Distilling Arbitration Logic from Traces using Machine Learning: A Case Study on NoC

Yuan Zhou
Cornell University
Ithaca, USA
yz882@cornell.edu

Hanyu Wang†
Shanghai Jiao Tong University
Shanghai, China
whynull@sjtu.edu.cn

Jieming Yin
Lehigh University
Bethlehem, USA
yin@lehigh.edu

Zhiru Zhang
Cornell University
Ithaca, USA
zhiruz@cornell.edu

***Abstract*— Arbitration logic is extensively used in modern computer architectures to dynamically determine how shared hardware resources are allocated or accessed. Recent work has shown that machine learning techniques can learn non-obvious yet effective arbitration policies, which in simulation demonstrate superior performance over human-designed heuristics. However, existing methods based on deep learning are too expensive to be directly implemented as an arbitration unit in hardware. While some prior efforts managed to manually analyze and reduce a deep learning model into relatively small circuits in certain cases, such ad hoc and labor-intensive approaches cannot easily generalize. In this work, we propose a new methodology to automatically "distill" the arbitration logic from simulation traces. Starting by training a deep learning model, we leverage tree-based models as a bridge to convert the more complex model to a compact logic implementation. This paper presents a case study of the proposed methodology on a network-on-chip port arbitration task. Compared with an array of combinational multipliers that exactly computes the neural network output, our arbitration logic achieves up to 282x area reduction without significant performance degradation. Under the training traffic, our arbitration logic achieves up to 64x reduction in average packet latency and up to 5% increase in network throughput over the FIFO arbitration policy. The distilled arbitration policy is also able to generalize to different injection rates and traffic patterns.**

***Index Terms*—Network-on-Chip, Machine Learning**

I. Introduction

Resource sharing is common in modern computer systems to balance performance and hardware cost. In order to maximize utilization and achieve 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, as modern computer architectures become increasingly complex and heterogeneous, it is 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 [7], [11] and memory controllers using table-based reinforcement learning algorithms [10], [17]. 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 [22], [23], prefetching [8], [31], network-on-chip (NoC) packet arbitration [30], and NoC dynamic frequency-voltage scaling [32]. In many cases, DL techniques have been shown to achieve a superior performance in simulation. However, it is not always feasible to directly use a DL accelerator as an arbitration unit due to its high overhead in both latency and area. As a 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 therefore needed to fill the missing link of applying DL to arbitration problems in computer architecture.

†This work was performed while Hanyu was a (remote) research intern with Cornell University.

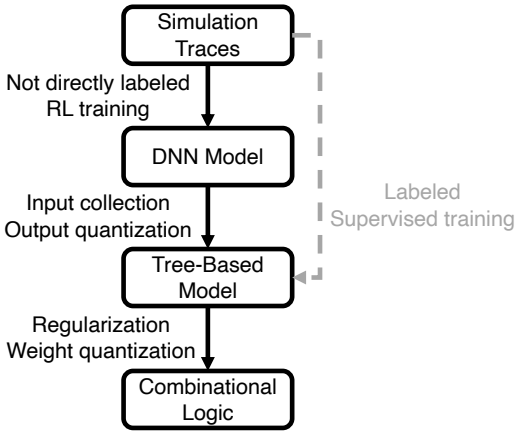


Fig. 1: Proposed flow of distilling logic from traces. For supervised tasks, tree models can be directly trained from simulation traces. This paper focuses on converting deep learning models to circuits.

In this work, we propose to tackle this challenge by leveraging tree-based models as a bridge between neural network models and circuit implementations. Figure 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 learning arbitration logic because arbitration policies can be effectively learned using deep reinforcement learning. 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 [3], random forests [9], or model trees [13] 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 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 [14], [21]. 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 threefold:

- We are the first to propose a methodology for automatically generating compact, application-specific arbitration logic from simulation traces.
- 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.
- 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 and traffic patterns. Compared with the DL agent, the generated arbitration logic achieves comparable performance with up to 282× area reduction.

II. Related Work

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 [11]. Similar ideas are later applied to cache replacement [24] and prefetching [2]. Another line of research focuses on using reinforcement learning (RL) to solve typical arbitration problems, including memory request scheduling [10], NoC routing policy [6], and cache prefetching [20]. 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 size of the Q-tables is subject to area and power constraints, which limits the complexity of the policies these approaches can learn.

With the development of deep learning, recent works explore the opportunity of applying deep learning techniques to computer architecture. Hashemi *et al.* performed a pure theoretical study of applying long-short term memory (LSTM) to cache prefetching [8]. Zeng *et al.* explored a similar idea, but embedded a small LSTM accelerator into the prefetcher to perform online training and inference [31]. 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 [23]. Zheng *et al.* proposed to use deep Q-Networks (DQN) for dynamic frequency-voltage scaling (DVFS) in a NoC [32]. 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 [30]. 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. Similarly, Sethumurugan *et al.* derived a cost-effective cache replacement policy from DQN by manually analyzing the trained neural network [22]. Notice that when implementing the DL model as hardware, existing works either directly use an expensive accelerator, or require extensive manual analysis and optimization to derive a compact hardware implementation. Our work proposes an automated approach to generate arbitration logic from data, which contains a core step of converting a DL model to logic. Specifically, our case study shows how the proposed approach generates efficient arbitration logic with a setup similar to [30].

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 [27] provides an efficient way of implementing binarized neural networks [5] 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 [26]. These techniques are designed for mapping low-precision networks onto FPGAs, while our approach maps full-precision networks to ASICs.

III. Background

A. NoC Arbitration

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 upon contention. NoC arbitration policy is critical for the network's performance — a good policy provides better fairness and achieves low 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 policy 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-based policy is considered one of the best policies, its hardware cost is largely impractical for use in on-chip routers [30].

NoC arbitration is a well-defined problem suitable for RL. Yin *et al.* presented 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 [30]. 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.

B. Reinforcement Learning

Reinforcement learning is an ML technique commonly used for decision making problems. During training, the agent interacts with the environment by observing the environment's states and immediate rewards, and learns a policy that maximizes the long-term cumulative reward. A numerical, scalar reward is returned by the environment for every action performed by the agent, which is then used by the agent to update its policy. The agent observes abundant data during training through repeated interaction with the environment. Although it is not guaranteed to cover all corner cases, the important common cases will likely be covered by running a sufficient number of episodes.

A popular reinforcement learning algorithm is Q-learning [28]. In Q-learning, the agent tries to learn a Q-value $Q(s, a)$ for each state-action pair (s, a), where $Q(s, a)$ corresponds to the cumulative reward of performing action a under state s. Traditionally, when the state space and action space are limited, a Q-table can be used to store the learned Q-values. When the state space becomes large, deep Q-learning [16] provides a more tractable solution, where a neural network agent is trained to predict the Q-values. While the neural network agent in deep Q-learning performs a regression task of predicting the Q-values, it can also be treated as a classifier because the policy always selects the action with the highest predicted reward after being deployed. When converting the agent to logic, the more hardware-friendly view should be selected. In our case study, we

consider the agent as a regressor and use an additional select-max unit to choose the best action.

C. Tree-Based ML Models

Tree-based ML models approximate the target function by repeatedly partitioning the input space. Common tree-based models include decision trees [3], random forests [9], gradient boosted trees such as XGBoost [4], and model trees [13]. In this paper we focus on decision trees and linear model trees, because these models can be efficiently implemented in hardware with proper regularization and quantization.

During training, a decision tree or linear 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. 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. For decision trees this function is constant, while for linear model trees this function is a linear function with respect to the input features. As a result, decision trees naturally learn a step function, while linear model trees learn piecewise linear functions. The linear models at the leaves of linear model trees can be fit using common linear regression or classification techniques. In this work we use LASSO linear regression [25] to minimize the number of features used at each leaf node.

IV. Distilling Arbitration Logic from Data

In this section, we introduce the details of our logic distillation process. As shown in Figure 1, starting from running simulation, the 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.

A. Step 1: Learning an Arbitration Policy

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

We use DQN to learn an arbitration policy for NoC routers. Figure 2 shows the architecture of routers in our simulation framework, where the same agent is shared by all VCs and all routers. Upon arbitration, each candidate VC queries the agent and the agent returns a priority score. Similar to [30], the agent uses four features of the packet to make predictions: 1) local age, i.e., amount of time the packet 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 [0, 1] 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

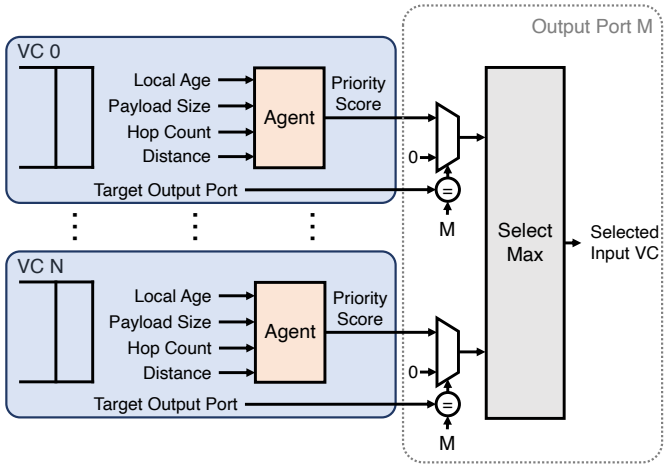


Fig. 2: Architecture diagram for router arbitration.

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 reward function optimizes for network latency, designers can emphasize other 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 V for more details on the hyperparameters and training dynamics.

B. Step 2: Selecting the Tree Model

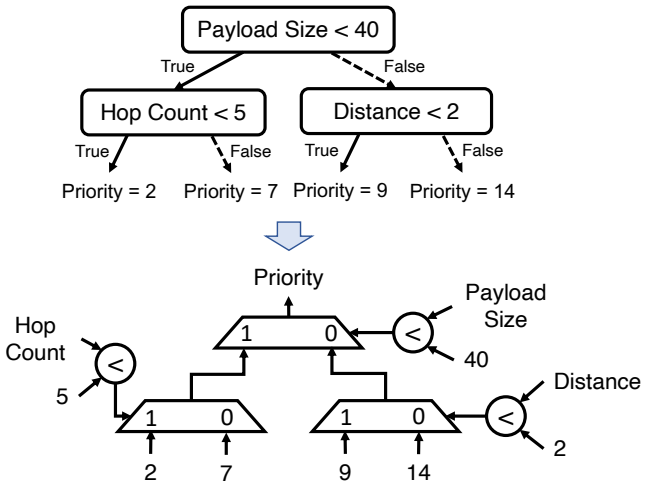
As introduced in Section III-C, decision trees and linear model trees are of particular interest in our approach because they can be easily converted to logic. However, these two types of models intrinsically learn different types of functions: decision trees learn step functions, while linear model trees learn piecewise linear functions. 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 for approximating the outputs of these neural networks. On the other hand, decision trees are more suitable for very nonlinear target functions.

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 V-A).

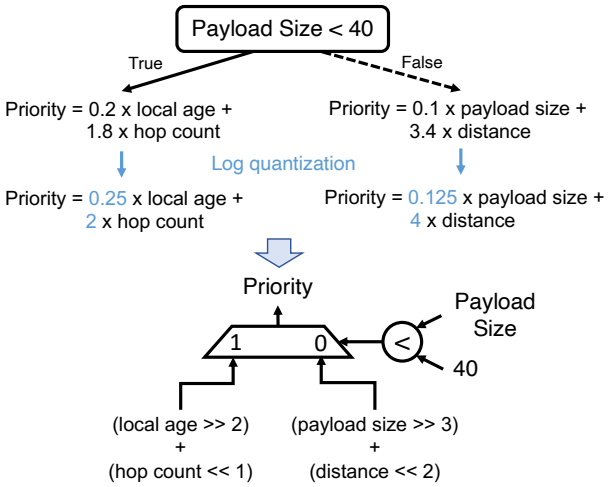
C. Step 3: Generating Implementable Logic

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. If the number of

[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.



(a) Convert decision tree to logic.



(b) Convert linear model tree to logic.

Fig. 3: Convert tree models to combinational logic (log quantization is applied to linear model tree).

input features is small, the inputs can be collected by exhaustively including all possible input combinations. Otherwise, inputs can also be collected from simulation, which 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 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

[2]The predicted values of the linear models might be negative due to the bias. In this case, we can either use signed comparison at the select-max unit, or add a constant to all leaves such that the prediction is always non-negative.

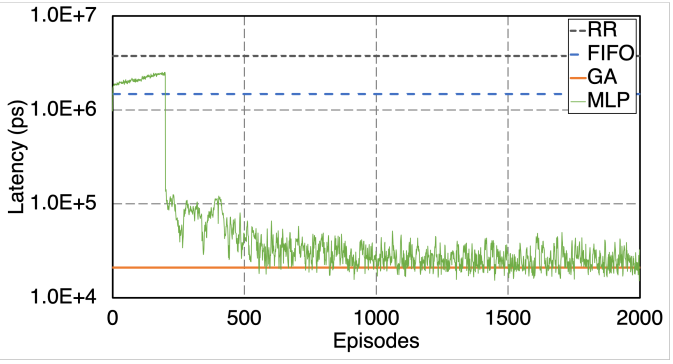


Fig. 4: Training dynamics of the MLP agent and comparisons with different policies — `FIFO`: local-age-based, `RR`: round-robin, `GA`: global-age-based, `MLP`: MLP agent.

simplify the logic, we perform log quantization 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.

V. Case Study: NoC Arbitration Policy

In this section we present a detailed case study on NoC arbitration policy. We use the Garnet [1] network model in gem5 [15] as our simulation platform, and modify Garnet for data collection and training. The training of RL agents, tree models, and the conversion to logic are implemented in Python leveraging PyTorch [18], sci-kit learn [19], and an open-source implementation of model trees [29]. 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 IV-A, 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 (average packet latency increases dramatically after this point) for global-age-based policy. As shown in Figure 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 of sixteen neurons.

During training, we launch gem5 ten times, where at each launch we warm up the network 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[3]. Figure 4

[3]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 [12] with an initial learning rate of 0.001. Exploration rate $\epsilon = 0.9e^{-\tau/500}$, where τ is the number of trained episodes.

Algorithm 1 LMT Arbitration Policy

Input: local age, payload size, hop count, distance
Output: Priority
if hop count $\leq$ 5 **then**
    Priority = (local age $\gg$ 3) + (payload size $\gg$ 3) + (hop count $\ll$ 1) + (distance $\gg$ 1) + 9
else
    Priority = (local age $\gg$ 2) + (payload size $\gg$ 1) + (hop count $\ll$ 2) + distance - 20
end if

Fig. 5: Arbitration policy learned by linear model tree with a max depth of one. All weights are quantized to powers of two.

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.

A. Area and Performance of the Distilled Arbitration Logic

Table I shows the performance and area comparison of different arbitration policies. We also include the manually constructed arbitration logic from [30][4]. 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 I 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$\times$ area reduction with slight performance degradation, while linear model trees achieve up to 282$\times$ 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$\times$ 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 40μm^2 area. As a reference, an eight-bit adder consumes ~17μm^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 their decision tree counterparts, which indicates that piecewise linear priority functions are more suitable for NoC arbitration. Interestingly, the model trees can achieve lower latency than the original neural network agent. We conjecture that the neural network agent might have overfit during training, while the conversion to linear model trees applied a proper regularization effect.

B. Analysis of the Distilled Arbitration Logic

Figure 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 [30], 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.

[4]Priority = (local age $\ll$ 1) + (hop count $\gg$ 1)

TABLE I: Performance and area comparison of different arbitration policies — Performance is measured under Uniform Random traffic with an injection rate of 0.32. `DT`: decision tree; `LMT`: linear model tree.

Model	Avg. Packet Latency (ps)	Area (μm^2)
MLP	25659 $\pm$ 346	11446
DT (no regularization)	27015 $\pm$ 1174	725.8
DT (max depth = 12)	27241 $\pm$ 263	684.1
DT (max depth = 8)	25655 $\pm$ 597	247.7
DT (max depth = 4)	77833 $\pm$ 23931	28.2
LMT (max depth = 4)	23146 $\pm$ 222	214.0
LMT (max depth = 3)	24029 $\pm$ 171	112.9
LMT (max depth = 2)	24537 $\pm$ 494	73.7
LMT (max depth = 1)	23354 $\pm$ 373	45.9
LMT (max depth = 0)	24256 $\pm$ 518	19.7
FIFO	2113339 $\pm$ 50046	0.0
Manually constructed [30]	2056407 $\pm$ 54344	8.6
Oracle (global-age)	21492 $\pm$ 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.

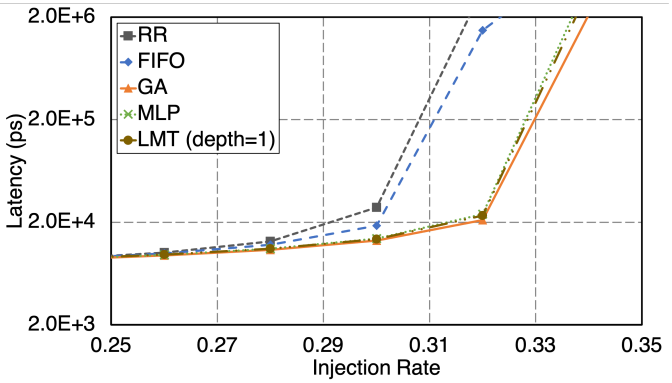


Fig. 6: Performance of different policies under Uniform Random traffic — `LMT (depth=1)`: distilled arbitration logic from linear model tree with depth one.

C. Generalization to Different Injection Rates

Figure 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 [30] 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 V-A. Under the same traffic pattern, our distilled arbitration logic is able to generalize in situations of both light and heavy

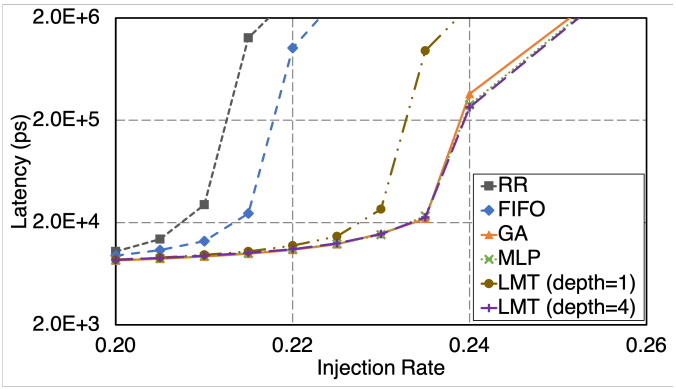


Fig. 7: Performance of different policies under Transpose traffic pattern (MLP and LMT are trained under uniform random traffic).

traffic and consistently outperform the baselines. The distilled policy improves the NoC throughput by 4.9% compared to FIFO.

D. Generalization to Different Traffic Patterns

Figure 7 shows the performance comparison between different arbitration policies under Transpose traffic pattern, where node (x, y) only communicates with node (y, x). Again, the manual policy from [30] (not shown in Figure 7) achieves similar performance with FIFO. While the MLP agent is trained using uniform random traffic, it generalizes very well to this new traffic pattern. The performance of the MLP agent is close to the global-age-based policy and significantly better than FIFO and round-robin. Linear model tree with depth of one also performs quite well, but it does have a noticeable performance gap with the global-age-based policy and the MLP agent. 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, the Transpose traffic pattern requires a more sophisticated priority function because it has more packets with larger hop count. With a linear model tree of depth four, the generated arbitration logic performs nearly identical to the MLP agent.

VI. Conclusion and Future Work

We have presented a novel methodology of automatically distilling arbitration logic from simulation data, and a detailed case study on NoC arbitration. Our case study shows that by leveraging tree-based models as a bridge, deep learning models can be converted to efficient combinational logic. The distilled arbitration logic can achieve significant area savings compared with a straightforward datapath for computing neural network outputs. Regularization schemes can be applied to the tree models as a trade-off between model accuracy and area consumption.

In this work we illustrated the effectiveness of our methodology on a small network with synthetic traffic patterns as a proof-of-concept. For future work, we plan to further evaluate our technique on larger and more realistic networks. Such networks may provide more generic features about input packets, which can be effectively utilized by the RL agent and tree-based models. While the learned arbitration logic can accommodate a different traffic pattern as shown in Section V-D, it is not guaranteed that an RL agent can generalize well to all unseen circumstances. As a result, a promising future direction is to design an efficient reconfigurable implementation of the RL agent so that the arbitration logic can be dynamically reconfigured at run time to adapt to different traffic patterns.

Acknowledgements

We thank the anonymous reviewers for their feedback on the earlier version of this manuscript. This research was supported in part by NSF Award #1909661.

References

[1] N. Agarwal et al. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. *ISPASS*, 2009.
[2] E. Bhatia et al. Perceptron-Based Prefetch Filtering. *ISCA*, 2019.
[3] L. Breiman et al. *Classification and Regression Trees*. CRC press, 1984.
[4] T. Chen and C. Guestrin. Xgboost: A Scalable Tree Boosting System. *KDD*, 2016.
[5] M. Courbariaux et al. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv preprint arXiv:1602.02830*, 2016.
[6] M. Ebrahimi et al. HARAQ: Congestion-Aware Learning Model for Highly Adaptive Routing Algorithm in On-Chip Networks. *NOCS*, 2012.
[7] E. Garza and otheres. Bit-Level Perceptron Prediction for Indirect Branches. *ISCA*, 2019.
[8] M. Hashemi et al. Learning Memory Access Patterns. *arXiv preprint arXiv:1803.02329*, 2018.
[9] T. K. Ho. The Random Subspace Method for Constructing Decision Forests. *IEEE TPAMI*, 20(8):832–844, 1998.
[10] E. Ipek et al. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *ISCA*, 2008.
[11] D. A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. *HPCA*, 2001.
[12] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[13] N. Landwehr et al. Logistic Model Trees. *Machine Learning*, 59(1-2):161–205, 2005.
[14] M. M. o. Lee. Probabilistic Distance-Based Arbitration: Providing Equality of Service for Many-Core CMPs. 2010.
[15] J. Lowe-Power et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.
[16] V. Mnih et al. Playing Atari with Deep Reinforcement Learning. *arXiv preprint arXiv:1312.5602*, 2013.
[17] J. Mukundan and J. F. Martinez. MORSE: Multi-Objective Reconfigurable Self-Optimizing Memory Scheduler. *HPCA*, 2012.
[18] A. Paszke et al. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *NeurIPS*, 2019.
[19] F. Pedregosa et al. Scikit-Learn: Machine Learning in Python. *JMLR*, 12:2825–2830, 2011.
[20] L. Peled et al. Semantic Locality and Context-Based Prefetching using Reinforcement Learning. *ISCA*, 2015.
[21] M. Poremba et al. There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes. *ISCA*, 2017.
[22] S. Sethumurugan, J. Yin, and J. Sartori. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. *HPCA*, 2021.
[23] Z. o. Shi. Applying Deep Learning to the Cache Replacement Problem. *MICRO*, 2019.
[24] E. Teran et al. Perceptron Learning for Reuse Prediction. *MICRO*, 2016.
[25] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.
[26] Y. Umuroglu et al. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. *arXiv preprint arXiv:2004.03021*, 2020.
[27] E. Wang et al. LUTNet: Rethinking Inference in FPGA Soft Logic. *FCCM*, 2019.
[28] C. J. Watkins and P. Dayan. Q-Learning. *Machine Learning*, 8(3-4):279–292, 1992.
[29] A. Wong. Deep & Classical Reinforcement Learning + Machine Learning Examples in Python. *https://github.com/ankonzoid/LearningX*, 2020.
[30] J. Yin et al. Experiences with ML-Driven Design: A NoC Case Study. *HPCA*, 2020.
[31] Y. Zeng and X. Guo. Long Short Term Memory Based Hardware Prefetcher: A Case Study. *MemSys*, 2017.
[32] H. Zheng and A. Louri. An Energy-Efficient Network-On-Chip Design using Reinforcement Learning. *DAC*, 2019.