Experiences with ML-Driven Design: A NoC Case Study

Jieming Yin* Subhash Sethumurugan** Yasuko Eckert* Chintan Patel* Alan Smith*
Eric Morton* Mark Oskin*† Natalie Enright Jerger‡ Gabriel H. Loh*

*Advanced Micro Devices, Inc.
{first.last}@amd.com

†University of Washington
oskin@cs.washington.edu

**University of Minnesota, Twin Cities
sethu018@umn.edu

‡University of Toronto
enright@ece.utoronto.ca

ABSTRACT

There has been a lot of recent interest in applying machine learning (ML) to the design of systems, which purports to aid human experts in extracting new insights leading to better systems. In this work, we share our experiences with applying ML to improve one aspect of networks-on-chips (NoC) to uncover new ideas and approaches, which eventually led us to a new arbitration scheme that is effective for NoCs under heavy contention. However, a significant amount of human effort and creativity was still needed to optimize just one aspect (arbitration) of what is only one component (the NoC) of the overall processor. This leads us to conclude that much work (and opportunity!) remains to be done in the area of ML-driven architecture design.

1. INTRODUCTION

In recent years, our community has shown increasing interest in applying Machine learning (ML) to improve computer system designs. There are now designated conferences and workshops such as MLSys and AIDArch organized around such ML-driven approaches. ML can potentially augment human expert intelligence to extract richer insights, find more effective optimizations, and provide better decision making for complex systems. In the computer-architecture field alone, there is already some work utilizing ML to improve branch predictors [1], memory controllers [2], reuse prediction [3], prefetchers [4], and dynamic voltage and frequency scaling (DVFS) management for NoCs [5, 6].

Given the excitement around ML for system design, we wanted to try wielding this "hammer" for ourselves, and we set off in search for some "nails." There are many aspects of a processor that could potentially benefit from ML-aided design. We decided to consider NoC arbitration policies as a test case because it is a sufficiently simple-to-define problem where we hoped applying ML would be tractable, but arbitration still plays a critical role in NoC performance. Even restricting our attention to just one specific aspect (arbitration) of only one component (the NoC) of the processor, we still found that there were several areas where ML falls short of being a panacea for all our architecture challenges. That said, despite these shortcomings, we did find that ML was useful and guided us to new and better solutions. The objective of this paper is to share with the community our experiences with ML-guided design, both to encourage continued research and development into this promising approach, as well as to draw attention to points along the journey where ML cannot (yet) replace human creativity and innovation.

From our experience, ML was a useful tool to process a large amount of data. We collected the NoC router states over a large number of simulated cycles. This includes states on the multiple input buffers, where requests are coming from, where they are headed to, the message types/classes, and more. Given the millions of cycles of simulation, it is impractical for a human to manually dig through so much data with any hope of extracting useful patterns or identifying interesting behaviors. For our chosen problem of NoC arbitration, we found that reinforcement learning (RL) was a good match to our problem definition. We could let the RL agent loose on our problem, and (hopefully) observe what new algorithms and strategies that it comes up with to improve NoC arbitration. The good news is that, with some effort, we were able to gain some useful and non-obvious (at least to us) insights that led to an improved NoC arbiter.

Also from our experience, ML for system design (at least in its current state) still leaves much to be desired in a few areas. The first is in hyperparameter tuning to get the RL agent to perform well. While this is a well-known challenge with ML in general, it still represents required human effort. The second area is the *interpretability* of neural networks (NN). While our trained RL agent was able to improve NoC arbitration, it required substantial human effort to extract even just some hints as to why/when/how the RL agent was behaving so. ML interpretability remains a large and active research area in the broader ML community [7], and continued advances are needed to maximize the benefit and impact of ML-aided computer architecture design.

The third area where we experienced a "gap" in ML-aided design methodology is in going from the neural network to a practical solution. In many other areas where ML is being successfully applied (e.g., image recognition), the complex deep neural networks can be executed with software implementations. However, for the vast majority of potential computer architecture applications, decisions need to be made on the order of at most a few clock cycles. In such timing-constrained (and silicon area-constrained) environments, even executing a single layer of a neural network is unlikely to be feasible. One can draw analogies to the pioneering work on perceptron branch predictors [1] where an ML technique provided the valuable insight that very long branch histories provide critical information for improving prediction accuracy, but then it took over an additional decade of creative work by the community to convert that insight into the implementable "neural inspired" predictors that we see today [8]. In our work, we identified some useful behaviors from our trained RL agent,

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00058

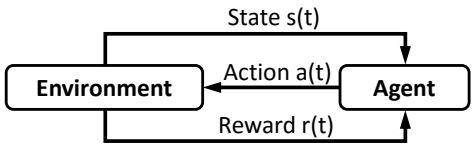


Figure 1: Conceptual diagram of reinforcement learning.

but we were then left on our own to derive something that captured the essence of the neural network's behaviors while being suitably simple for industrial application.

The goals of this work are neither to over-hype ML for computer architecture design, nor to condemn the overall approach. We do see a lot of promise in further research in these directions, but we also want the community to approach this type of work with "eyes wide open" about how ML may augment, but not replace, the role of human computer architects.

2. BACKGROUND

In this section, we provide a brief background on NoC arbitration and reinforcement learning. We refer the interested reader to other sources [9, 10, 11, 12] for more details.

2.1 NoC Arbitration

A NoC consists of interconnected routers that link together numerous on-chip components (e.g., cores, caches, memory controllers). Typically, a router has multiple input and output ports and within each input port, there could be one or more input buffers and support for virtual channels (VCs). Arbitration is required when messages from multiple input buffers or VCs compete for the same resource such as output ports and output VCs. Arbitration policies are critical to NoC latency, throughput, and fairness. Nevertheless, designing an arbitration policy can be challenging. First, a local decision could have a delayed effect: a seemingly reasonable decision made in one cycle could result in congestion in downstream routers many cycles later. Second, it is difficult to associate overall application performance with any one specific arbitration decision. Third, routes in a NoC overlap and intersect. A single message route involves multiple arbitration decisions. Similarly, a single arbitration decision can affect multiple routes.

Researchers have explored a variety of arbitration policies. A traditional round-robin policy provides a high degree of local fairness by treating each input port equally. Only considering local fairness at each router, however, can lead to poor global *equality* of service [13]. Approximated age-based packet arbitration [14] provides equality of service, but has limitations regarding fairness of bandwidth allocation. Fair Queuing [15] and Virtual Clock [16] improve fairness and network utilization by maintaining per-flow state and queues, but are costly to implement. Link bandwidth allocation becomes increasingly unfair as a message traverses more routers. Global-age arbitration is considered one of the best policies and has been effectively implemented in off-chip routers, but its hardware cost is largely impractical for use in on-chip routers [17].

2.2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique used for decision-making problems. In RL, an *agent* attempts to learn a policy for navigating an environment that leads to maximum long-term reward. During training, the *environment* returns a numerical reward for each action it takes (Fig. 1). The agent then uses this reward to train itself.

RL differs from *supervised* learning in that correctly labeled data are not required. Instead, RL uses a reward to adjust behavior. As we demonstrate in Section 6.3, different reward functions lead to different arbiter policies with varying degrees of effectiveness. RL also differs from *unsupervised* learning in that RL learns a *classifier*, while unsupervised learning typically learns a *classification*. The distinction is subtle but important. The result of RL is an algorithm for making a decision on new inputs. With unsupervised learning, new inputs are either only approximately classified or require a learning update phase.

One well-known branch of RL is value-based learning. For example, with Q-learning [18], a Q-value represents the quality of taking a particular action a when the environment is in state s. In its simplest form, for each state s, there are several possible actions to take. The environment can either choose an action a that has the highest (currently estimated) Q-value among all possible actions or take a random action to explore new trajectories. After taking the action, the environment transitions to a new state s' and provides a reward r. With the tuple $\langle s, a, r, s' \rangle$, the algorithm uses the Bellman Equation as an iterative update to maximize the expected cumulative reward achievable from a given state-action pair:

$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$

where α is a learning rate and γ is a discounting factor. To perform well in the long term, both the immediate reward and future rewards need to be accounted for. γ determines how much weight is given to the future rewards.

Traditional Q-learning uses a Q-function. It returns the action with the highest Q-value given the environment's state. A straightforward approach to implementing this function is to use a table lookup. In RL this table is referred to as the Q-table and it stores the Q-value for each state-action pair. For many real-world problems, however, the state-action space can be extremely large. For example, when using RL to play video games, a Q-value for each image frame and all possible actions must be tracked, requiring an impractical amount of storage space [12]. For the NoC arbitration problem in this paper, a vector of hundreds of numbers is required to represent a state (more details in Section 4), and each element in the vector can have over a dozen values. The Q-table approach is therefore not practical for such a large state space. Deep Q-learning (DQL) uses a neural network to approximate the Q-function [12]. Given a state s, the neural network can output an approximated Q-value for each possible action. We use the DQL technique with a multi-layer perceptron neural network to approximate the Q-function.

3. HIGH-LEVEL OVERVIEW

An important step in architecture design is exploring alternatives and optimizing the preferred approaches. Architects typically use high-level simulation along with a combination of intuition and brute force to explore the design space. ML (and RL in particular) is a potentially powerful tool to ac-

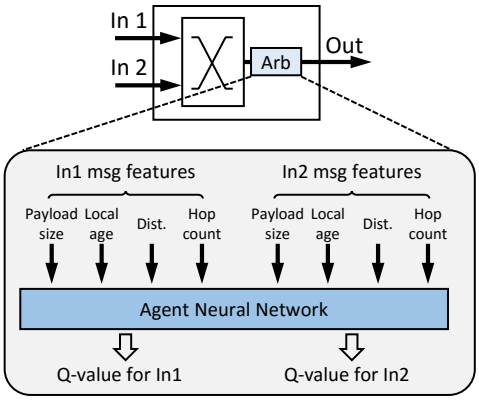


Figure 2: Conceptual diagram of the proposed offline RL model for NoC arbitration.

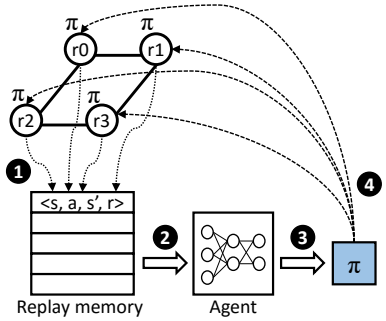


Figure 3: Agent training process.

celerate this process. To use RL, designers replace portions of their simulator with RL algorithms. While typical feature engineering and hyperparameter tuning are still required, little human interaction is needed once the RL algorithm starts exploration. However, human involvement is required after training. The trained models might provide good performance, but the resulting neural network could be infeasible to implement in hardware. In this case, the search is on for humans to understand the behavior of the model and learn *from* the machine. Due to the usual architectural constraints (area, delay, energy, etc.), it might not be feasible to copy the exact behavior of the RL algorithm. Designers must fall back on their domain expertise and intuition to evaluate different practical implementations that are inspired by RL.

Traditional ML algorithms rely on human experts to provide meaningful input features (feature engineering), high quality training data, hyperparameter tuning, and manual construction of the NN architecture. Using ML to discover new features without any human intervention might be possible in the future with substantial advancements in NN interpretability. Given the state-of-the-art for NN interpretability, designers must still manually interpret NN results.

3.1 Application to NoC Arbitration

In this work, we apply RL to NoC arbitration in the following way: every cycle, each router interacts with an agent by sending its own router state(s). The agent then evaluates the router states and computes Q-values for all possible actions. Routers use these Q-values to select input buffers and grant output ports. Meanwhile, a reward is calculated and used by the agent for further training. In this section, we demonstrate the overall concept using a simple setup with synthetic traffic before diving into a detailed exploration and implementation in Section 4.

3.1.1 The RL Framework

Fig. 2 shows how we apply RL to NoC arbitration. We implement a single Agent shared by all routers. Routers query this Agent by sending vectors that represent their current internal state. The Agent uses a neural network to generate Q-values that are sent back to the router where final arbitration decisions are made.

State Vector: Routers in our NoC query the Agent for Q-values to make one arbitration decision per output port. For each arbitration, the router gathers internal state relevant to that output port into a state vector. A state vector consists of a list of features from all messages that compete for the same output port. In Fig. 2, both *In*_1 and *In*_2 have a message requesting *Out*, where each message has four features (see Table 2 for more details on the features) for a total state vector with eight entries denoted by the eight downward-pointing arrows in this example.

Agent: Based on the state vector provided by the router, the Agent computes Q-values for each input port. The Q-value should indicate the expected quality or effectiveness for selecting a message from that port. Every cycle, each output port of each router queries the same agent independently. Note that the same neural-network weights are used to calculate Q-values across all output ports and routers. This is conceptually similar to how a conventional router applies the same arbitration heuristic (e.g., round-robin) to all routers and all ports. However, this is not fundamental; designers can use multiple agents for training, where each agent is trained with only a fixed subset of routers.

Reward: After each arbitration, a reward is generated and sent to the Agent. The reward is a numerical score that should be correlated with how effective an arbitration decision was. The Q-learning algorithm is designed to maximize the long-term (accumulated) reward. In the context of NoC arbitration, several different metrics are possible for defining a reward; examples include message latency, network throughput, and fairness. In this example, we use the global age of the message to compute a reward: we provide a fixed positive reward for selecting the oldest message and zero otherwise. While global age is effective, it is not practical to implement in real hardware NoCs [17], but recall that it *does not have to be*. We use global age to drive our exploration process to identify more practical features that hopefully can provide similar benefits. Section 6.3 evaluates performance sensitivity to different reward functions.

3.1.2 Agent Training

Fig. 3 shows the training process for a small four-router network. After each arbitration decision, the router generates and stores a tuple of ⟨state, action, next state, reward⟩ in a replay memory ❶. The replay memory is a circular buffer used for improving the quality of training. When training the model ❷, instead of using the most recent record, a batch of records is randomly sampled from the replay memory. This

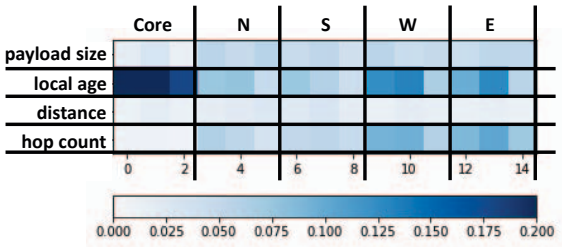


Figure 4: Average weight heatmap of hidden-layer neurons. Each row corresponds to a feature, and each column corresponds to an input buffer.

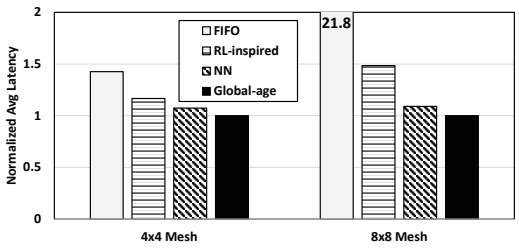


Figure 5: Average message latency comparison for FIFO, our proposed (RL-inspired), agent neural network (NN), and global-age policies. Results are normalized to Global-age.

technique is called experience replay [19]. Depending on the specific application of RL to computer systems design, the exact set of techniques to train the neural network may vary. In this work, experience replay along with the utilization of a second target neural network [12] were effective in stabilizing the training process. Through training, the agent improves its arbitration policy π ❸, which is then used by all routers for making subsequent arbitration decisions ❹.

3.1.3 *Drawing Insights from the Neural Network*

Although the agent might learn a promising policy that leads to significant performance improvement, the hardware and power costs of implementing a NN could be prohibitive for NoCs. Furthermore, performing inference for every arbitration decision at each router would increase the router's critical path and inadvertently degrade system performance. Therefore, rather than building a neural network directly in hardware, designers need to leverage the NN-derived insights to find more practical arbiter designs.

After training stabilizes, we analyze the Agent's neural network to try to understand its behavior. In general, this process can be very challenging, and neural network "interpretability" remains an open research problem in the machine learning community [7]. In this work, because the neural network architecture is shallow (one hidden layer), we were able (with some effort) to directly interpret and derive some insights from the neural network's weights. More discussion of this interpretation is provided in the next section, along with a concrete example.

3.2 Synthetic Traffic Example

This section provides a quick example using a simple 4×4 mesh network; we present a more detailed and realistic exploration in Section 4. Each router has five input ports (N/S/W/E, plus one for a core to inject/eject network messages), where each input port has three virtual channels. For this example, we simply use a uniform random synthetic traffic pattern.

Agent Neural Network Architecture: As is typical of ML, some effort is needed to find an effective neural network organization. We evaluated numerous neural network architectures and performed hyperparameter tuning (number of hidden layers, hidden layer widths, activation functions, learning rate, batch size, discount factor, and exploration rate). The resulting neural network is a multi-layer perceptron with one hidden layer.

Analysis of the Agent Neural Network: To understand the agent neural network's behavior, we used a visualization of the neural network's weights. The agent neural network has 60 input neurons (5 ports×3 buffers/port×4 features/buffer) and one hidden layer with 15 neurons. Fig. 4 shows a heatmap where each pixel is the average of the absolute value of a specific weight across all 15 neurons in the hidden layer. A darker pixel has a higher magnitude of weight than a lighter pixel. Each row in the figure corresponds to a feature, and each column corresponds to an input buffer.

Fig. 4 suggests that the hidden layer neurons tend to make the most use of the local age and hop count features. Payload size also factors into some neurons, and distance is largely ignored. Reflecting upon what the RL process has come up with, local age makes sense in that the longer a request has been waiting at a router, the more likely it is that the request is blocking other traffic or holding on to critical resources (e.g., MSHRs at the originating core). Hop count also makes sense in that requests with the largest hop counts are the ones that have traveled the furthest in the network and therefore are already facing longer latencies. Note that this analysis provides *suggestions* for promising features, but it is still up to the human designer to convert these observations into practical arbitration policies.

Example Arbitration Policy: Based on the observations made above, we demonstrate a simple example arbitration policy. Our proposed policy consists of two steps: compute the priority level for each candidate input buffer and then select the input buffer with the highest priority level. With a 4×4 mesh, we compute priority as follows:

$$Priority\_level = local\_age \ll 1 + hop\_count \gg 1$$

where *local_age* is a 5-bit value and *hop_count* is a 3-bit value associated with each input message. Upon arrival at a router, *local_age* is initialized to zero and then incremented by one on each subsequent cycle, saturating at 31. The *hop_count* feature is set to zero when a message is first injected into the NoC, and then incremented each time the request is forwarded to another router. In contrast with a full neural network that can require thousands of addition and multiplication operations, our example priority computation is simple enough (constant shifts and a single low-bit-width addition) for direct hardware implementation. While this example policy is simple, we emphasize that the current state of the art in ML does not provide an automatic method or process to go from a trained NN to an implementable algorithm such as this; this methodological gap in ML-driven design must still be bridged by human effort.

Results: Fig. 5 (left) compares the performance among four policies: FIFO (prioritizes messages based on their arrival

time to the local router), our proposed arbitration policy ("RL-inspired"), an impractical RL-trained neural-network (NN), and global-age policies for the 4×4 mesh NoC. FIFO policy is simple to implement in hardware unlike global age, but still captures some notion of age (local age). The results show that our proposed policy can significantly reduce message latency compared to the FIFO policy and cover a significant portion of the performance gap between FIFO and the impractical neural-network and global-age policies. Note that these results are specific to the uniform random synthetic traffic, and other traffic patterns may require different policies.[1]

8×8 Mesh Network: We also repeated this exercise for an 8×8 mesh network. In this case, hop count carried more weight than local age due to the longer routes through the larger network. We conjecture that in a larger network, global age can be better approximated through hop count instead of local age. For this network, we compute priority as:

$$Priority\_level = local\_age + hop\_count \ll 2$$

which reflects the larger influence of *hop_count*. Fig. 5 (right) shows the performance of the different arbitration policies for this larger network. The performance of FIFO arbitration is significantly worse, as the difference between a request's local age at a router and its global age since initially entering the NoC can diverge drastically. Both the implementable RL-inspired and NN policies perform relatively close to the global-age policy.

These examples illustrate that an ML-approach can provide useful guidance, but the human architect must still follow through to conceive practical circuits that efficiently implement the essential learned behaviors from the neural network.

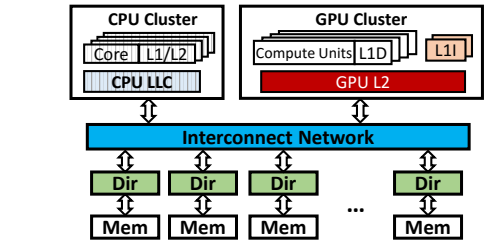
4. IN-DEPTH DESIGN EXPLORATION

In this section, we look at more realistic application traffic and more complex networks that interconnect dozens of CPU and GPU components. We begin by describing the chip architecture and then explain the workloads. After that, we return to RL in an attempt to develop an efficient NoC arbiter for this more complex chip architecture and our application-driven workloads.

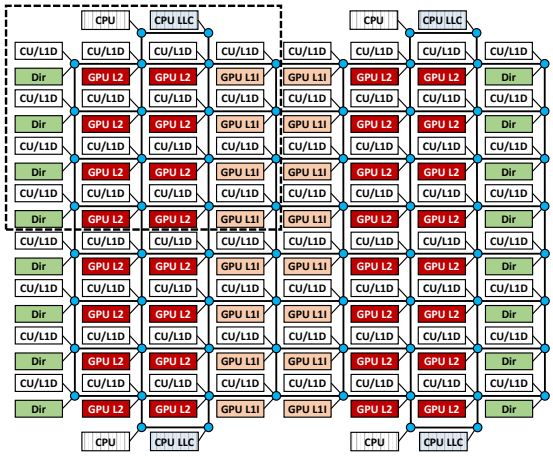
4.1 Baseline Architecture

We use an APU simulation platform consisting of gem5 [20] and a modified version of the AMD GPU model [21] for evaluation. Fig. 6a shows our baseline chip architecture that contains both CPU and GPU clusters. The CPU cluster consists of CPU cores, private L1 and L2 caches, and a last-level-cache (LLC). The GPU cluster consists of compute units (CUs) with private L1 data caches. GPU L1 instruction caches are shared by every four CUs. The GPU uses a banked unified L2 cache. Individual coherence directories are connected to the memory controllers for off-chip memory accesses. These directories are responsible for keeping the CPU LLC and the GPU L2 cache coherent. CPU caches are write-back and are kept coherent through a MOESI directory protocol. GPU caches are write-through and write-no-allocate. GPU L1 caches are kept coherent by writing through

[1] Additional performance could potentially be extracted by further tuning our RL-inspired algorithm (e.g., by prioritizing new requests from the "core" input as suggested by Fig. 4), but we do not explore this here as this current example is merely for illustrative purposes.



(a) Baseline system overview.



(b) Baseline topology.

Figure 6: Baseline system.

dirty data and invalidating the caches at kernel launch.

Fig. 6b shows the baseline NoC topology. The GPU cluster has 64 CUs connected by an 8×8 mesh network. 16 GPU L1 instruction caches are located in the center of the mesh. The 16 coherence directories are placed along the left and right edges, where each directory is also connected to a corresponding memory controller and associated DRAM channel (not pictured). The GPU L2 cache banks are address interleaved. Each quadrant of the mesh is augmented with two additional nodes for a CPU cluster. One node is connected to the CPU including its L1 and L2 caches, and the other is connected to the CPU LLC. This system requires seven network classes for coherence, and therefore, each router has seven virtual channels per input port. Request and coherence messages are 1 flit in length, and data messages are 5 flits (1 header, 4 data). We use Garnet [22] to model the NoC and implement the RL framework inside Garnet.

4.2 Workloads

We use the APU-SynFull methodology in this work to reduce simulation time while still driving our experiments with realistic application-based coherence and memory traffic [23]. APU-SynFull is based on SynFull [24], which analyzes NoC traffic traces from a detailed cycle-level simulation of application execution and creates stochastic Markov-model-based traffic generators. The generated traces are statistically similar to the original applications in terms of program phases, CPU vs. GPU traffic, distributions of message sources and destinations, per-node injection rates, etc. More importantly, APU-SynFull captures memory instruction dependencies of the original program, which allows us to evaluate how arbitration decisions impact total program execution time.

Benchmark Suite	Applications
AMD SDK [25]	dct, histogram, matrixmul, reduction
OpenDwarfs [26]	SpMV
Rodinia [27]	bfs, hotspot
HPC proxy app [28]	CoMD, miniFE

Table 1: List of traffic-intensive workloads.

We evaluate the applications listed in Table 1 from the AMD SDK [25], OpenDwarfs [26], Rodinia [27], and ECP Proxy Apps [28] suites. We focus on traffic-intensive applications on which arbitration matters, but we also evaluated other workloads and verified that nothing we do inadvertently hurts performance in those cases. We first execute each application in the gem5 simulator and collect NoC traffic traces. Then we use APU-SynFull to analyze the trace and generate a model file for each benchmark. Traces are collected from a smaller system with 16 CUs/L1Ds, 4 L1Is, 8 L2s, and 16 coherence directories. We simulate a multi-program scenario by running four independent copies of an application (as codified by its model file) in the baseline 64-CU system, one in each quadrant (marked with the dashed line in Fig. 6b). The GPU L2 cache banks are organized so that they are private to each quadrant, therefore cache coherence traffic does not cross the quadrant boundaries. The memory controllers are shared by all quadrants and therefore CPU LLC and GPU L2 misses can cross quadrant boundaries.

Because each of the four applications are executed independently, each copy may have a different completion time. A quadrant becomes idle when the launched application finishes. We define the **average program execution time** as the average completion time across all four instances of the application, and **tail program execution time** as the completion time of the slowest instance. We also evaluate scenarios consisting of mixes of different applications. To fully train the agent, we execute the same set of model files repeatedly until the training converges.

4.3 Message Features

For this in-depth design, we consider the complete set of message features listed in Table 2. Note that these features are a combination of general NoC features (e.g., hop count) and ones specific to the NoC of a chip multiprocessor (e.g., cache coherence request type). These features should be meaningful for making arbitration decisions. For example, local age and hop count are related to message latency. In-flight messages and inter-arrival time are related to NoC traffic load. Payload size, distance, message type, and destination type are message properties that may correlate to underlying application behaviors and patterns. Note that both message type and destination type are one-hot encoded features. As a result, each message needs 12 elements (one each for the first six features, and then three each for the last two) to represent all of its features.[2] Each feature is normalized such that all values in the state vector are between 0.0 and 1.0 (see Section 6.2 for more discussion on feature engineering). All of this feature selection and engineering represent another point in the process that requires human input.

4.4 Router State Vector

[2]We emphasize that this work does *not* aim to identify new *input* features, but rather which features or higher-order features (i.e., combinations thereof) are the most effective.

Algorithm 1 Router Decision Making Algorithm

```
Q: Q-value vector
P: A set of input requesters that request the same output
r: An input requester (buffer)
p: The input port a requester belongs to
for each output port do
    Q ← get_Q_value()
    R ← get_input_requester()
    if (output port not busy) then
        while (R ≠ ∅) do
            select r = { random requester (with probability ε)
                       { requester with the largest Q-value
            p ← get_input_port(r)
            if (p not granted) then
                Grant output port to input buffer r
                Record granted input port p
                break
            else
                Remove r from R, and set the corresponding Q-value to 0 in Q
            end if
        end while
    end if
end for
```

In Section 3.1.1, we introduced the router state vector and described how it is used by the Agent to compute Q-values. A state vector consists of a list of features from all messages that compete for the same output port. Consider a router with n input ports, m output ports, k input buffers per port (used for multiple message classes, virtual channels, etc.), and p features per message. Each router generates m separate state vectors of length $n \times k \times p$ each, as each of the m output ports will be independently arbitrated.

The number of ports per router can be different, depending on the location of the routers. For example, corner routers have fewer ports than center routers in a 2D mesh. Because all routers share the same agent in our present implementation, the input-layer width of the agent neural network must be equal to the width of the largest router's state vector. To guarantee that all state vectors have the same width, we align the vector based on the input ports, and zero out any empty and/or non-existent ports for the routers with fewer ports.

4.5 Decision Making

We only consider router designs where each message requests a single output port, and the arbitration decision for each output port is independent of the others. Because of this, depending on the input status and output port availability, a router queries the Agent up to m times every cycle, where m is the number of output ports. If an output port is not requested or not available (e.g., in the middle of transmitting a multi-flit message), the router does not issue a query for that particular output. If an output port is requested by only one input message, the output port is directly granted to that input buffer without querying the agent.

As with our simplified network in Section 3, the Agent computes the Q-value score for each potential input port, from which an output port could route a message. We call this collection of Q-values the *Q-value vector*. The Q-value vector is sent back to the router, which ultimately makes an arbitration decision. The router grants the output port to the input buffer that has the highest Q-value, with the following exceptions: (1) The highest Q-value element might correspond to an empty or irrelevant buffer, especially during

Feature Name	Description
Payload size	size of the message (in flits)
Local age	number of network cycles spent from the time at which the message arrived at the current router
Distance	number of hops from the message's source to destination routers
Hop count	number of hops the message has traversed so far
In-flight messages	number of outstanding requests from the message's source router
Inter-arrival time	number of network cycles between the arrival of two consecutive messages at the same buffer
Message type	type of the message: request, response, or coherence
Destination type	type of the destination node: core, cache, or memory

Table 2: Message features considered in this work.

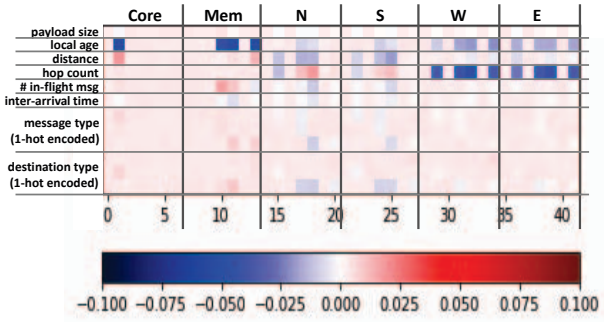


Figure 7: Average weight heatmap of the hidden layer for Bfs's agent neural network. Each row corresponds to a feature, and each column corresponds to an input buffer.

the initial training phase; (2) An input port can route at most one message per cycle; therefore, no more than one output port should be granted to the same input port in the same cycle. In these two cases, the router selects the next eligible input buffer with the highest Q-value; and (3) The Agent must explore its Environment to cover as much decision space as possible. Hence, with a probability ε, the router randomly selects a candidate input buffer. Algorithm 1 presents the pseudocode for the router's arbitration decision making.

4.6 Analysis of the Neural Network

After extensive exploration of neural network architectures and hyperparameter tuning, we found that many models showed similar prediction accuracy. Considering the training time and interpretability, we chose the simplest neural network (one input layer, one hidden layer, and one output layer) among the best-performing models. The largest router in the target NoC has 6 input ports (core, memory, north, south, west, and east), each with 7 buffers, and each message needs 12 numbers to represent all of its features. Therefore, the agent neural network has $6\times7\times12 = 504$ input neurons. Both hidden and output layers have 42 neurons, and their activation functions are Sigmoid and ReLU, respectively. The learning rate, discount factor, and exploration rate ε are 0.001, 0.9, and 0.001, respectively. We train the Agent every cycle using a batch of two records randomly sampled from a 4000-entry replay memory.

We use a similar manual NN visualization process as before. For illustrative purposes, we derive the arbitration policy based on the training of a single application (i.e., Bfs in Fig. 7), but our results show that the derived policy generalizes well to other applications. Similar to before, the local age and hop count features are heavily used by the neural network. However, there is additional variation and sensitivity across message classes and input ports. Certain message classes (indicated by the different columns within an input port in the heatmap) have more significant weights than others, which suggests that properly handling these messages may have a larger impact on the performance of the arbiter. Unlike local age and hop count, which can be used to approximate the global age of a single message, the message class feature tells us among all messages, which types of message are likely to have larger global age. The message classes with more significant weights in our system send GPU coherence, memory response, and GPU L2 response messages, which correspond to the 4th, 5th, and last columns/pixels for each input port. Intuitively, prioritizing coherence and response messages allows the compute units to make better forward progress. As a result, when designing the final arbitration policy, we would likely want to prioritize messages from these classes, but we also note that the degree of prioritization varies by input port. The NN analysis not only identifies important features, but it also illuminates the context for when those features matter. As another example, we also observed that hop count shows positive weights on North and South ports, but negative on West and East ports. We analyzed the weights in the output layer and discovered that they are mostly positive, so the negative weights in the hidden layer indicate that a smaller input value is preferred. As a result, in our final arbitration policy (next subsection), we prioritize messages with larger hop counts for North and South ports, while prioritizing smaller hop counts for West and East ports. These compound, multi-feature prioritizations are effectively higher-order features that the NN has learned. These more complex relationships between input features and port directions were not immediately obvious nor intuitive to us, and this highlights an example where in fact ML-aided design exploration uncovered new behaviors that we could potentially exploit.

4.7 An RL-inspired Arbiter

Algorithm 2 presents an algorithm for computing arbitration priority levels, inspired by our analysis of the neural network learned by the RL agent. We combine local age, hop count, and message class. The overall rationale for our arbitration algorithm follows, but much of this is driven by the heatmap from Fig. 7. First, to avoid starvation and ensure forward progress, we always prioritize the oldest message when there exists any messages with a local age beyond a certain threshold (24 cycles). Next, we give higher priority to coherence and response messages, as draining these out of the NoC as quickly as possible tends to unblock stalled computation in the CPU and GPU cores. We then prioritize messages with larger hop counts for North and South ports

Algorithm 2 Priority level for each input buffer

```
1: LA: 5-bit counter for local age
2: HC: 4-bit counter for hop count
3: if (LA > 11000_2) then
4:     priority_level = LA
5: else
6:     if (message comes from Core/Memory/North/South) then
7:         if (message is coherence or GPU response) then
8:             priority_level = HC ≪ 1
9:         else
10:            priority_level = HC
11:        end if
12:    else
13:        // message comes from West/East
14:        if (messages is coherence or GPU response) then
15:            priority_level = (1111_2 − HC) ≪ 1
16:        else
17:            priority_level = 1111_2 − HC
18:        end if
19:    end if
20: end if
```

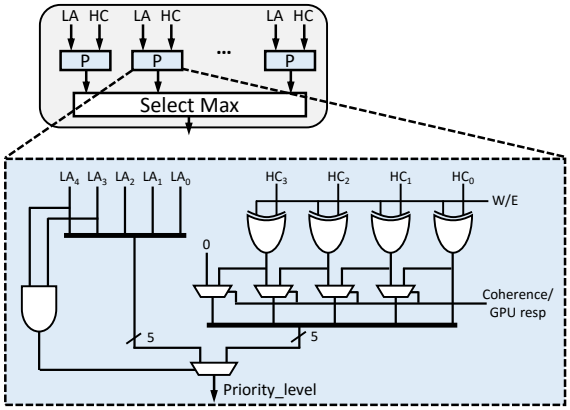


Figure 8: Diagram for the proposed arbiter. W/E denotes that the input port is from the East or West direction.

but smaller hop counts for West and East ports. This is less intuitive, but we suspect that it is related to the underlying X-Y routing of the mesh.

4.8 Hardware Implementation

A few simple hardware modifications are required to support tracking the local age and hop count of each message. First, each input buffer is augmented with a small 5-bit *LA* (local age) counter, which is initialized to zero on message arrival and incremented every cycle thereafter until it saturates at 31. There are multiple ways to track the hop count for each message. If there are a sufficient number of bits available in the header flit, then a 4-bit *HC* field can be carried by the header flit that is incremented upon arrival at each router along its way to its destination. With a sufficiently regular network topology like a mesh with regular routing (e.g., X-Y), the hop count can be directly computed based on the source node ID and the location of the current router. Other techniques like lookup tables can be utilized, and it is likely that approximations can also be used (e.g., probabilistically incrementing the hop count to reduce the number of bits required in the flit header).

The logic for Algorithm 2 must also be implementable in relatively simple circuitry that can fit within a single clock cycle. Fig. 8 shows one possible implementation of our algorithm. Each input buffer computes a priority level using a P-block shown in the top of the figure. A simple Select-Max circuit chooses the input buffer with the highest priority [29]. Despite the 20 lines of pseudocode in the algorithm listing, the P-block can be implemented in a simple circuit, shown in the bottom section of the figure. Careful selection of the local age threshold enables the logic for line 3 to be implemented with a simple AND gate (any 5-bit value 24 or greater will always have its two MSBs equal to one). The subtraction operations on lines 15 and 17 can be implemented by simply inverting the bits of the hop-count counter, and the conditional nature of the inversion (line 6) can all be combined into a single XOR gate. The final selections can be implemented with simple muxes. This overall process of generating the algorithm in Algorithm 2, selecting "implementation-convenient" constants such as 24 in the age threshold, and ultimately distilling everything down to logic gates was an entirely human-driven effort. While ML provided critical clues, the human architect was still responsible for solving the final puzzle.

	Agent NN	**Round-robin**	**Proposed Arbiter**
Latency (ns)	8.17	0.89	1.10 (0.18 + 0.92)
Area (mm^2)	1.2344	0.0012	0.0044
Power (mw)	63.67	0.07	0.27

Table 3: Synthesis results.

We evaluate hardware cost with Synopsys Design Compiler at a 32nm technology node. Table 3 shows the synthesis results for the Agent inference NN (quantizing to INT8), round-robin arbiter, and the proposed arbiter in a 6-port router. Even though we largely parallelized the inference NN (at the cost of larger area and power overhead), it cannot complete the inference within one cycle for a NoC at 1GHz. For our proposed arbiter, the priority level computation and select-max circuit incur 0.18ns and 0.92ns latency, respectively. Note that priority level computation can be performed in parallel with route computation or VC allocation, and hence does not affect the router's critical path. The proposed arbiter can easily satisfy the timing constraint of a 2-stage router with minor optimization.

4.9 Summary

This section presented one viable way to derive an arbitration policy for more complex systems using RL. The process heavily relies on interpretability of neural networks and domain knowledge. Smaller neural network architectures are easier to understand and more tractable for directly analyzing the weights to extract insights. In addition, domain expertise is required to reason about design choices and trade-offs.

After training the RL agent, the following human-driven steps were followed to arrive at the final (implementable) policy. 1) Analyze the agent neural network's weights and select features that show significant positive and negative weights. 2) Compare features' weights, apply domain knowledge, and reason about their relative importance. 3) Derive algorithms based on the features, their inter-relationships, and their relative importance. This step might result in multiple possible algorithms and evaluations might be required. 4) Derive implementable combinational logic based on the algorithm. Again, domain expertise is required to make design trade-offs

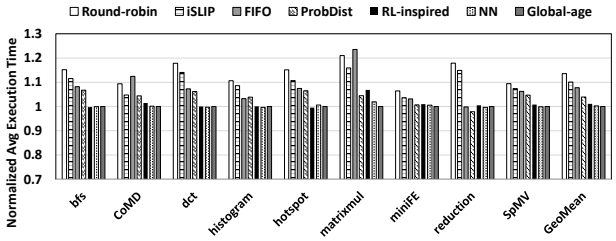


Figure 9: Average execution time comparison. Results are normalized to global-age arbitration.

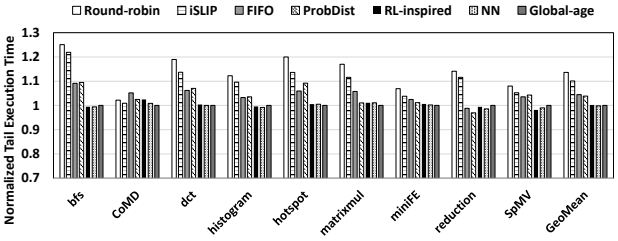


Figure 10: Program tail execution time comparison. Results are normalized to global-age arbitration.

and comply with design constraints. 5) Repeat steps 3 and 4 as necessary. In our experience, ML is useful for discovering patterns and behaviors that we might have otherwise missed, but it is still up to the humans to distill insights from NNs.

5. RESULTS

5.1 Average Execution Time

Fig. 9 compares the average program execution time among seven arbitration policies, in which iSLIP [30] and ProbDist [14] are policies proposed in prior work. On average, our proposed RL-inspired policy reduces average execution time by 12.5%, 9.0%, 6.7%, and 2.9% compared to Round-robin, iSLIP, FIFO, and ProbDist policies, respectively. The results also show the performance for two impractical/ideal arbitration policies. The first (NN) uses the complete neural network learned from our RL process, which would take too much area and be too slow to use in practice. The second (Global-age) uses the global age of the messages, which is typically not available due to the complexity of maintaining global timestamps throughout the entire system. Despite being limited to only readily-available message features, our RL-inspired arbitration algorithm achieves performance on par with these two other impractical approaches.

In a few cases, our proposed policy even outperforms the Global-age policy. One contributing factor is that we explicitly prioritize coherence and response messages over request messages, whereas Global-age arbitration treats all messages equally, which could slow down program execution by prioritizing requests over responses, especially when the network is congested. Also, while Global-age has been shown to be effective, it is not a provably optimal policy.

Our proposed policy uses port information (E-W/N-S asymmetry) and message type information (prioritizing response messages). To understand the importance of these features, we de-featured Algorithm 2 by removing port (Line 6) and message type (Lines 7, 14) conditions one at a time which effectively reduces the richer higher-order features back down to the basic standalone input feature (e.g., hop count alone). Compared to our proposed policy, ignoring port information increases average program execution time by up to 6.5% (2.2% on average); and ignoring message type information increases average program execution time by up to 5.1% (1.2% on average). This helps to illustrate the value of using the NNs to mine and uncover such higher-order features.

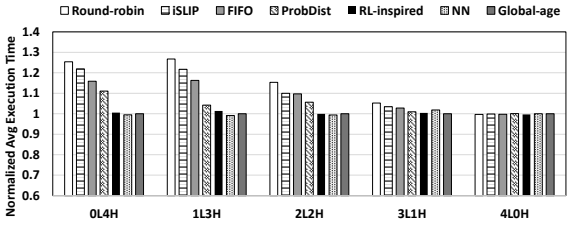


Figure 11: Average program execution time comparison of mixed workloads. **L** stands for low-injection, **H** stands for high-injection workloads.

5.2 Tail Execution Time

Our workload scenario executes four copies of the same GPU workload in each quadrant of the overall chip. While the previous section's results showed performance improvements on average across the four copies of the workload, this may not be sufficient if some copies suffer slowdowns. Fig. 10 shows the results for the tail execution latency of the workloads. Similar to the average latency, the RL-inspired approach outperforms Round-robin, iSLIP, FIFO, and ProbDist arbiters, and matches the performance of the impractical NN and Global-age policies. The round-robin policy shows variations in program execution time across the four copies of the workload. Our RL-inspired policy (as well as NN and Global-age policies) has much more balanced execution time distributions, resulting in 13.4%, 9.8%, 4.3%, and 3.6% improvements in tail execution times compared to Round-robin, iSLIP, FIFO, and ProbDist policies, respectively. In addition to reducing program tail execution time, we also observe that our proposed policy reduces the tail latency of network messages, and therefore provides better fairness compared to Round-robin and FIFO policies.

5.3 Mixed-application Workloads

In Fig. 11, we evaluate the proposed arbitration policy in a mixed workload environment, such as one might find in public Cloud scenarios. In particular, we run four different applications simultaneously in the CPU/GPU system, one in each quadrant. We divide the applications into a high-injection (> 0.05 flit/cycle/node) group and a low-injection group, and classify the experiments into five categories. When the NoC is reasonably congested (2L2H, 1L3H, and 0L4H), our RL-inspired arbitration policy performs favorably compared to the ideal global-age policy. Not surprisingly, when the overall NoC is under-utilized (4L0H), the choice of arbitration policy hardly impacts performance.

6. LEARNINGS AND DISCUSSION

In this section, we share some observations and learnings from our experiences with using RL to drive NoC design.

6.1 Limitations of ML Approach

While applying ML in microarchitecture design processes is a promising approach, it has limitations, and some gaps in the process must be filled. Our case study highlights the following limitations and gaps. First, distilling information from the ML model largely relies on a NN's interpretability and the architect's domain expertise. Filtering out less important features is an unavoidable step toward concise circuit implementation. This process might involve additional experiments to determine feature importance. Secondly, human involvement is necessary to convert an ML model into the final implementation. Especially under hardware and power constraints, design trade-offs must be made between performance and hardware/power cost. This might result in a final implementation that consists of only a portion of the ML model's functionality. Last but not least, critical functionalities (e.g., starvation avoidance) might not be explicit from the model; therefore, certain design requirements may need to be addressed separately.

6.2 Feature Encoding

The input features for our RL training have different ranges of values. Some features only have values from a bounded range. For example in our work, the message distance is bounded by the topology and routing of the underlying NoC. On the other hand, a feature like local age can potentially take on unbounded values, especially if the RL algorithm has not yet been fully trained (see also Section 6.4 on starvation issues in training). We found that it was necessary to normalize the inputs (Section 4.3) to deal with this wide range of possible values. For example, without normalization, a long-delayed message could take on an arbitrarily large value, which could then dominate the weighted sum of a neuron's computation. The large values also led to correspondingly larger gradients, which we found could destabilize RL training.

The other type of features that required different consideration are categorical features that encode a class or attribute where the enumerative value does not correlate to any physical property. For example, messages are requests, responses, or coherence, which could naturally be encoded as 0, 1, and 2, respectively. However, multiplying such a value by a neural network weight would generally only provide stable results if the importance or priority of the message types followed in the same order (i.e., coherence > response > request), but the relative importance/ordering may differ across applications. One-hot encoding enables the neural network to independently learn the importance of each message class. The process of engineering effective features to feed into the RL agent was a human-driven task outside of the automated aspects of ML.

6.3 Reward Function

Rewards are an integral part of RL; training with the wrong reward can cause the neural network to improperly optimize the overall system (and also impact training stability and convergence rates). Thus far, we have only used global age as the reward function. However, there are many other possible metrics, such as quality of service (QoS) and power consumption. Selection of an effective reward function is yet another human-driven component of the overall ML-aided

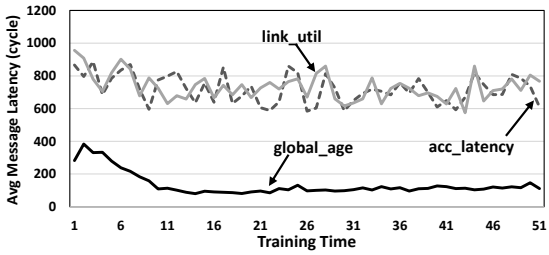


Figure 12: Training result comparison for the agent neural networks using different reward functions.

methodology. Below, we consider two additional reward functions:

- **Reciprocal of average accumulated latency:** Periodically (e.g., 10 cycles), we calculate the average latency L of messages that arrived at their destinations and those still in-transit. Because lower latency generally indicates better performance, we define the reward function as the reciprocal of average message latency (i.e., $\frac{1}{L}$); the lower the latency, the higher the reward. Because L is computed once per period, it is then used as a fixed reward for all actions in the following period. We found that considering only completed messages in the reward function can lead to starvation and livelock. The agent learns to optimize for the latency too aggressively by prioritizing newer messages and delaying older messages indefinitely. Including in-transit messages in the reward helps the agent to learn how to balance between newer and older messages.
- **Link utilization in the previous cycle:** Link utilization is calculated as the number of links that transferred at least one message in the previous cycle divided by the total number of network links. This reward can help the agent to maximize link utilization, which in turn generally improves network throughput. Similar to the above reward, this reward is used for all actions in the next cycle regardless of the actual actions taken.

Fig. 12 compares the three reward functions (global-age, reciprocal of average accumulated latency, and NoC link utilization). The results show that only global_age effectively reduces average message latency and converges to a steady state. The other reward functions, acc_latency and link_util, hardly converge. The key advantage that global_age has over acc_latency and link_util is that the reward is directly tied to the specific arbitration decision made and is given immediately. In contrast, both acc_latency and link_util reflect global behaviors, which are harder to assess the individual actions taken by the agent. While global objectives should conceptually be better (e.g., optimize for overall program execution time rather than other more loosely correlated metrics), we discovered that the agent learns more effectively when given more direct and immediate feedback.

6.4 Starvation Avoidance

Our RL methodology for NoCs did not explicitly address starvation. As discussed in Section 6.3, if the reward function is not defined properly, the learned strategy can potentially lead to starvation or livelock. Including starvation/livelock

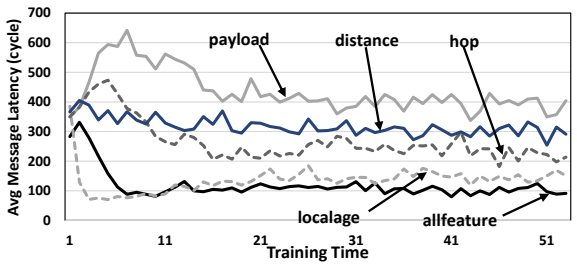


Figure 13: Comparison of agent training results over time with different neural-network input features.

avoidance explicitly (integrated in agent decision making algorithm) or implicitly (through reward function) as part of the training is possible, and the agent might learn to avoid starvation/livelock. However, designers may still need to distill the learned behavior and address starvation separately when designing the final arbitration policies to implement in hardware. While lines 3-4 of Algorithm 2 ended up making use of message local age in a way that prevents starvation, during our initial attempts at designing implementable policies, we did run into failed approaches that caused livelocks (for example by putting the consideration of local age inside the other clauses). Domain expertise is still required to distill the insights from the neural network into a design that works in practice.

6.5 Alternative Neural Network Analysis

As an alternative to directly visualizing the neural network weights, we also experimented with selectively enabling features to the RL framework in a hill-climbing fashion. We started by individually training the neural network with only one feature at a time. Fig. 13 shows the performance for a few individual features, plus "allfeatures" for reference, which shows the performance when all features are considered at once. Not surprisingly, local age was the feature that performed the best. We then retrained the neural network utilizing all pairs of features combining local age with one other feature (not shown in the figure), which resulted in local age and hop count. We repeated this once more with these two features plus one additional feature, but this did not result in any further performance improvements. While hill-climbing identified the same features as our detailed analysis of the NN heatmap (the consistency in feature sets provides additional confidence in our methodology), this does not imply that hill-climbing alone is sufficient. In particular, knowing the set of features still does not tell the architect how the features should be combined or in what contexts they should be utilized or deemphasized.

The message for this section is that there may be multiple options for how a computer architect can extract insights from the RL analysis, and that designers need to utilize their domain expertise to choose an approach that suits their neural network architecture and overall architectural application.

7. RELATED WORK

In additional to the work outlined in Section 2.1, iSLIP [30] is a round-robin-based policy that performs multiple iterations to find a conflict-free input-to-output mapping. Ping-pong arbitration [31] is another round-robin-based policy that divides the inputs into groups and applies arbitration recursively to provide fair sharing of switch bandwidth among inputs. Das *et al.* [32] propose a slack-aware arbitration policy that utilizes memory access criticality information for packet scheduling within the NoC. Cai *et al.* [33] propose a NoC arbitration policy that considers both traffic type and packet slack for heterogeneous systems. Wavefront allocation [34] tries to maximally match input to output requests but suffers from long latency as the number of requesters or resources increases. Packet chaining [35] improves switch matching by reusing switch allocation for short packets across multiple cycles, allowing an efficient matching to be built incrementally. Packet chaining observes that short packets associated with many coherence messages should be handled differently in the router. We also observe that the type of message has an impact on the arbitration decision but do not consider consecutive messages when making arbiter decisions.

Machine learning in microarchitecture is not new, and there is a large body of prior work that utilizes machine learning techniques to improve architectural designs. A thorough survey can be found in [36]. The perceptron branch predictor [1] uses a linear classifier that is trained online to predict whether a branch is taken or not. More recently, Garza *et al.* [37] propose a perceptron-based predictor for indirect branch prediction. Ipek *et al.* [2] propose a reinforcement-learning-based memory controller that interacts with the system to optimize performance. Teran *et al.* [3] propose perceptron learning for reuse prediction, which uses tags and program counters to learn correlations between past cache access patterns and future accesses. Peled *et al.* [38] introduce semantic locality to capture the relationship between data elements in a program and propose a memory prefetcher to approximate semantic locality using reinforcement learning. Zeng *et al.* [4] propose a long short-term memory-based memory prefetcher that learns to capture regular memory access patterns. Hashemi *et al.* [39] relate contemporary prefetching strategies to n-gram models in natural language processing and propose a recurrent-neural-network-based prefetcher that handles irregular benchmarks. Bhatia *et al.* [40] propose a perceptron-based prefetch filtering mechanism that increases the coverage of the prefetches without negatively impacting accuracy.

In the NoC, machine learning has been used for routing, fault tolerance, DVFS control, and router optimization. Ebrahimi *et al.* [41] apply reinforcement learning to make adaptive routing decisions. DiTomaso *et al.* [42] utilize ML techniques to train a decision tree to predict faults in an NoC. Decision trees and offline-trained regression models have been used to dynamically control the power consumption in traditional electrical NoCs [43, 44] and photonic NoCs [45]. Fettes *et al.* [5] propose learning-enabled energy-aware DVFS for multicore architectures using both supervised learning and reinforcement learning approaches. Zheng *et al.* [6] use RL to learn an DVFS policy and propose an artificial neural network to efficiently implement the large state-action table required by RL. Yin *et al.* [46] propose using RL in NoC arbitration. Lin *et al.* [47] use DRL to optimize routerless NoC designs for better throughput and latency.

8. CONCLUSION

Technology has brought disruptive changes and machines are becoming more powerful. However, in many domains including computer architecture, human expertise is still crucial. In this paper, we leverage machine learning (reinforcement learning in particular) to design NoC arbiters. We show that machine learning can be a valuable tool to augment human creativity and intuition in the exploratory design phases, but we still relied on humans (ourselves) to distill the underlying neural network into simple and implementable circuits. We highlight the limitations and gaps in the current process of ML-aided architecture designs. While NoC arbitration is but one component of modern computing systems, we hope that this work can provide guidance in how to fill the gaps and apply ML techniques to design better systems.

Acknowledgment

AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

9. REFERENCES

[1] D. A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *HPCA 2001*.

[2] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach," in *ISCA 2008*.

[3] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron Learning for Reuse Prediction," in *MICRO 2016*.

[4] Y. Zeng and X. Guo, "Long Short Term Memory Based Hardware Prefetcher: A Case Study," in *MEMSYS 2017*.

[5] Q. Fettes *et al.*, "Dynamic voltage and frequency scaling in nocs with supervised and reinforcement learning techniques," *IEEE Transactions on Computers*, vol. 68, no. 3, March 2019.

[6] H. Zheng and A. Louri, "An energy-efficient network-on-chip design using reinforcement learning," in *DAC 2019*.

[7] Q. Zhang, Y. Nian Wu, and S.-C. Zhu, "Interpretable convolutional neural networks," in *CVPR 2018*.

[8] P. M. André Seznec, "A case for (partially)-tagged geometric history length predictors," *JILP 2006*.

[9] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *DAC 2001*.

[10] N. Enright Jerger, T. Krishna, and L.-S. Peh, *On-Chip Networks, 2nd Edition*. Morgan and Claypool Publishers, 2017.

[11] R. Sutton and A. Barto, *Reinforcement Learning*. MIT Press, 1998.

[12] V. Mnih *et al.*, "Human-level Control through Deep Reinforcement Learning," *Nature, 518(7540):529–533*, Feb 2015.

[13] M. Poremba *et al.*, "There and Back Again: Optimizing the Interconnect in Networks of Memory Cubes," in *ISCA 2017*.

[14] M. M. Lee *et al.*, "Probabilistic Distance-Based Arbitration: Providing Equality of Service for Many-Core CMPs," in *MICRO 2010*.

[15] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *SIGCOMM 1989*.

[16] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," in *SIGCOMM 1990*.

[17] D. Abts and D. Weisser, "Age-based Packet Arbitration in Large-radix K-ary N-cubes," in *SC 2007*.

[18] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992.

[19] L.-J. Lin, "Self-improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching," *Machine Learning*, vol. 8, no. 3, pp. 293–321, May 1992.

[20] N. Binkert *et al.*, "The gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[21] A. Gutierrez *et al.*, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *HPCA 2018*.

[22] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator," in *ISPASS 2009*.

[23] J. Yin *et al.*, "Efficient Synthetic Traffic Models for Large, Complex SoCs," in *HPCA 2016*.

[24] M. Badr and N. Enright Jerger, "SynFull: Synthetic Traffic Models Capturing Cache Coherent Behaviour," in *ISCA 2014*.

[25] AMD Inc., "AMD SDK," http://developer.amd.com/tools-and-sdks.

[26] K. Krommydas *et al.*, "On the characterization of OpenCL dwarfs on fixed and reconfigurable platforms," in *ASAP 2014*.

[27] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC 2009*.

[28] "ECP Proxy Application," https://proxyapps.exascaleproject.org/app/.

[29] B. Yuce *et al.*, "A Fast Circuit Topology for Finding the Maximum of N k-bit Numbers," in *ARITH 2013*.

[30] N. McKeown, "The iSLIP Scheduling Algorithm for Input-queued Switches," *IEEE/ACM Trans. Netw.*, vol. 7, no. 2, April 1999.

[31] H. J. Chao, C. H. Lam, and X. Guo, "A Fast Arbitration Scheme for Terabit Packet Switches," in *GLOBECOM 1999*.

[32] R. Das, O. Mutlu, T. Moscibroda, and C. Das, "Aergia: Exploiting Packet Latency Slack in On-Chip Networks," in *ISCA 2010*.

[33] X. Cai, J. Yin, and P. Zhou, "An orchestrated noc prioritization mechanism for heterogeneous cpu-gpu systems," *Integration*, vol. 65, March 2019.

[34] J. Howard *et al.*, "A 48-core IA-32 processor in 45 nm CMOS using on-die message-passing and DVFS for performance and power scaling," *JSSC 2011*, vol. 46, no. 1.

[35] G. Michelogiannakis *et al.*, "Packet chaining: Efficient single-cycle allocation for on-chip networks," in *MICRO 2011*.

[36] D. D. Penney and L. Chen, "A survey of machine learning applied to computer architecture design," *arXiv:1909.12373 [cs.AR]*.

[37] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez, "Bit-level perceptron prediction for indirect branches," in *ISCA 2019*.

[38] L. Peled *et al.*, "Semantic Locality and Context-based Prefetching Using Reinforcement Learning," in *ISCA 2015*.

[39] M. Hashemi *et al.*, "Learning Memory Access Patterns," *arXiv:1803.02329 [cs.LG]*.

[40] E. Bhatia *et al.*, "Perceptron-based prefetch filtering," in *ISCA 2019*.

[41] M. Ebrahimi *et al.*, "Haraq: Congestion-aware learning model for highly adaptive routing algorithm in on-chip networks," in *NOCS 2012*.

[42] D. DiTomaso *et al.*, "Dynamic error mitigation in nocs using intelligent prediction techniques," in *MICRO 2016*.

[43] M. Clark *et al.*, "LEAD: Learning-enabled energy-aware dynamic voltage/frequency scaling in NoCs," in *DAC 2018*.

[44] D. DiTomaso *et al.*, "Machine learning enabled power-aware network-on-chip design," in *DATE 2017*.

[45] S. Van Winkle *et al.*, "Extending the power-efficiency and performance of photonic interconnects for heterogeneous multicores with machine learning," in *HPCA 2018*.

[46] J. Yin *et al.*, "Toward more efficient noc arbitration: A deep reinforcement learning approach," *AIDArch 2018*.

[47] T.-R. Lin *et al.*, "Optimizing routerless network-on-chip designs: An innovative learning-based framework," *arXiv:1905.04423 [cs.AR]*.