithful splits [43]. The DecText algorithm exploits a similar idea and introduces specialized splitting and pruning techniques to maximize the fidelity of the decision tree to the DL model [22]. Krishnan et al. uses genetic algorithms and probabilistic methods to sample promising input samples for a black-box DL model when the original training data is unavailable [85]. More recently, Liu et al. propose to use the output logits of the DL model instead of the predicted labels to distill knowledge to a decision tree [98]. Similar approaches have seen applications in the health care domain [27, 28]. Frosst and Hinton propose to distill soft decision trees from DL models [52]. While this approach is systematic and yields high accuracy, soft decision trees are not directly applicable to latency-constrained problems in computer architecture because they make decisions using path probabilities instead of the labels at the leaf nodes. Currently, an active line of research tries to combine the strengths of DL models and tree-based models using a hybrid approach [144, 153]. These hybrid models usually have components from DL models, thus are still prohibitive to be computed using logic.

CHAPTER 5

TRACE-BASED ON-CHIP MEMORY BANKING FOR SOFTWARE-PROGRAMMABLE FPGAS

With the general-purpose CPU performance scaling significantly slowing down in the past decade, FPGAs have become an attractive option for fulfilling the role of application-specific hardware acceleration. An FPGA-based hardware accelerator is typically highly parallelized and/or deeply pipelined in order to achieve a desirable throughput. As a result, multiple parallel accesses to a single on-chip memory are often required to provide the necessary data bandwidth to sustain the high throughput of the accelerator. However, the embedded memory blocks available on modern FPGA devices (e.g., BRAMs) only provide a very limited number of ports for concurrent reads/writes[1]. Simply replicating the memory blocks would not be feasible due to the steep area overhead and potential memory coherence overhead resulting from write operations.

A more viable solution is memory banking, which partitions a memory block into several smaller banks; thus, concurrent memory accesses are distributed to different banks, avoiding or minimizing banking conflicts. Since each memory bank only holds a subset of the original memory contents, memory banking usually yields a significantly lower storage overhead compared to memory duplication. Nevertheless, additional banking logic is required to orchestrate the data movement between banked memories and compute units in the accelerator. For non-expert FPGA designers, devising a minimum-conflict banking scheme with low hardware overhead is certainly a challenging task. While

[1]Even for ASICs, it is not feasible to have a large number of memory ports due to the excessive area and power overhead [145].

commercial HLS tools provide some basic support for array partitioning [40], the users remain responsible for manually specifying the banking scheme via vendor-specific pragmas or directives. For this reason, there is an active body of HLS research tackling the problem of automatic array partitioning (i.e., memory banking) given a throughput constraint that is usually specified in terms of pipeline initiation interval (II) [37, 105, 139, 156, 157].

In this chapter, we present a trace-based banking algorithm, denoted as TraceBanking, which is very different from the existing methods[2]. Specifically, our approach mines a stream of memory address bits to determine a banking scheme that minimizes the number of access conflicts and simplifies the banking logic. Unlike mainstream techniques that are mostly based on static compiler analysis, TraceBanking only relies on simple source-level instrumentation to provide the memory trace of interest without enforcing any coding restrictions (such as static control parts often required by polyhedral analysis [17]). Our work has been published in FPGA'17 [184]. The major technical contributions of our work are threefold:

- We offer a fresh look at memory banking, by waiving the requirements of using static compile-time analysis. We show that from a trace of memory addresses, we can identify a set of "interesting" address bits that form the basis of the hardware-efficient memory banking function. In addition, our technique is able to form banking functions that do not belong to the solution space of the existing linear-transformation-based techniques.
- We propose a two-step heuristic to solve the trace-based memory banking problem. This heuristic is not only able to exploit regular memory access patterns, but can also generate efficient solutions for applications with ir-

[2]This work is in cooperation with Khalid Al-Hawaj from Prof. Christopher Batten's group.

regular memory accesses.

- We propose an SMT-based checker that can formally verify if a memory banking solution is free of access conflicts under all possible execution traces. This allows the usage of a reduced (or incomplete) memory trace to significantly speed up TraceBanking, but without the risk of accepting an inferior banking solution.

The rest of this chapter is organized as follows: Section 5.1 uses an motivational example to explain the memory banking problem and illustrate the intuition behind our work; Section 5.2 describes the TraceBanking algorithm in detail; Section 5.3 introduces the SMT-based banking solution checker; Section 5.4 reports the experimental results on commonly used benchmarks with affine memory accesses, as well as a case study on a face detection application with irregular memory accesses; Section 5.5 discusses related work on memory optimization for software-programmable FPGAs.

5.1 Motivational Example

The rest of this chapter assumes the hardware architecture shown in Figure 5.1, which contains K compute units and N memory banks connected by a crossbar. For simplicity, we assume that each compute unit only has one memory load port, and each memory bank only has one read port, but TraceBanking can be generalized to handle multi-port memories. Memory bank n is connected to a multiplexer with M_n inputs, where each input connects to one compute unit that needs to access this bank. A *conflict-free* memory banking solution partitions a monolithic memory array to be accessed into the N memory banks, such that

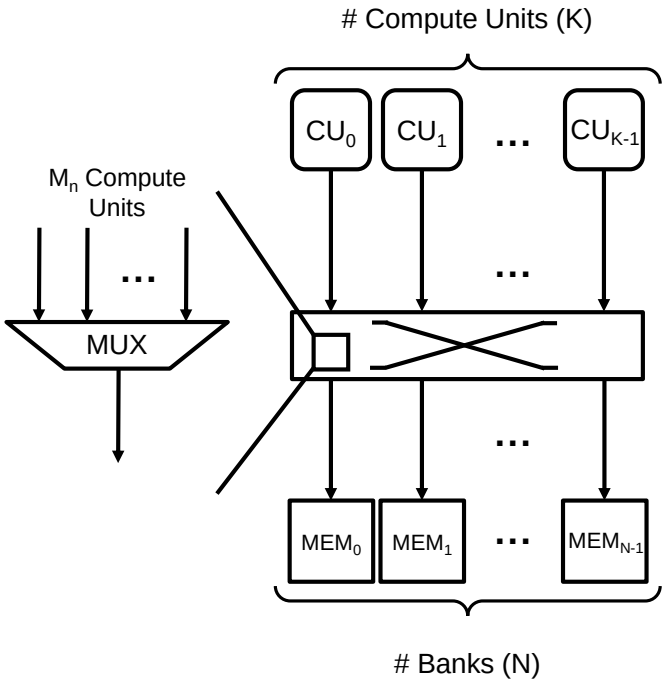


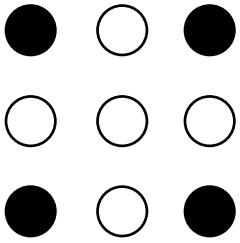
Figure 5.1: Hardware template for memory banking.

each memory bank is never accessed by more than one compute unit concurrently during execution. When multiple compute units access the same memory bank in the same clock cycle, we say there is a banking conflict. In certain cases, the designer might prefer a *conflict-less* banking solution which allows a small number of banking conflicts but uses fewer hardware resources. Our discussions in this chapter will focus on conflict-free memory banking. Chapter 6 will discuss how to extend TraceBanking to support conflict-less memory banking.

Figure 5.2a shows a simplified loop nest from bicubic interpolation. To achieve an II of one, the accelerator must perform four concurrent memory accesses to a two-dimensional array in every cycle with the memory access pattern illustrated in Figure 5.2b. In order to provide sufficient memory bandwidth under the hardware template specified by Figure 5.1, we need $K = N = 4$. Existing techniques, such as GMP [156], analyze the symbolic expression of memory ac-

```
int A[Rows][Cols];
for (int i=1; i<Rows-1; i++)
  for (int j=1; j<Cols-1; j++)
    #pragma HLS pipeline II=1
    foo(A[i-1][j-1], A[i-1][j+1],
        A[i+1][j-1], A[i+1][j+1]);
```

(a) Loop kernel.



(b) Memory access pattern.

Iteration	addr0	addr1	addr2	addr3
0	0$\underline{0}$0, 0$\underline{0}$0	0$\underline{0}$0, 0$\underline{1}$0	0$\underline{1}$0, 0$\underline{0}$0	0$\underline{1}$0, 0$\underline{1}$0
1	0$\underline{0}$0, 0$\underline{0}$1	0$\underline{0}$0, 0$\underline{1}$1	0$\underline{1}$0, 0$\underline{0}$1	0$\underline{1}$0, 0$\underline{1}$1
...	...	...	...	...
5	0$\underline{0}$0, 1$\underline{0}$1	0$\underline{0}$0, 1$\underline{1}$1	0$\underline{1}$0, 1$\underline{0}$1	0$\underline{1}$0, 1$\underline{1}$1
...	...	...	...	...
Cols-1	0$\underline{0}$1, 0$\underline{0}$0	0$\underline{0}$1, 0$\underline{1}$0	0$\underline{1}$1, 0$\underline{0}$0	0$\underline{1}$1, 0$\underline{1}$0
...	...	...	...	...

(c) Sample memory trace.

i / j	0	1	2	3	4	5	6	7
0	0	1	2	3	0	1	2	3
1	0	1	2	3	0	1	2	3
2	1	2	3	0	1	2	3	0
3	1	2	3	0	1	2	3	0
4	2	3	0	1	2	3	0	1
5	2	3	0	1	2	3	0	1
6	3	0	1	2	3	0	1	2
7	3	0	1	2	3	0	1	2

(d) GMP solution.

i / j	0	1	2	3	4	5	6	7
0	0	0	1	1	0	0	1	1
1	0	0	1	1	0	0	1	1
2	2	2	3	3	2	2	3	3
3	2	2	3	3	2	2	3	3
4	0	0	1	1	0	0	1	1
5	0	0	1	1	0	0	1	1
6	2	2	3	3	2	2	3	3
7	2	2	3	3	2	2	3	3

(e) TraceBanking solution.

Figure 5.2: Bicubic interpolation example — **(a)** Pipelined loop kernel with four memory accesses in each cycle. **(b)** Memory access pattern of the loop kernel in two-dimensional memory space. **(c)** Memory trace generated by concatenating array indexes: Addresses are formed by concatenating the two-dimensional array indexes i and j (for simplicity, i and j are both truncated to three bits). **(d)** GMP banking solution [156]. **(e)** An alternative solution generated by TraceBanking.

cesses and search for appropriate coefficients to construct a banking function in the form of $bank(i, j) = \lfloor(\alpha_0 i + \alpha_1 j)/B\rfloor \% N$. Figure 5.2d shows the resulting 4-bank partitioning scheme, where $\alpha_0 = 1$, $\alpha_1 = 2$, and $B = 2$.

Figure 5.2e shows an alternative banking scheme, which is not in the solution space of the GMP approach. By examining the memory trace in Figure 5.2c, we can identify two important address bits: the second-to-last bit of i, plus the second-to-last bit of j. We refer to such bits as **mask bits**. These two bits combined can differentiate the four memory accesses belonging to the same iteration. As a result, we can divide the original array into four memory banks according to the values of the two mask bits and arrive at the alternative scheme in Figure 5.2e.

This example demonstrates the possibility of performing memory banking based on a stream of memory addresses. Although the example has an affine memory access pattern, TraceBanking is also capable of generating memory partitioning for applications with irregular memory accesses. Regardless of the memory access pattern, it is important to identify the mask bits that form the basis of banking. The value of mask bits is referred to as **mask ID**. In the following sections, we will discuss how TraceBanking identifies mask bits and derives efficient banking solutions accordingly.

5.2 TraceBanking Algorithm

Given a memory address stream and a fixed number of memory banks, a straightforward method to find a banking solution with a minimum number of banking conflicts is to use an ILP solver. However, ILP solvers are not scalable in general. Therefore, there is a need for heuristics which can find an optimal mapping between addresses and banks with a reasonable execution time. In this section, we introduce the TraceBanking algorithm, which is a flow of heuristics

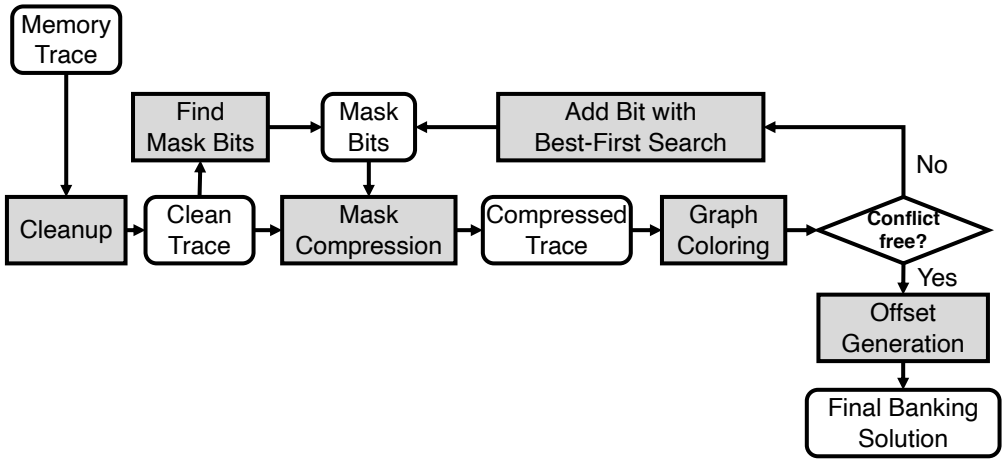


Figure 5.3: TraceBanking flow.

to solve the memory banking problem described in Section 5.1. Specifically, TraceBanking uses the number of available banks as a constraint and finds an optimized mapping by solving three sub-problems: (1) Finding a set of promising address bits to form mask bits, (2) Finding a mapping between the generated mask IDs and available banks, and (3) Computing the offset of each memory element after banking.

The complete flow of TraceBanking is shown in Figure 5.3. For clarity, here we limit our discussion to finding conflict-free banking solutions, but it is possible to apply a more complex objective function to find conflict-less solutions (e.g., area-optimized banking, power-aware banking) because TraceBanking can accept different objective functions in the search process. Assuming the array to be partitioned is accessed in a loop, TraceBanking takes the number of memory banks and a memory trace that specifies the memory accesses in each iteration as inputs. The memory trace is first cleaned up to combine redundant accesses and duplicate iterations[3]. The clean memory trace is used to find the

[3]We consider memory accesses to be redundant if multiple accesses in the same iteration request the same address. These accesses are combined into one. Two iterations in the memory trace are considered as duplicates if they access the same set of addresses. A weight property is

mask bits. After the initial set of mask bits is retrieved, the trace is further compressed, then our graph coloring engine constructs the memory conflict graph and tries to search for a feasible bank assignment. If a conflict-free solution is found, then TraceBanking proceeds to offset generation and produces the final banking solution. Otherwise, we iteratively add more bits to the mask using a best-first search so that the memory accesses in the same iteration can be better distinguished. This iteration terminates when the graph coloring engine finds a conflict-free solution, or after we include all address bits into the mask.

5.2.1 Finding Mask Bits

TraceBanking finds a set of mask bits that can distinguish all memory accesses within the same iteration using the `findMaskBits` algorithm shown in Figure 5.4. This algorithm takes the available number of banks, N, as well as the clean memory trace, T_c, as inputs. It evaluates any candidate mask using two objectives: mask IDs' conflicts and conflict graph colorability. The search starts with masks that include $\lceil log_2(N_A) \rceil$ bits, where N_A is the maximum number of memory accesses in all iterations in the clean memory trace. It tries all possible combinations of $\lceil log_2(N_A) \rceil$ bits; each combination constructs a unique mask which maps addresses to mask IDs. Going through the clean memory trace, the algorithm evaluates mask IDs conflicts by counting the number of times when two addresses in the same iteration have the same mask ID. After finding a mask that has the lowest number of mask IDs conflicts, the algorithm constructs a conflict graph — every node in the graph represents a mask ID; edges between nodes indicate mask IDs that appeared together in at least one iteration, and the

added to each iteration indicating its frequency in the memory trace.

Algorithm 2 findMaskBits

```
Input  : N – number of available banks;
         T_c – clean memory trace.
Output: mask – initial mask.
for nbits ← ⌈log2(N_A)⌉ to address_size do
    mask ← all possible mask combinations with size nbits
    while mask ≠ null do
        num_conflicts ← calculateConflicts(T_c, mask)
        if num_conflicts ⩽ min_conflicts then
            /* Possible Solution                                  */
            min_conflicts ← num_conflicts
            /* Test graph colorability                            */
            G ← constructGraph(T_c, mask)
            if χ(G) ⩽ min_χ then
                min_χ ← χ(G)
            end
            if min_conflicts = 0 and min_χ ⩽ N then
                /* Early stopping if a feasible mask is found */
                return mask
            end
        end
        mask ← next(mask)
    end
end
return mask
```

Figure 5.4: The heuristic in TraceBanking to find mask bits.

edges' weights represent the frequency. Thus, the banking problem is transformed to a graph coloring problem. The `findMaskBits` algorithm calculates a lower bound for the colorability of the conflict graph by computing the maximum clique size of the graph $\chi(G)$; because graphs with maximum clique size greater than the number of banks, N, cannot be colored with N colors.

Algorithm 3 mapMaskIDsToBanks

Input : N – number of available banks;
T_c – clean memory trace;
mask – initial mask.
Output: *addr_mapping* – a mapping from addresses to banks.
do
  `/* Perform mask-compression */`
  $T_{mc} \leftarrow$ maskCompression(T_c, *mask*)
  `/* Construct a graph */`
  $G \leftarrow$ constructGraph(T_{mc})
  `/* Create a seed using greedy coloring */`
  $S \leftarrow$ greedyGraphColoring(G, N)
  `/* Color G using evolutionary algorithm */`
  $num\_conflicts, addr\_mapping \leftarrow$ eaGraphColoring(S, N)
  `/* Ending conditions */`
  if *bits_remaining(mask)* = 0 **then**
    break
  else if $num\_conflicts \neq 0$ **then**
    $mask \leftarrow$ performBestFirstSearch(T_c, *mask*)
  end
while $num_{conflicts} \neq 0$;
return *addr_mapping*

Figure 5.5: The heuristic in TraceBanking to map mask IDs to banks.

5.2.2 Mapping Mask IDs to Banks

After the initial mask is found by `findMaskBits`, the second step is to find bank assignments for mask IDs such that the number of potential conflicts is minimized. This step is performed by the `mapMaskIDsToBanks` algorithm shown in Figure 5.5. To reduce complexity and redundant work, the clean trace is further compressed by applying mask-compression, which is similar to the cleanup step explained earlier except that the addresses are replaced with their corresponding mask IDs. After that, the algorithm will construct a conflict graph from the mask-compressed trace in the same way as described in Sec-

tion 5.2.1. With the conflict graph, the banking problem is directly converted to a maximum coloring problem. The number of banks represents the number of colors available for coloring. The `mapMaskIDsToBanks` algorithm first generates a colored seed S using multiple order-based greedy heuristics [10, 77]. If the seed is not conflict-free, the `mapMaskIDsToBanks` algorithm attempts to minimize the number of conflicts using an evolutionary algorithm [77]. In each evolutionary step, it performs a set of heuristics that show efficiency in coloring memory-accesses graphs. Once a coloring for a conflict graph is found, the evolutionary algorithm concludes and returns the banking function *addr_mapping* constructed from the coloring.

If the algorithm cannot find a conflict-free coloring in a bounded number of evolutionary steps, it is assumed that the graph is not colorable. Then, TraceBanking proceeds to perform a best-first search. The search will modify the mask by adding one extra bit to it. It is reasonable to assume that address bits that are part of the final mask have an additive effect in reducing conflicts when considered; as a result, the best-first search tests the colorability of remaining bits by adding them to the mask in isolation. Then, the search includes the bit that yields a graph with the minimum number of conflicts permanently to the mask. Since this is a rough assumption, TraceBanking might use more bits than theoretically needed to find a feasible banking solution.

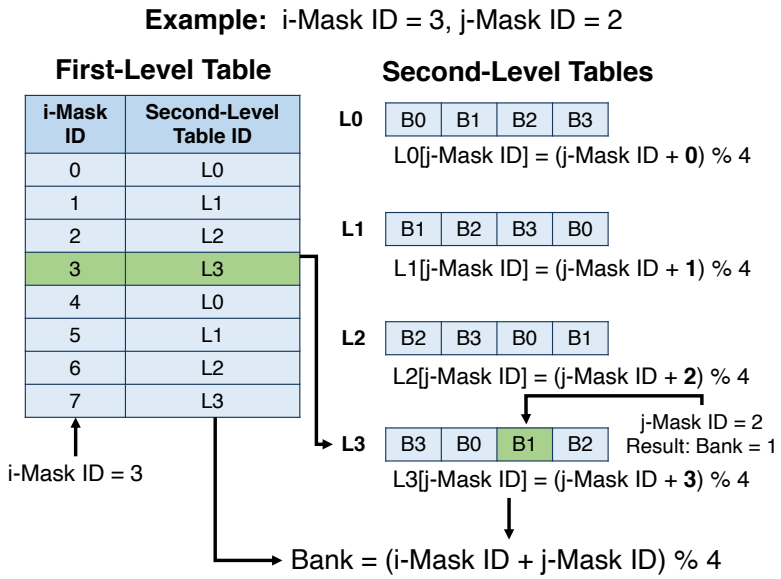
5.2.3 Offset Generation

The banking function generated by the `mapMaskIDsToBanks` algorithm only specifies the **bank assignment** of the memory elements. To properly access the

Address: i (6 bit) | j (6 bit)
Original Mask: 000111 | 000011
Partitioned Mask: i-Mask: 000111; j-mask: 000011

Mask ID	Bank ID	i-Mask ID	j-Mask ID	
0	B0	0	0	L0
1	B1	0	1	
2	B2	0	2	
3	B3	0	3	
4	B1	1	0	L1
5	B2	1	1	
6	B3	1	2	
7	B0	1	3	
…	…	…	…	…

(a) Banking solution in a look-up table.



(b) Multi-level look-up table and closed-form solution.

Figure 5.6: Example of mapping banking solution into closed-form equations — **(a)** Mask bits and the banking solution: An address bit noted as '1' is a mask bit, while an address bit noted as '0' is not. The mask bits are divided into two parts, i-Mask and j-Mask, according to the concatenation of array indices. **(b)** i-Mask is used to index the first-level table, and j-Mask is used to index the corresponding second-level table. Each second-level table can be represented with a closed-form equation. Constants in bold indicate the relationship between bank ID and i-Mask ID.

memory banks, we also need an offset function that maps each memory element to an intra-bank offset. An intuitive method to generate the offset function is

to simply scan every data element in the data domain and assign consecutive integers to data elements in each bank. Without any constraints on the offset function, this integer counting method is effective for both regular and irregular banking solutions. In addition, this method is optimal in terms of storage overhead since the data elements in each bank are guaranteed to have consecutive intra-bank offsets.

The generated banking and offset functions are represented in the form of look-up tables by default. For applications with regular memory access patterns, it is possible to convert the look-up tables generated by TraceBanking into equivalent closed-form equations, which essentially uncovers and exploits the regularity in the original application. The key idea is to decompose the look-up table into multiple stages of smaller look-up tables, and use a simple search to map the sub-tables into equations. The composition of memory addresses is retrieved from source-level instrumentation. An example is shown in Figure 5.6. The original 5-bit mask shown in Figure 5.6a is divided into two disjoint sub-masks: i-Mask and j-Mask — according to the corresponding array indices. By grouping the entries with the same i-Mask ID, the original banking solution shown in Figure 5.6a is decomposed into two levels, where the first level is used to determine the look-up table for the second level. Figure 5.6b shows how to index the decomposed look-up tables, where the i-Mask ID is used to index the first-level table and j-Mask ID is needed to index the second-level table and retrieve the actual bank ID. As illustrated in Figure 5.6b, each second-level look-up table can be represented by a modulus operation. By searching for coefficients to represent the relationship between i-Mask ID and the constants in the equations (highlighted in bold in Figure 5.6b), we can represent the original banking solution with one closed-form equation shown in Figure 5.6b. Clearly,

this approach can easily be generalized to arrays with higher dimensions. We also use a similar method to uncover the closed-form equation for an offset function, if such representation exists.

According to our experiments on a set of benchmarks with affine memory accesses, all of the results generated by TraceBanking can be represented by our equation template which is generalized from block-cyclic partitioning. Some of our solutions fall into the category of the cyclic partitioning scheme mentioned in the LTB approach [157]. Other solutions are not in the solution space of block-cyclic partitioning. Nonetheless, they can be efficiently represented with a few number of mask bits (e.g., bicubic solution in Figure 5.2e).

5.3 SMT-Based Verification

The discussion in Section 5.2 assumes that the input memory trace to TraceBanking is complete. In other words, the input trace captures all memory accesses from the entire software execution. In this case, our solution is supposed to be sound in terms of guaranteeing no banking conflicts. When the given memory trace is incomplete, it is necessary to have a formal mechanism to verify if the resulting solution remains conflict-free under all possible scenarios.

To this end, we propose an SMT-based checker to validate the soundness of the solution with the aid of a simple compiler analysis. The checker takes the memory banking solution from TraceBanking, and the address expressions of the loop kernel from compiler analysis. With this information, we can formulate the SMT problem as shown in Figure 5.7a. The integer variables for the SMT problem correspond to loop induction variables in the original application. We

Define loop induction variables as SMT variables:
int $\vec{i}$
Define banking function:
int B($\overrightarrow{idx}$)
*/*definition of the banking function*/*
Define expressions of array indices:
int[ ] $idx_0(\vec{i})$
*/*array indices in the 1st load*/*
int[ ] $idx_1(\vec{i})$
*/*array indices in the 2nd load*/*
...
*/*the total number of loads*/*
const int instr_cnt = K
Construct iteration domain $\mathbb{D}$:
assert ((i[0] > 0) and (i[1] > 0) and ...)
Constraint for having at least one conflict:
assert

$$\bigvee_{\forall \vec{i} \in \mathbb{D}} \bigvee_{\forall a,b \in [0,K-1], a \neq b} B(idx_a(\vec{i})) = B(idx_b(\vec{i}))$$

(a) General SMT formulation.

Define loop induction variables as SMT variables:
int i, j
Define banking function:
int B(i, j)
*/*select the mask bits from indices*/*
return (i & 0x2) || ((j & 0x2) >> 1)
Define expressions of array indices:
int[ ] $idx_0(i, j) = (i - 1, j - 1)$
int[ ] $idx_1(i, j) = (i - 1, j + 1)$
int[ ] $idx_2(i, j) = (i + 1, j - 1)$
int[ ] $idx_3(i, j) = (i + 1, j + 1)$
Construct iteration domain $\mathbb{D}$:
assert ((i > 1) and (j > 1) and
(i < Rows-1) and (j < Cols-1))
Constraint for having at least one conflict:
assert ((B(i-1, j-1) = B(i-1, j+1)) or
(B(i-1, j-1) = B(i+1, j-1)) or
(B(i-1, j-1) = B(i+1, j+1)) or
(B(i-1, j+1) = B(i+1, j-1)) or
(B(i-1, j+1) = B(i+1, j+1)) or
(B(i+1, j-1) = B(i+1, j+1)))

(b) Example of Bicubic interpolation.

Figure 5.7: SMT formulation of the banking solution checker — **(a)** General formulation. **(b)** Example of Bicubic interpolation.

represent the banking solution as a function of array indices, and expressions of array indices as functions of loop induction variables. Then, we specify the iteration domain as a constraint. Additionally, we add one constraint of having at least one banking conflict in the whole iteration domain. If the SMT problem is proven to be unsatisfiable, there is no memory access conflict in all iterations and the banking solution is valid.

The example shown in Figure 5.7b illustrates how the SMT-based checker validates the banking solution for bicubic interpolation shown in Figure 5.2e: the loop induction variables i and j are used as SMT variables, and the banking

function is represented symbolically. The constraints specify boundaries for the SMT variables, and compare every pair of addresses from the same iteration to check for conflicts.

5.4 Experimental Results

In our experiments, memory traces are generated by source-level instrumentation of the loop kernels. The addresses in the memory traces are constructed by concatenating multi-dimensional array indices. The core algorithm of TraceBanking processes the memory trace and generates banking and offset functions. This algorithm is implemented in C. We use Vivado Design Suite 2016.2 from Xilinx [168] for high-level synthesis (HLS), logic synthesis and simulation. The target FPGA device is Xilinx Virtex-7. The memory banking flow takes in the memory trace and generates solutions in the form of look-up tables or close-form equations. We use Z3, an SMT theorem prover, to verify the generated solutions [45]. Each verified banking solution as well as the corresponding application are implemented as synthesizable HLS code.

5.4.1 Results on Stencil Benchmarks

We adopt six stencil loop kernels from the GMP work [156]. In addition, we add the Stencil3D benchmark from MachSuite [127], which accesses a three-dimensional array, to stress test the robustness and scalability of our approach. We substitute the processing phase of these loop kernels with a simple summation to better compare the overhead of different memory banking solutions.

Table 5.1: Timing and resource usage comparison with the GMP [156] baseline, where the minimum number of memory banks is used — target clock period = 5ns; `BRAM` = # of BRAMs; `Slice` = # of slices; `LUT` = # of lookup-tables; `FF` = # of flip-flops; `DSP` = # of DSPs; `CP` = achieved clock period.

Benchmark	# Accesses	Method	# Banks	Mask Width	BRAM	Slice	LUT	FF	DSP	CP(ns)
BICUBIC	4	Baseline	4	-	4	74	217	163	0	3.89
		Ours	4	2	4	74 (+0.0%)	212 (-2.3%)	184 (+13%)	0 (+0.0%)	3.66
DECONV	5	Baseline	5	-	5	185	531	383	10	3.52
		Ours	5	12	5	182 (-1.6%)	541 (+1.9%)	383 (+0.0%)	10 (+0.0%)	3.37
DENOISE-UR	8	Baseline	8	-	8	180	616	391	0	4.15
		Ours	8	4	8	188 (+4.4%)	623 (+1.1%)	427 (+9.2%)	0 (+0.0%)	3.62
MOTION_C	4	Baseline	4	-	4	76	186	153	0	3.58
		Ours	4	2	4	68 (-11%)	193 (+3.8%)	190 (+24%)	0 (+0.0%)	3.65
MOTION_LV	6	Baseline	6	-	6	146	425	392	6	3.31
		Ours	6	6	6	146 (+0.0%)	425 (+0.0%)	392 (+0.0%)	6 (+0.0%)	3.31
SOBEL	9	Baseline	9	-	9	405	1296	692	27	3.93
		Ours	9	12	9	350 (-14%)	1059 (-18%)	719 (+3.9%)	27 (+0.0%)	3.96
STENCIL3D	7	Baseline	7	-	14	322	966	700	7	3.82
		Ours	7	15	14	308 (-4.3%)	932 (-3.5%)	624 (-11%)	7 (+0.0%)	3.74
Average						-3.8%	-2.4%	+5.6%	+0.0%	

We also implemented the GMP method [156] as the baseline. All the designs are pipelined with II of one for maximum throughput. The input image size of the designs is 64×48 ($5 \times 64 \times 48$ for Stencil3D), and the data is 8-bit wide. We employ efficient algorithms from [159] to implement our own area-efficient modulus functions. These customized modulus functions are used in both the baseline and our approach.

Area Comparison Table 5.1 shows a comparison with the baseline when the minimum number of banks is used. Both GMP and TraceBanking can generate valid banking solutions with the minimum number of memory banks. TraceBanking is able to reduce the number of slices by 3.8% on average. One of the reasons is that our banking function does not always use all the bits in the address or array indices, which in turn reduces the complexity of the banking

Table 5.2: Timing and resource usage comparison with the GMP [156] baseline, where the number of memory banks is restricted to be a power-of-two — target clock period = 5ns; `BRAM` = # of BRAMs; `Slice` = # of slices; `LUT` = # of lookup-tables; `FF` = # of flip-flops; `DSP` = # of DSPs; `CP` = achieved clock period.

Benchmark	# Accesses	Method	# Banks	Mask Width	BRAM	Slice	LUT	FF	DSP	CP(ns)
DECONV	5	Baseline	8	-	8	129	418	278	0	3.63
		Ours	8	4	8	125 (-3.1%)	411 (-1.7%)	302 (+8.6%)	0 (+0.0%)	3.11
MOTION_LV	6	Baseline	8	-	8	117	369	237	0	3.56
		Ours	8	3	8	119 (+1.7%)	391 (+6.0%)	282 (+19%)	0 (+0.0%)	3.77
SOBEL	9	Baseline	16	-	16	328	1114	525	0	4.34
		Ours	16	4	16	340 (+3.7%)	1129 (+1.3%)	472 (-10%)	0 (+0.0%)	3.89
STENCIL3D	7	Baseline	8	-	8	195	649	443	0	3.87
		Ours	8	6	8	201 (+3.1%)	655 (+0.9%)	450 (+1.6%)	0 (+0.0%)	3.70
Average						+1.4%	+1.6%	+4.8%	+0.0%	

logic. For example, in Motion_C, we are able to save 11% of slices with a 2-bit mask. Another reason is that our approach is able to discover additional banking solutions that are not in the search space of the GMP method. For example, in Sobel, our design uses all the 12 index bits but still saves 14% of slices compared to the baseline. While the GMP solution has to perform `mod 9` operations due to its block-cyclic nature, our solution alternates among three consecutive bank IDs in each row of the image, thus only requiring `mod 3` operations which are more area-efficient.

As pointed out by [156], an important design trade-off between logic complexity and storage overhead in memory partitioning is to enforce the number of memory banks to be a power-of-two instead of the minimum. Therefore, we conduct this experiment for the four benchmarks whose number of banks is not a power-of-two and compare our results with the baseline. Detailed results are shown in Table 5.2. Compared with the corresponding entries in Table 5.1, the designs in Table 5.2 generally have less area even though they use more memory banks and a more complex crossbar, because banking functions

Table 5.3: Execution time of TraceBanking on Motion_LV with different input array sizes.

Array Size	12×12	32×24	64×48	128×96	320×240	640×480
Runtime (s)	0.0096	2.19	4.88	6.94	12.87	33.38

are significantly simplified when the number of banks is a power-of-two. For GMP designs, multiplication and division become simple shifting operations, while modulus operations are just selecting LSBs. For our designs, the resource saving comes from the reduction in mask width. Compared with baseline, our designs use a negligible 1.4% more slices. In general, the hardware generated by our trace-based memory banking approach is comparable with GMP in terms of area and timing.

Scalability TraceBanking is able to generate competitive memory banking solutions from memory traces. However, using a complete memory trace may be expensive when the memory trace is large. Table 5.3 shows how the execution time of TraceBanking scales with an increasing array size. For applications with affine memory accesses, we can apply trace reduction to reduce the runtime. The general idea is to use a partial memory trace which covers an adequate number of steps. Because of memory access pattern redundancy in the trace, the generated banking scheme is likely to comply with banking schemes generated from a full trace. Since the banking scheme generated from a partial trace is not guaranteed to be valid, we use the SMT-based checker proposed in Section 5.3 to validate it. If the validation fails, we revert to using the complete memory trace.

We perform experiments with reduced memory traces for all the benchmarks listed in Table 5.1. For the size of the reduced trace, we use an empirical value

Table 5.4: Execution time of TraceBanking with reduced memory trace — Initial mask refers to the mask found by the `findMaskBits` algorithm described in Section 5.2.1, while the final mask refers to the mask found by the `mapMaskIDsToBanks` algorithm described in Section 5.2.2.

Benchmark	Reduced Array Size	Initial Mask Width	Final Mask Width	Runtime (s)
BICUBIC	8×8	2	2	0.0093
DENOISE	10×10	4	8	3.45
DENOISE2	16×16	4	4	0.017
MOTION_C	8×8	2	2	0.0094
MOTION_LV	12×12	4	4	0.0096
SOBEL	18×18	6	10	5.94
STENCIL3D	$5 \times 14 \times 14$	6	11	4.37

of $2 \times \#Banks$ in each dimension of the array. For example, if the loop kernel conducts Sobel edge detection on a VGA image (640×480), rather than iterating through the whole image, we execute the loop kernel on an 18×18 sub-image and use this reduced trace as the input to TraceBanking. For all the benchmarks listed in Table 5.1, TraceBanking is able to generate solutions which are proven to be valid using the reduced traces as inputs. Moreover, these solutions are identical to the ones generated from complete traces. The execution time of the SMT-based checker is less than a second. As shown in Table 5.4, the execution time of TraceBanking is reduced significantly by using partial traces without sacrificing the quality of the solutions.

A critical observation from Table 5.4 is that, in most benchmarks, the final solution is either in the beginning or at the very end of the search space. TraceBanking exploits the aforementioned observation in pruning the search space by performing two simultaneous searches: forward search and backward search. Forward search starts from the mask with minimum number of bits upward to the mask with maximum number of bits, stopping with the first mask that

```
pixel window[25][25];
pixel coord[12];
int filter_no;

CLASSIFIER:
for (filter_no=0; filter_no<2913; filter_no++){
  #pragma HLS pipeline II=1
  // read array indexes from look-up tables
  int x0 = rectangles_array0[filter_no];
  int y0 = rectangles_array1[filter_no];
  int w0 = rectangles_array2[filter_no];
  ...
  // access 8 data elements from array
  coord[0] = window[y0][x0];
  coord[1] = window[y0][x0+w0];
  ...
  // if condition met, access 4 more elements
  if ( (w2!=0) && (h2!=0) ) {
    coord[8] = window[y2][x2];
    ...
  }
  else {
    coord[8] = 0;
    ...
  }
  // process data
  foo(coord);
}
```

Figure 5.8: Classifier loop kernel in a face detection accelerator [138].

yields no conflicts. On the other hand, backward search starts from the mask with the maximum number of bits downward to the mask with the minimum number of bits, stopping when no bit can be removed without causing conflicts.

5.4.2 Case Study: Haar Face Detection

In this section, we use Haar face detection [152] as a case study to show the efficacy of TraceBanking on applications with non-affine memory accesses. The Haar algorithm uses cascaded classifiers to detect human faces rapidly and robustly. Thousands of weak classifiers are used in one run of the Haar algorithm, and each of them has a distinct memory access pattern. Figure 5.8 shows a code snippet of applying the weak classifiers in an HLS face detection accelera-

Table 5.5: Timing and resource usage comparison of two face detection designs — target clock period = 5ns; `BRAM` = # of BRAMs; `Slice` = # of slices; `LUT` = # of lookup-tables; `FF` = # of flip-flops; `DSP` = # of DSPs; `CP` = achieved clock period; `Latency` = latency of the loop kernel.

Implementation	BRAM	Slice	LUT	FF	DSP	CP(ns)	Latency
TraceBanking	34	4915	8266	12559	6	4.52	2923
Full Mux	22	21275	53553	23785	3	9.22	2919

tor [138]. The array `window` is a 25×25 image buffer and is steadily shifted in from the input image. Therefore, it is implemented with discrete registers. In each iteration, the loop kernel reads pixels into the array `coord` and processes them in the function `foo()`. There are 2913 classifiers in total. The constant arrays `rectangles_array[]` store the constants needed to compute the array indices in each iteration. There is an `if` statement inside the loop kernel. When the condition is met, the loop kernel accesses twelve pixels from the `window` array in that iteration; otherwise, eight pixels are accessed.

In order to maximize throughput, we need to fully pipeline the `CLASSIFIER` loop in Figure 5.8, where each classifier requires eight or twelve parallel accesses to the image buffer. Existing techniques cannot generate an efficient banking solution for this problem due to two reasons: (1) The 2913 classifiers have more than 2000 different memory access patterns in total, and (2) The array indices are non-affine without any linear relationship with the iteration variable `filter_no`. With TraceBanking, we are able to generate a conflict-free banking solution to partition the image buffer `window[25][25]` into 28 memory banks using the whole address as mask bits. The execution time is less than a second. Because the `window` array is a shifting window implemented using discrete registers, in this scenario, the memory banks are actually register banks.

Our baseline is a straightforward design that uses twelve instances of 625-to-1 multiplexer. We compare our design with this baseline and the result is shown in Table 5.5. The TraceBanking design in Table 5.5 refers to the memory banking design generated by our approach, and the Full Mux design refers to the baseline. For these two designs, we only extract the loop kernel part shown in Table 5.8 to better compare the banking hardware overhead. Our TraceBanking design reduces Slice, LUT and Flip-Flop usage by 76.9%, 84.6% and 47.2%, respectively. Meanwhile, the clock period is improved by 51.0%. BRAM usage increases because of the overhead in storing look-up tables for banking and offset functions. The reduction in logic resource usage results from the simplified muxing network in the TraceBanking design. In the TraceBanking design, two levels of multiplexers are used to connect the registers with the compute units, and each multiplexer has less than 30 inputs. In contrast, the Full Mux design uses twelve instances of 625-to-1 multiplexers, which consumes a lot more area. Even worse, the Full Mux design is extremely hard to route and unable to meet the 5ns timing target. Therefore, even though the Full Mux design has similar latency with the TraceBanking design, the total execution time of the loop kernel is much worse. Clearly, the banking scheme generated by TraceBanking helps improve both area and performance of the design, which contains very irregular memory accesses.

5.5 Related Work

There is a recent line of research that investigates the problem of automatic array partitioning in the context of HLS [180]. Initial efforts focus on one-dimensional arrays and attempt to find a proper cyclic partitioning with optimal schedul-

ing to ensure conflict-free parallel data accesses [39, 94]. More recent proposals such as [156, 157] generalize these results to handle nested loops and multi-dimensional arrays.

Notably, linear transformation is extensively used among the existing array partitioning techniques. For example, the LTB approach [157] searches for a coefficient vector $\vec{\alpha}$ to construct a cyclic banking function $bank(\vec{x}) = (\vec{\alpha} \cdot \vec{x})\%N$, given the number of banks N and the affine memory access pattern. Meng et al. proposed a fast algorithm to generate the LTB coefficient vector $\vec{\alpha}$ according to the topology of the memory access pattern in a multi-dimensional memory space [105]. The GMP approach generalizes the LTB algorithm and is able to generate block-cyclic banking functions in the form of $bank(\vec{x}) = \lfloor(\vec{\alpha}\cdot\vec{x})/B\rfloor\%N$ [156]. Cilardo et al. proposed a lattice-based banking algorithm using polyhedron analysis [37]. After our work was published, more recent works on memory banking are no longer limited to linear transformations. For example, Escobedo and Lin proposed to derive optimal banking schemes for stencil applications using graph coloring [49].

The aforementioned techniques all employ static compile-time analysis and are only effective with affine data access patterns. To the best of our knowledge, we are the first to introduce a comprehensive trace-based banking algorithm that is not limited to affine memory accesses. Along the lines of trace-based memory optimization, one relevant proposal is [18], which attempts to partition an array of data structures into distinct arrays by leveraging hints from software memory traces. However, this technique does not directly tackle memory banking for multi-dimensional arrays. After our work, Escobedo and Lin proposed a line of techniques to tackle memory partitioning for applications with non-

stencil memory access patterns [48, 50]. Zou and Lin proposed *Graph-Morphing*, which automatically generates memory partitioning solutions for non-stencil applications by analyzing dependence graphs [187]. Chen and Anderson proposed to leverage trace-based simulation for selecting the optimal memory partitioning scheme supported by LegUp [32].

Aside from memory partitioning, parallel data accesses can be further facilitated by creating data reuse buffers that exploit the locality in memory access patterns. For many image processing and signal processing applications, data reuse is a more hardware-efficient solution due to the regular memory access patterns in stencil-like operations. Along these lines, Su et al. introduced an efficient method of combining linear reuse analysis and cyclic memory partitioning to generate application-specific reuse-chains and memory-banking [139]. Li et al. proposed a more generic approach which is able to handle the combination of multiple affine memory access patterns [95]. TraceBanking does not directly tackle the memory reuse problem, but it can be used to generate effective memory banking solutions after the reuse pattern is determined.

CHAPTER 6

CONCLUSION

The increasing scale and complexity of modern computational platforms is causing unprecedented challenges in the productivity of hardware design. Manual design of effective heuristics and optimized architectural implementations is becoming more and more difficult, which results in an increasing number of design iterations. In addition, accurate early-stage design modeling becomes increasingly challenging, which hinders design space exploration by forcing designers to rely on the time-consuming evaluation steps at low levels of abstraction.

Recent advances in machine learning (ML) provides an opportunity to further improve the modeling accuracy and automate the optimization process for digital designs. While modern ML models can achieve superhuman performance on a wide range of tasks, they must be trained using a large amount of data. In the hardware design flow, simulation traces are an abundant source of information that can be easily obtained from various evaluation and verification steps. Specifically, we propose trace-based learning solutions for three challenging problems in hardware design: early-stage power estimation for IP cores, automated logic generation for partially reconfigurable modules, and memory banking for FPGA accelerators. In this dissertation, we have demonstrated the effectiveness of the trace-based learning methodology on these three problems, and we believe this methodology should have wider applications for many other problems in digital design.

6.1 Dissertation Summary and Contributions

This dissertation presents three trace-based learning techniques for agile hardware design and design automation. By exploiting the recent advances in ML, the techniques presented in this dissertation aim to assist hardware designers by (1) providing accurate quality-of-result (QoR) estimations early in the design flow to reduce design turn-around time, and (2) automating heuristic design and hardware optimization to reduce manual design effort. At a high level, this dissertation demonstrates that trace-based, design-specific learning can effectively improve a single design and shows that this methodology is applicable to a variety of different problems in the electronic design automation (EDA) flow. Chapter 1 starts this dissertation by outlining the productivity bottlenecks in hardware design and the challenges in applying ML to EDA. These challenges motivate design-specific learning using simulation traces, which is the common methodology shared by the three techniques proposed in this dissertation. Chapter 2 prepares readers with sufficient background knowledge by introducing the preliminaries of the digital design flow and the ML techniques used in this dissertation.

For early-stage QoR modeling, this dissertation focuses on fine-grained power estimation for fixed-function IP cores. Chapter 3 describes how we use state-of-the-art ML models to enable fast and accurate power estimation at register-transfer level (RTL) and cycle-level. In this work, we explore different ML models and feature encoding mechanisms, and achieve gate-level estimation accuracy for realistic designs at both RTL and cycle-level. The DAC'19 publication of PRIMAL-RTL is the first work to introduce deep learning to power estimation [186].

This dissertation further presents two approaches to automating the hardware design and optimization process. Chapter 4 introduces Circuit Distillation, a key technique for filling the missing link between deep learning (DL) models and compact logic implementations. Using a combination of reinforcement learning (RL) and ML techniques, Circuit Distillation is the first fully automated flow to generate combinational arbitration logic from traces. Circuit Distillation leverages tree-based models to effectively convert a pretrained DL model into combinational arbitration logic. Users can balance the trade-off between area and performance by regularizing and quantizing the tree-based models. The Circuit Distillation technique shows strong results on our network-on-chip (NoC) arbitration case study, and we believe similar methodologies can be effective for many other problems in computer architecture design. The TraceBanking technique introduced in Chapter 5 takes another direction by automatically optimizing the on-chip memory subsystem for FPGA accelerators. TraceBanking uses a self-designed, graph-based algorithm to learn from functional-level simulation traces and search for conflict-free memory banking solutions. Aside from being able to handle any arbitrary memory access patterns that are known at compile time, TraceBanking is also the first trace-based memory banking technique for high-level synthesis.

6.2 Future Directions

Trace-Based Learning and Design-Agnostic Learning Despite the efforts presented in this dissertation, the productivity of hardware design is still facing significant challenges, and a lot of problems in the digital design flow can be potentially solved using ML. Trace-based learning can be of particular interest for

problems that require accurate models to make fine-grained predictions, such as performance debugging, verification, and micro-architectural optimization. Such problems can benefit more from the rich details exposed by simulation traces.

Design-specific learning, on the other hand, tries to find a sweet spot by trading off the generality of the learned ML model with the effectiveness of learning and the effort of training data collection. Ideally, developers would prefer design-agnostic ML models that can make accurate predictions for any arbitrary design. To achieve this goal, the ML models need to understand the fundamentals of digital design, obtain enough knowledge about the target technology library, and predict the sophisticated optimizations that might be performed by the EDA tools. To accomplish such a challenging learning task, the first step is to make available a comprehensive, open-source collection of hardware designs for researchers to train ML models, establish common baselines, and evaluate new learning techniques. To this end, we have seen exciting developments in open-source hardware and EDA flows in the past several years [7, 16, 25, 128]. In addition, there is an active body of research on applying and adapting emerging ML models to the EDA problems. For example, graph neural networks (GNNs) [59, 84] are attractive for many tasks in the EDA toolchain because digital circuits can be represented as a graph of gates or modules from different abstraction levels. Recent works have demonstrated the effectiveness of GNNs on FPGA timing estimation [150], test-point insertion and timing model selection [101], instruction vulnerability detection [78], and switching activity estimation [179].

Transferable Power Estimation While the PRIMAL techniques introduced in Chapter 3 can achieve gate-level estimation accuracy from higher levels of abstraction, the ML models need to be retrained once the gate-level netlist of the IP core is modified. This drawback limits the application of PRIMAL in design space exploration. However, as discussed above, directly training a design-agnostic model for fine-grained power estimation is difficult due to the lack of training data and the difficulty of this learning task itself. Existing works on using design-agnostic learning to accelerate power estimation target average power estimation instead of time-based power estimation [96, 179]. One alternative is to extend PRIMAL to domain-specific learning instead of design-agnostic learning. During design space exploration, the ML models can be gradually trained to learn the power characteristics of a family of similar designs. A graph-based feature representation can be used to represent the connections between operators and modules, and GNN models might be able to provide accurate predictions after observing several design points. GRANNITE [179] explores using GNNs for average power estimation. While GRANNITE only estimates average switching factors of circuit nodes and still relies on the power analysis tool to provide the final power estimation, it demonstrates that GNN models can generalize well to unseen designs on the power estimation task.

Learning Reconfigurable and Sequential Logic One limitation of our Circuit Distillation approach is that the learned logic is not guaranteed to generalize to unseen workloads. Since our approach generates combinational logic, one straightforward solution is to incorporate more realistic workloads into the training process. Instead of generating hard-wired combinational logic, an alternative solution is to generate hardware modules that are reconfigurable. Ex-

amples include look-up table networks and multiplexer networks, which can be configured using programmable registers to implement different functions. However, compared to combinational logic, adding reconfigurability introduces significant overhead in area and latency. An efficient implementation is necessary to achieve a good trade-off between flexibility, area, and latency.

For many problems in computer architecture, such as branch prediction, cache prefetching, and cache replacement, the hardware must make decisions based on history information. As a result, the hardware module to fulfill the functionality must be stateful rather than combinational. Similarly, when using ML to solve these problems, stateful models such as recurrent neural networks (RNNs) are usually more suitable. Circuit Distillation is only capable of generating combinational logic from stateless ML models, thus cannot be directly applied to these problems. Extending Circuit Distillation to convert stateful ML models to sequential logic implementations would be a promising direction for future work.

TraceBanking for More Generic Hardware Templates TraceBanking can be extended to support different hardware templates other than the one discussed in Chapter 5. Modern FPGA devices usually contain dual-port BRAMs. Assuming the target device has M-ported memories, the `findMaskBits` algorithm and the graph coloring engine can be modified such that cases where no more than M concurrent accesses are made to the same memory bank are not considered as conflicts. Under certain circumstances, designers may prefer a non-unit II to balance throughput and resource utilization. TraceBanking can cooperate with the HLS scheduling algorithm in this situation, where the scheduling algorithm decides which memory accesses are performed concurrently and Trace-

Banking generates a banking solution.

Multi-Objective Search for Design Optimization Both Circuit Distillation and TraceBanking can be considered as search-based design optimization techniques. While in this dissertation these techniques only consider one objective during the search process, they can be extended to support multi-objective search and reflect the designers' emphasis on other design objectives. The reward function in Circuit Distillation can be tuned to optimize different quality-of-service (QoS) metrics, or even incorporate the area of the arbitration logic to co-optimize network performance and area. The core of TraceBanking is a multi-objective search engine. TraceBanking can be easily extended to support conflict-less banking by specifying a different objective function. For example, suppose a high-level area model of the banking hardware is available, TraceBanking can use a weighted sum of the number of banking conflicts and the area estimate to achieve a good trade-off between latency and area. Other metrics such as frequency and power can also be included.

From another angle, Circuit Distillation and TraceBanking can also be viewed as performing design space exploration (DSE) using learning-based methods. With the growing complexity of modern hardware, rapid DSE has become an active line of research, especially for high-level synthesis (HLS) where a large number of design points can be explored without major changes in the source code. Earlier works inspect the control-data flow graph generated by the HLS compiler front-end, use carefully-designed analytical models to estimate the performance and resource utilization of each design point, and automatically explore different design points using exhaustive search or smart grid search [132, 155, 181, 183]. A more recent work [162] exploits state-of-the-art

GNN models for performance estimation, and leverages RL techniques to explore the design space. The modeling methodologies and search algorithms used in these works can potentially be integrated into Circuit Distillation and TraceBanking, thus enabling more intelligent multi-objective search.

BIBLIOGRAPHY

[1] IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual. *IEEE Std 1666.1-2016*, 2016.

[2] Keras Implementation of ShuffleNet V2. *https://github.com/opconty/keras-shufflenetV2*, 2018.

[3] Keras: The Python Deep Learning Library. *https://keras.io/*, 2018.

[4] PyTorch. *https://pytorch.org*, 2021.

[5] scikit-learn: machine learning in Python. *https://scikit-learn.org/stable/*, 2021.

[6] TensorFlow. *https://www.tensorflow.org*, 2021.

[7] Vitis HLS LLVM Source Code and Examples. *https://github.com/Xilinx/HLS*, 2021.

[8] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K Jha. GARNET: A Detailed On-Chip Network Model inside A Full-System Simulator. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[9] Sumit Ahuja et al. Power Estimation Methodology for A High-Level Synthesis Framework. *Int'l Symp. on Quality of Electronic Design (ISQED)*, 2009.

[10] Hussein Al-Omari and Khair Eddin Sabri. New Graph Coloring Algorithms. *American Journal of Mathematics and Statistics*, 2006.

[11] Amazon Web Services, Inc. Amazon EC2 F1 Instances: Enable Faster FPGA Accelerator Development and Deployment in the Cloud. *https://aws.amazon.com/ec2/instance-types/f1/*, 2019.

[12] Jason Helge Anderson and Farid N Najm. Power Estimation Techniques for FPGAs. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 2004.

[13] Apple. Apple M1 Chip. *https://www.apple.com/mac/m1/*, 2021.

[14] Andrés Arévalo, Jaime Niño, German Hernández, and Javier Sandoval. High-Frequency Trading Strategy based on Deep Neural Networks. *International Conference on Intelligent Computing*.

[15] Krste Asanović et al. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 2016.

[16] Krste Asanovic and Andrew Waterman. The RISC-V Instruction Set Manual. In *Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. RISC-V Foundation, 2019.

[17] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. Putting Polyhedral Loop Transformations to Work. In *Languages and Compilers for Parallel Computing*. Springer, 2003.

[18] Yosi Ben-Asher and Nadav Rotem. Automatic Memory Partitioning: Increasing Memory Parallelism via Data Structure Partitioning. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2010.

[19] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V Gratz, and Daniel A Jiménez. Perceptron-Based Prefetch Filtering. *Int'l Symp. on Computer Architecture (ISCA)*, 2019.

[20] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 And Beyond. Technical report, Institute for Formal Models and Verification, Johannes Kepler University, 2011.

[21] Alessandro Bogliolo et al. Regression-Based RTL Power Modeling. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2000.

[22] Olcay Boz. Extracting Decision Trees from Trained Neural Networks. *ACM Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, 2002.

[23] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and Regression Trees*. Wiley Online Library, 1984.

[24] Randal E Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers (TC)*, 1986.

[25] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski.

LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[26] Satrajit Chatterjee. Learning and Memorization. *Int'l Conf. on Machine Learning (ICML)*, 2018.

[27] Zhengping Che, Sanjay Purushotham, Robinder Khemani, and Yan Liu. Distilling Knowledge from Deep Networks with Applications to Healthcare Domain. *arXiv preprint arXiv:1512.03542*, 2015.

[28] Zhengping Che, Sanjay Purushotham, Robinder Khemani, and Yan Liu. Interpretable Deep Models for ICU Outcome Prediction. *AMIA Annual Symposium*, 2016.

[29] Deming Chen et al. High-Level Power Estimation and Low-Power Design Space Exploration for FPGAs. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2007.

[30] H. Chen and M. Shen. A Deep-Reinforcement-Learning-Based Scheduler for FPGA HLS. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2019.

[31] Tianqi Chen and Carlos Guestrin. Xgboost: A Scalable Tree Boosting System. *ACM Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, 2016.

[32] Yu Ting Chen and Jason H Anderson. Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2017.

[33] Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. Rapid Cycle-Accurate Simulator for High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[34] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[35] François Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[36] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 2018.

[37] Alessandro Cilardo and Luca Gallo. Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning. *ACM Trans. on Architecture and Code Optimization (TACO)*, 2015.

[38] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[39] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2011.

[40] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[41] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 1995.

[42] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.

[43] Mark W Craven and Jude W Shavlik. Extracting Tree-Structured Representations of Trained Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 1996.

[44] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[45] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. *Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[46] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research (JMLR)*, 2011.

[47] Masoumeh Ebrahimi, Masoud Daneshtalab, Fahimeh Farahnakian, Juha Plosila, Pasi Liljeberg, Maurizio Palesi, and Hannu Tenhunen. HARAQ: Congestion-Aware Learning Model for Highly Adaptive Routing Algorithm in On-Chip Networks. *Int'l Symp. on Networks-on-Chip (NOCS)*, 2012.

[48] Juan Escobedo and Mingjie Lin. Extracting Data Parallelism in Non-Stencil Kernel Computing by Optimally Coloring Folded Memory Conflict Graph. *Design Automation Conf. (DAC)*, 2018.

[49] Juan Escobedo and Mingjie Lin. Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[50] Juan Escobedo and Mingjie Lin. Exploiting Irregular Memory Parallelism in Quasi-Stencils through Nonlinear Transformation. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[51] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. *Int'l Symp. on Computer Architecture (ISCA)*, 2011.

[52] Nicholas Frosst and Geoffrey Hinton. Distilling a Neural Network into A Soft Decision Tree. *arXiv preprint arXiv:1711.09784*, 2017.

[53] Elba Garza, Samira Mirbagher-Ajorpaz, Tahsin Ahmad Khan, and Daniel A Jiménez. Bit-Level Perceptron Prediction for Indirect Branches. *Int'l Symp. on Computer Architecture (ISCA)*, 2019.

[54] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep Learning*. MIT press Cambridge, 2016.

[55] Google. Pixel Visual Core: Image Processing and Machine Learning on Pixel 2. *https://www.blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/*, 2017.

[56] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu.

A Survey of Deep Learning Techniques for Autonomous Driving. *Journal of Field Robotics*, 2020.

[57] Aditya Grover and Jure Leskovec. node2vec: Scalable Feature Learning for Networks. *ACM Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, 2016.

[58] Aric Hagberg et al. Exploring Network Structure, Dynamics, and Function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[59] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[60] Milad Hashemi, Kevin Swersky, Jamie A Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning Memory Access Patterns. *arXiv preprint arXiv:1803.02329*, 2018.

[61] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *Int'l Conf. on Computer Vision (ICCV)*, 2015.

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[63] Tin Kam Ho. The Random Subspace Method for Constructing Decision Forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1998.

[64] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 1997.

[65] Arthur E Hoerl and Robert W Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 1970.

[66] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied Logistic Regression*. John Wiley & Sons, 2013.

[67] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017.

[68] Hanbin Hu, Qingran Zheng, Ya Wang, and Peng Li. HFMV: Hybridizing Formal Methods and Machine Learning for Verification of Analog and Mixed-Signal Circuits. *Design Automation Conf. (DAC)*, 2018.

[69] Guyue Huang, Jingbo Hu, Yifan He, Jialong Liu, Mingyuan Ma, Zhaoyang Shen, Juejian Wu, Yuanfan Xu, Hengrui Zhang, Kai Zhong, et al. Machine Learning for Electronic Design Automation: A Survey. *arXiv preprint arXiv:2102.03357*, 2021.

[70] IBM. See the Story behind Summit, the World's Most Powerful Supercomputer. *https://www.ibm.com/thought-leadership/summit-supercomputer/*, 2021.

[71] Intel Corporation. Intel High Level Synthesis Compiler Reference Manual. *https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/mnl-hls-reference.pdf*, 2018.

[72] Intel Corporation. Intel® Core™ i9-10900K Processor. *https://www.intel.com/ content/www/us/en/products/processors/core/i9-processors/i9-10900k.html*, 2021.

[73] Intel Corporation. Intel® FPGA SDK for OpenCL™ Software Technology. *https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html*, 2021.

[74] Intel Corporation. Intel® Xeon® W-2295 Processor. *https://www.intel.com/ content/www/us/en/products/processors/xeon/w-processors/w-2295.html*, 2021.

[75] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *Int'l Conf. on Machine Learning (ICML)*, 2015.

[76] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *Int'l Symp. on Computer Architecture (ISCA)*, 2008.

[77] Tommy R Jensen and Bjarne Toft. *Graph Coloring Problems*. John Wiley & Sons, 2011.

[78] Jiajia Jiao, Debjit Pal, Chenhui Deng, and Zhiru Zhang. GLAIVE: Graph Learning Assisted Instruction Vulnerability Estimation. *Design, Automation, and Test in Europe (DATE)*, 2021.

[79] Daniel A Jiménez and Calvin Lin. Dynamic Branch Prediction with Perceptrons. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2001.

[80] Ian Jolliffe. Principal Component Analysis. In *International Encyclopedia of Statistical Science*. Springer, 2011.

[81] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[82] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *Journal on Scientific Computing*, 1998.

[83] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[84] Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907*, 2016.

[85] R Krishnan, G Sivakumar, and P Bhattacharya. Extracting Decision Trees from Trained Neural Networks. *Pattern Recognition*, 1999.

[86] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. *Technical Report, University of Toronto*, 2009.

[87] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[88] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[89] Niels Landwehr, Mark Hall, and Eibe Frank. Logistic Model Trees. *Machine Learning*, 2005.

[90] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for

Lifelong Program Analysis & Transformation. *Int'l Symp. on Code Generation and Optimization (CGO)*, 2004.

[91] Yann LeCun. The MNIST Database of Handwritten Digits. *http://yann. lecun. com/exdb/mnist/*, 2017.

[92] Dongwook Lee et al. Dynamic Power and Performance Back-Annotation for Fast and Accurate Functional Hardware Simulation. *Design, Automation, and Test in Europe (DATE)*, 2015.

[93] Michael M Lee, John Kim, Dennis Abts, Michael Marty, and Jae W Lee. Probabilistic Distance-Based Arbitration: Providing Equality of Service for Many-Core CMPs. *Int'l Symp. on Microarchitecture (MICRO)*, 2010.

[94] Peng Li, Yuxin Wang, Peng Zhang, Guojie Luo, Tao Wang, and Jason Cong. Memory Partitioning and Scheduling Co-Optimization in Behavioral Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2012.

[95] Wensong Li, Fan Yang, Hengliang Zhu, Xuan Zeng, and Dian Zhou. An Efficient Memory Partitioning Approach for Multi-Pattern Data Access via Data Reuse. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 2019.

[96] Zhe Lin, Jieru Zhao, Sharad Sinha, and Wei Zhang. HL-Pow: A Learning-Based Power Modeling Framework for High-Level Synthesis. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2020.

[97] S. Liu, F. C. Lau, and B. C. Schafer. Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration. *Design Automation Conf. (DAC)*, 2019.

[98] Xuan Liu, Xiaoguang Wang, and Stan Matwin. Improving the Interpretability of Deep Neural Networks with Knowledge Distillation. *IEEE Int'l Conf. on Data Mining Workshops (ICDMW)*, 2018.

[99] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Srikant Bharadwaj, Gabe Black, et al. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[100] Ningning Ma et al. Shufflenet v2: Practical Guidelines for Efficient CNN Architecture Design. *arXiv preprint arXiv:1807.11164*, 2018.

[101] Yuzhe Ma, Zhuolun He, Wei Li, Lu Zhang, and Bei Yu. Understanding Graphs in EDA: From Shallow to Deep Learning. *Int'l Symp. on Physical design (ISPD)*, 2020.

[102] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier Nonlinearities Improve Neural Network Acoustic Models. *Int'l Conf. on Machine Learning (ICML)*, 2013.

[103] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research (JMLR)*, 2008.

[104] Llew Mason et al. Boosting Algorithms as Gradient Descent. *Advances in Neural Information Processing Systems (NeurIPS)*, 2000.

[105] Chenyue Meng, Shouyi Yin, Peng Ouyang, Leibo Liu, and Shaojun Wei. Efficient Memory Partitioning for Parallel Data Access in Multidimensional Arrays. *Design Automation Conf. (DAC)*, 2015.

[106] Julian Francis Miller and Simon L Harding. Cartesian Genetic Programming. *The 10th Annual Conference Companion on Genetic and Evolutionary Computation*, 2008.

[107] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Sungmin Bae, et al. Chip Placement with Deep Reinforcement Learning. *arXiv preprint arXiv:2004.10746*, 2020.

[108] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.

[109] V. Mrazek, M. A. Hanif, Z. Vasicek, L. Sekanina, and M. Shafique. autoAx: An Automatic Design Space Exploration and Circuit Building Methodology utilizing Libraries of Approximate Components. *Design Automation Conf. (DAC)*, 2019.

[110] Janani Mukundan and Jose F Martinez. MORSE: Multi-Objective Reconfigurable Self-Optimizing Memory Scheduler. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2012.

[111] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.

[112] Vinod Nair and Geoffrey E Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. *Int'l Conf. on Machine Learning (ICML)*, 2010.

[113] NVIDIA Corporation. NVIDIA DRIVE AGX. *https://www.nvidia.com/en-us/self-driving-cars/drive-platform/hardware/*, 2019.

[114] NVIDIA Corporation. RTX: It's On: Ultimate Ray Tracing & AI. *https://www.nvidia.com/en-us/geforce/rtx/*, 2021.

[115] NVIDIA Corporation. Tensor Cores: Versatility for HPC & AI. *https://www.nvidia.com/en-us/data-center/tensor-cores/*, Jan 2021.

[116] Arlindo L Oliveira and Alberto Sangiovanni-Vincentelli. Learning Complex Boolean Functions: Algorithms and Applications. *Advances in Neural Information Processing Systems (NeurIPS)*, 1994.

[117] OpenAI. OpenAI Spinning Up. *https://spinningup.openai.com/en/latest/user/introduction.html*, 2021.

[118] OpenCores.org. Fixed Point Math Library for Verilog :: Manual. *https://opencores.org/project/verilog_fixed_point_math_library/manual*, 2018.

[119] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[120] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)*, 2011.

[121] F. Pedregosa et al. Scikit-Learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)*, 2011.

[122] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic Locality and Context-Based Prefetching using Reinforcement Learning. *Int'l Symp. on Computer Architecture (ISCA)*, 2015.

[123] Boris T Polyak. Some Methods of Speeding up the Convergence of Itera-

tion Methods. *USSR Computational Mathematics and Mathematical Physics*, 1964.

[124] Matthew Poremba, Itir Akgun, Jieming Yin, Onur Kayiran, Yuan Xie, and Gabriel H Loh. There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes. *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[125] Shubham Rai, Walter Lau Neto, Yukio Miyasaka, Xinpei Zhang, Mingfei Yu, Qingyang Yi Masahiro Fujita, Guilherme B. Manske, Matheus F. Pontes, Leomar S. da Rosa Junior, Marilton S. de Aguiar, Paulo F. Butzen, Po-Chun Chien, Yu-Shan Huang, Hoa-Ren Wang, Jie-Hong R. Jiang, Jiaqi Gu, Zheng Zhao, Zixuan Jiang, David Z. Pan, Brunno A. de Abreu, Isac de Souza Campos, Augusto Berndt, Cristina Meinhardt, Jonata T. Carvalho, Mateus Grellert, Sergio Bampi, Aditya Lohana, Akash Kumar, Wei Zeng, Azadeh Davoodi, Rasit O. Topaloglu, Yuan Zhou, Jordan Dotzel, Yichi Zhang, Hanyu Wang, Zhiru Zhang, Valerio Tenace, Pierre-Emmanuel Gaillardon, Alan Mishchenko, and Satrajit Chatterjee. Logic Synthesis Meets Machine Learning: Trading Exactness for Generalization, 2020.

[126] Srivaths others Ravi. Efficient RTL Power Estimation for Large Designs. *Int'l Conf. on VLSI Design (VLSID)*, 2003.

[127] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. *Int'l Symp. on Workload Characterization (IISWC)*, 2014.

[128] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2012.

[129] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 1987.

[130] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *Int'l Journal of Computer Vision (IJCV)*, 2015.

[131] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. Improved Protein Structure Prediction using Potentials from Deep Learning. *Nature*, 2020.

[132] Yakun Sophia Shao et al. Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. *ACM SIGARCH Computer Architecture News*, 2014.

[133] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying Deep Learning to the Cache Replacement Problem. *Int'l Symp. on Microarchitecture (MICRO)*, 2019.

[134] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 2016.

[135] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the Game of Go without Human Knowledge. *Nature*, 2017.

[136] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[137] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research (JMLR)*, 2014.

[138] Nitish Kumar Srivastava, Steve Dai, Rajit Manohar, and Zhiru Zhang. Accelerating Face Detection on Programmable SoC Using C-Based Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[139] Jincheng Su, Fan Yang, Xuan Zeng, and Dian Zhou. Efficient Memory Partitioning for Parallel Data Access via Data Reuse. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[140] Dam Sunwoo et al. PrEsto: An FPGA-Accelerated Power Estimation Methodology for Complex Systems. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2010.

[141] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the Importance of Initialization and Momentum in Deep Learning. *Int'l Conf. on Machine Learning (ICML)*, 2013.

[142] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to Sequence Learning with Neural Networks. *arXiv preprint arXiv:1409.3215*, 2014.

[143] Aysa Fakheri Tabrizi, Logan Rakai, Nima Karimpour Darav, Ismail Bustany, Laleh Behjat, Shuchang Xu, and Andrew Kennings. A Machine Learning Framework to Identify Detailed Routing Short Violations from a Placed Netlist. *Design Automation Conf. (DAC)*, 2018.

[144] Ryutaro Tanno, Kai Arulkumaran, Daniel Alexander, Antonio Criminisi, and Aditya Nori. Adaptive Neural Trees. *Int'l Conf. on Machine Learning (ICML)*, 2019.

[145] Y Tatsumi and HJ Mattausch. Fast Quadratic Increase of Multiport-Storage-Cell Area with Port Number. *Electronics Letters*, 1999.

[146] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron Learning for Reuse Prediction. *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[147] Robert Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 1996.

[148] Yaman Umuroglu, Yash Akhauri, Nicholas J Fraser, and Michaela Blott. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. *arXiv preprint arXiv:2004.03021*, 2020.

[149] E. Ustun, S. Xiang, J. Gui, C. Yu, and Z. Zhang. LAMDA: Learning-Assisted Multi-stage Autotuning for FPGA Design Closure. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[150] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. Accurate Operation Delay Prediction for FPGA HLS using Graph Neural Networks. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2020.

[151] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *arXiv preprint arXiv:1706.03762*, 2017.

[152] Paul Viola and Michael J Jones. Robust Real-Time Face Detection. *Int'l Journal of Computer Vision (IJCV)*, 2004.

[153] Alvin Wan, Lisa Dunlap, Daniel Ho, Jihan Yin, Scott Lee, Henry Jin, Suzanne Petryk, Sarah Adel Bargal, and Joseph E Gonzalez. NBDT: Neural-Backed Decision Trees. *arXiv preprint arXiv:2004.00221*, 2020.

[154] Erwei Wang, James J Davis, Peter YK Cheung, and George A Constantinides. LUTNet: Rethinking Inference in FPGA Soft Logic. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[155] Shuo Wang, Yun Liang, and Wei Zhang. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. *Design Automation Conf. (DAC)*, 2017.

[156] Yuxin Wang, Peng Li, and Jason Cong. Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2014.

[157] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory Partitioning for Multidimensional Arrays in High-Level Synthesis. *Design Automation Conf. (DAC)*, 2013.

[158] Z. Wang and B. C. Schafer. Machine Learning to Set Meta-Heuristic Specific Parameters for High-Level Synthesis Design Space Exploration. *Design Automation Conf. (DAC)*, 2020.

[159] Henry S Warren. *Hacker's Delight*. Pearson Education, 2013.

[160] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 1992.

[161] Anson Wong. Deep & Classical Reinforcement Learning + Machine Learning Examples in Python. *https://github.com/ankonzoid/LearningX*, 2020.

[162] Nan Wu, Yuan Xie, and Cong Hao. IronMan: GNN-Assisted Design Space Exploration in High-Level Synthesis via Reinforcement Learning. *arXiv preprint arXiv:2102.08138*, 2021.

[163] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming

He. Aggregated Residual Transformations for Deep Neural Networks. *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[164] Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. RouteNet: Routability Prediction for Mixed-Size Designs using Convolutional Neural Network. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[165] Xilinx. Versal: The First Adaptive Compute Acceleration Platform (ACAP). *https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf*, 2018.

[166] Xilinx. PYNQ - Python Productivity for Zynq. *http://www.pynq.io*, 2021.

[167] Xilinx. Vitis Unified Software Platform. *https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html*, 2021.

[168] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf*, 2018.

[169] Jianlei Yang et al. Early Stage Real-Time SoC Power Estimation using RTL Instrumentation. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2015.

[170] W. Ye, M. B. Alawieh, Y. Lin, and D. Z. Pan. LithoGAN: End-to-End Lithography Modeling with Generative Adversarial Networks. *Design Automation Conf. (DAC)*, 2019.

[171] Jieming Yin, Subhash Sethumurugan, Yasuko Eckert, Chintan Patel, Alan Smith, Eric Morton, Mark Oskin, Natalie Enright Jerger, and Gabriel H Loh. Experiences with ML-Driven Design: A NoC Case Study. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2020.

[172] Quanzeng You, Hailin Jin, Zhaowen Wang, Chen Fang, and Jiebo Luo. Image Captioning with Semantic Attention. *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[173] Cunxi Yu and Zhiru Zhang. Painting on Placement: Forecasting Routing Congestion using Conditional Generative Adversarial Nets. *Design Automation Conf. (DAC)*, 2019.

[174] Yuan Zeng and Xiaochen Guo. Long Short Term Memory Based Hardware Prefetcher: A Case Study. *Int'l Symp. on Memory Systems (MEMSYS)*, 2017.

[175] Jeff Jun Zhang and Siddharth Garg. FATE: Fast and Accurate Timing Error Prediction Framework for Low Power DNN Accelerator Design. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[176] Liheng Zhang, Charu Aggarwal, and Guo-Jun Qi. Stock Price Prediction via Discovering Multi-Frequency Trading Patterns. *ACM Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, 2017.

[177] X. Zhang, J. Shiely, and E. F. Y. Young. Layout Pattern Generation and Legalization with Generative Learning Models. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2020.

[178] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[179] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. GRANNITE: Graph Neural Network Inference for Transferable Power Estimation. *Design Automation Conf. (DAC)*, 2020.

[180] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A Platform-Based ESL Synthesis System. In *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2008.

[181] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2017.

[182] Hao Zheng and Ahmed Louri. An Energy-Efficient Network-on-Chip Design using Reinforcement Learning. *Design Automation Conf. (DAC)*, 2019.

[183] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. Lin-Analyzer: A High-Level Performance Analysis Tool for FPGA-Based Accelerators. *Design Automation Conf. (DAC)*, 2016.

[184] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. A New Approach

to Automatic Memory Banking using Trace-Based Address Mining. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[185] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[186] Yuan Zhou, Haoxing Ren, Yanqing Zhang, Ben Keller, Brucek Khailany, and Zhiru Zhang. PRIMAL: Power Inference Using Machine Learning. *Design Automation Conf. (DAC)*, 2019.

[187] Yu Zou and Mingjie Lin. Graph-Morphing: Exploiting Hidden Parallelism of Non-Stencil Computation in High-Level Synthesis. *Design Automation Conf. (DAC)*, 2019.