AN ABSTRACT OF THE THESIS OF

Drew Penney for the degree of Master of Science in Electrical and Computer Engineering presented on December 9, 2020.

Title: Machine Learning for Architectural Design Space Exploration and Resource Control

Abstract approved: _____

Lizhong Chen

Machine learning has enabled significant advancements in diverse fields, yet, with a few exceptions, has had limited impact on computer architecture. Recent work, however, has begun to explore broader application to design, optimization, and simulation. Notably, machine-learning-based strategies often surpass prior state-of-the-art analytical, heuristic, and human-expert approaches. This thesis first reviews existing work applying machine learning to architecture, ranging from simulation and run-time optimization, to individual component design involving the memory system, branch predictors, networks-on-chip, and GPUs. Next, the thesis presents a novel deep-reinforcement-learning framework for design space exploration. Finally, the thesis introduces an innovative strategy for resource optimization with multiple co-scheduled workloads. Taken together, these works present a promising future for machine-learning-based architectural design.

Machine Learning for Architectural Design Space Exploration and Resource Control

by

Drew Penney

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 9, 2020
Commencement June 2021

Master of Science thesis of Drew Penney presented on December 9, 2020.

APPROVED:

Major Professor, representing Electrical and Computer Engineering

---

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Drew Penney, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Lizhong Chen, for his pivotal role throughout my studies. His wisdom as a professor has been an inspiration and shaped my own career in computer architecture. His ambitious goals for our research have pushed me to achieve far more than I ever thought possible. I am truly fortunate to have him as my advisor.

I am deeply grateful to my father, Bruce, who shared his passion for science, math, and engineering in all the challenging projects we have pursued together. His unparalleled insight, albeit as an analog engineer, has been my foundation when solving practically any problem – "impossible" only makes the task more worthwhile. These lessons will serve me well in life.

Lots of love to my mother, Shirley, and my brother, Sean, for always being there. Their encouragement and support have kept me going and their laughter puts it all in perspective.

The work presented in Chapters 3 and 4 was supported by the National Science Foundation (NSF) grant #1750047.

The work presented in Chapter 5 was supported by Intel Corporation's Academic Research funding. I gratefully acknowledge both the financial support of Intel Corporation as well as the opportunity to collaborate with Bin Li and her colleagues at Intel.

CONTRIBUTION OF AUTHORS

Ting-Ru Lin developed the reinforcement learning framework in Chapter 4 and Massoud Pedram served as his advisor. Bin Li helped conceptualize the predictive resource controller in Chapter 5 and assisted in configuring the evaluation platform.

TABLE OF CONTENTS

Page

1 General Introduction 1

1.1 The Scope of Machine Learning in Architecture 2

1.2 Generalized Design Frameworks 2

1.3 Proactive Resource Management 3

2 General Background 5

2.1 Supervised learning 5

2.2 Unsupervised learning 6

2.3 Semi-supervised learning 7

2.4 Reinforcement Learning 7

2.4.1 General Formulation 7

2.4.2 Deep Reinforcement Learning 8

3 A Survey of Machine Learning Applied to Computer Architecture Design 10

3.1 Literature Review 11

3.1.1 System Simulation 11

3.1.2 GPUs 13

3.1.3 Memory Systems and Branch Prediction 14

3.1.4 Networks-on-Chip 17

3.1.5 System-level Optimization 20

3.1.6 ML-Enabled Approximate Computing 23

3.2 Analysis of Current Practice 24

3.2.1 Online ML Application 25

3.2.2 Offline ML Applications 27

3.2.3 Domain Knowledge & Model Interpretation 29

3.3 Future Work 30

3.3.1 Investigating Models & Algorithms 30

3.3.2 Enhancing Implementation Strategies 31

3.3.3 Developing Generalized Tools 32

3.3.4 Embracing Novel Applications 33

3.4 Conclusion 36

TABLE OF CONTENTS (Continued)

Page

4 A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study 38

4.1 Introduction . . . 38

4.2 Background . . . 40
4.2.1 NoC Architecture . . . 40
4.2.2 Design Space Complexity . . . 43

4.3 Motivation . . . 43
4.3.1 Design Space Exploration . . . 43
4.3.2 Reinforcement Learning Challenges . . . 44

4.4 Proposed Scheme . . . 45
4.4.1 Overview . . . 45
4.4.2 Routerless NoCs Representation . . . 46
4.4.3 Returns After Loop Addition . . . 47
4.4.4 Deep Neural Network . . . 48
4.4.5 Routerless NoC Design Exploration . . . 51
4.4.6 Multi-threaded Learning . . . 54

4.5 Methodology . . . 54

4.6 Results & Analysis . . . 56
4.6.1 Design Space Exploration . . . 56
4.6.2 Framework Capabilities . . . 57
4.6.3 Synthetic Workloads . . . 58
4.6.4 PARSEC Workloads . . . 60
4.6.5 Power . . . 62
4.6.6 Area . . . 63
4.6.7 Discussion . . . 64
4.6.8 Broad Applicability . . . 66

4.7 Conclusion . . . 66

5 Intelligent Resource Optimization for Edge Networks Using Machine Learning 69

5.1 Introduction . . . 69

5.2 Practical Co-scheduling Opportunities . . . 70
5.2.1 Resource Management Options . . . 70
5.2.2 Workload Characterization . . . 71

5.3 Proactive Control Framework . . . 76

TABLE OF CONTENTS (Continued)

Page

5.3.1 Overview 76
5.3.2 QoS Prediction 76
5.3.3 Dynamic Resource Allocation Controller 79
5.4 Initial Results 81
5.4.1 Setup 81
5.4.2 Evaluation 82
5.5 Conclusion 83

6 General Conclusion 84

Bibliography 84

LIST OF FIGURES

Figure Page

3.1 Publications on machine learning applied to architecture (for works examined in Section 3.1) . . . 11

4.1 NoC Architecture. (a) Single-Ring (b) Mesh (c) Hierarchical Ring 42

4.2 A 4x4 NoC with rings. (a) A NoC with one isolated node. (b) A NoC without isolated nodes. (c) A 4x4 routerless NoC with rings. 42

4.3 Deep reinforcement learning framework. 46

4.4 Hop count matrix of a 2x2 routerless NoC. 47

4.5 Deep residual networks. (a) A generic building block for residual networks. (b) A building block for convolutional residual networks. (c) Proposed network. 49

4.6 Monte Carlo tree search. (a) Search. (b) Expansion+evaluation using DNN. (c) Backup. 52

4.7 Multi-threaded framework. 54

4.8 A 4x4 NoC topology generated by DRL. 57

4.9 Average packet latency for synthetic workloads in 10x10 NoC. 59

4.10 Packet latency for PARSEC workloads. 59

4.11 Average hop count for PARSEC workloads. 59

4.12 Power-performance tradeoffs for 8x8 (labels on the data points are node overlapping caps). 62

4.13 Power consumption for PARSEC workloads. 62

4.14 Area comparison (after P&R). 63

4.15 Synthetic Scaling for NoC Configurations. 65

5.1 Isolated vBNG performance (packet drop rate) at 150 Gbps injection rate. 72

5.2 Isolated vBNG performance (packet drop rate) at 113 Gbps injection rate. 72

LIST OF FIGURES (Continued)

Figure Page

5.3 Isolated vBNG performance (packet drop rate) at 75 Gbps injection rate. 72

5.4 Isolated bwaves (BE) execution time. 72

5.5 Isolated omnetpp (BE) execution time. 73

5.6 Isolated cactusADM (BE) execution time. 73

5.7 Co-scheduled behavior of vBNG and jbb2005. 74

5.8 Co-scheduled behavior of vBNG and Resnet50. 75

5.9 Resource control framework. 76

5.10 Branching dueling Q-network (BDQ) model architecture. 79

5.11 Resource allocation comparison. 82

LIST OF TABLES

Table Page

4.1 Hyperparameter Exploration 56

4.2 DRL supports larger NoCs with 18 overlapping. 58

4.3 DRL utilizes additional wiring resources; 8x8. 58

4.4 DRL utilizes additional wiring resources; 10x10. 58

4.5 8x8 PARSEC workload execution time (ms) 61

5.1 Selected Features 78

5.2 Platform & Workload Configuration 81

LIST OF ALGORITHMS

Algorithm Page

1 Greedy Search 53

PREFACE

This thesis comprises three closely-related manuscripts on machine learning applied to computer architecture design. These manuscripts are preceded by a general introduction (Chapter 1) and background (Chapter 2) that unify the three works.

"A Survey of Machine Learning Applied to Computer Architecture Design" (thesis Chapter 3) was published on arXiv and co-authored with Dr. Chen. This paper reviews existing works that have applied machine learning to architecture design tasks, analyzes the techniques used in these works, and highlights some topics for future research. This material was also presented as an invited talk at the 2nd International Workshop on AI-assisted Design for Architecture (AIDArc). Since the original arXiv publication, this material has been expanded to serve as a basis for the book, *AI for Computer Architecture: Principles, Practice, and Prospects*, published by Morgan & Claypool in 2020.

"A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study" (thesis Chapter 4) was presented at the 26th IEEE International Symposium on High Performance Computer Architecture (2020) and was co-authored with Dr. Chen along with Ting-Ru Lin and Dr. Massoud Pedram, both from the University of Southern California. This paper presents a novel deep reinforcement learning framework for design space exploration and a detailed case study on network-on-chip (NoC) design.

"Intelligent Resource Optimization for Edge Networks Using Machine Learning" (thesis Chapter 5) presents preliminary work on a promising application for machine-learning-based resource management. This is an ongoing collaborative research project with Dr. Chen and with Dr. Bin Li at Intel. We plan to expand upon this work and submit the paper for conference publication in the near future.

Common background material, which now appears in Chapter 2, has been omitted from the manuscript chapters.

Chapter 1: General Introduction

Advances in computer architecture design and computational capabilities have historically been closely associated with improvements in semiconductor processing capabilities, particularly those defined by Moore's law and Dennard scaling. Moore's law posited that the components integrated on a single silicon chip would double every two years. Similarly, Dennard scaling suggested that power density of transistors would remain constant as transistor size, voltage, and current decreased together over time. In the past, these two laws have all but guaranteed advances in architectural designs and greater computational capabilities. More recently, however, we have witnessed these technology-based advances break down, thus mandating dramatic shifts in computer architecture design practices.

With the definitive end of Dennard scaling in the 2000s, improvements in clock frequency have slowed dramatically. In response, computer architecture design has shifted into the current multi-core era, which places heavy emphasis on parallel resource scaling and parallel workload execution. Naturally, these increasingly parallel designs have introduced a variety of complex design considerations beyond those in traditional single-core architectures. In particular, networks-on-chip (NoCs) have emerged as a critical factor in multi-core design due to the need for high-performance, yet efficient communication between cores and other resources, such as remote caches, memory controllers, and accelerators. The multi-application aspect of this paradigm has also prompted complex control strategies to maximize machine utilization while maintaining performance guarantees, especially in latency-critical environments. Finally, with the gradual slowing of Moore's law in recent years, the demand for continued advancements in computational capabilities has placed increasing burden on architects to supplant Moore's law with architectural innovation. These growing demands have, in turn, motivated a new paradigm in computer architecture design.

Concurrently, machine learning has transformed from a largely theoretical novelty into a revolutionary factor that has enabled significant advances in many challenges tasks. In particular, machine learning has facilitated breakthroughs in long-standing

tasks such as image classification, where individual objects can now be recognized with near-perfect accuracy [1]. Notably, the principles behind these breakthroughs are broadly applicable. These machine learning models can leverage a generic framework in which important information is derived from the data itself, rather than from painstakingly crafted (and often extremely task-specific) routines developed by human experts. Consequently, machine learning models can be applied to diverse tasks, including those that are too difficult to represent using traditional programming methods or too complex for humans to fully understand. These sophisticated, yet broadly applicable, learning capabilities have therefore become a promising alternative for computer architecture design.

1.1 The Scope of Machine Learning in Architecture

Notable early works including the perceptron-based branch predictor [2] and reinforcement-learning-based memory controllers [3] demonstrated the potential for AI-based methods to surpass prior design techniques, even in well-studied components. Since then, machine learning applications in architecture have quickly expanded to provide state-of-the-art results in practically all major components.

The first manuscript in this thesis is the paper, "A Survey of Machine Learning Applied to Computer Architecture Design," which can be found on arXiv [4]. This paper reviews the growing range of research papers that have applied machine learning to computer architecture design problems, thereby serving as a resource for computer architects to better understand the breadth of existing applications and identify areas for future applications. Further analysis of techniques employed in these works provides a guide for newer machine learning practitioners to adopt best-practice strategies when beginning to apply machine learning in their own work.

1.2 Generalized Design Frameworks

Continued advancements in complex many-core designs necessitate sophisticated, yet broadly applicable, strategies for design space exploration. Although conventional design approaches involving heuristics have proven useful for many years, this paradigm has been pushed to the limit and, in many tasks, cannot provide acceptable results due to growing demands. Furthermore, tools built upon these principles are often limited in

their application to a specific task, set of design criteria, or system configuration, etc. These challenges have led to the adoption of deep reinforcement learning in computer architecture design.

The second manuscript in this thesis is the paper, "A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study," which was presented at the IEEE International Symposium on High Performance Computer Architecture [5]. This paper introduces a deep reinforcement learning for computer architecture design space exploration tasks. In this framework, efficient exploration is realized using a Monte-Carlo tree search (MCTS) that generates training for a deep neural network, which then guides MCTS towards promising subspaces in the overall design. Arbitrary design constraints are integrated directly into this search process, rather than relegated to an after-the-fact design selection process. This integration results in more precise control during search and more optimal designs. Additional functionality provided by multi-threaded exploration further increases the consistency and speed at which satisfactory designs can be generated. Practical application is demonstrated in a NoC design case study in which the deep reinforcement learning framework is shown to successfully explore a design space exceeding 10^{130} for 100-core chip multiprocessors (CMPs), all while adhering to strict constraints.

1.3 Proactive Resource Management

Increasingly parallel CMP designs provide greatly improved computational resources on a single device/machine, yet also pose a challenge in terms of resource management. In the past, service providers for high-priority latency-sensitive applications (e.g., Google Gmail and Microsoft Bing) have typically provided quality of service (QoS) guarantees by provisioning computing resources for the peak load. This practice ensures satisfactory performance at all times, but often leads to significant resource over-provisioning during periods of lower demand. Service providers have therefore sought to improve machine utilization (and lower total cost of ownership) by opportunistically co-scheduling best effort applications and granting these applications any unused resources. Nevertheless, naive co-scheduling can lead to significant contention between applications and adversely impact QoS guarantees. These challenges require more sophisticated and dynamic resource management strategies.

The third manuscript in this thesis is the paper, "Intelligence Resource Optimization for Edge Networks Using Machine Learning," which is an ongoing collaboration between Oregon State University and Intel's Network Platforms Group. This preliminary work describes a proactive resource manager that integrates several machine learning techniques. First, we describe how dynamic resource management, especially at fine-grained intervals, can benefit from proactive QoS predictions made by supervised learning models. Second, we detail a deep reinforcement learning control approach that can integrate proactive QoS predictions and eliminate the need for extended sampling periods found in existing works.

Chapter 2: General Background

The learning approach and the model are both fundamental considerations in applying machine learning to any problem. In general, there are four main categories of learning approaches: supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. These approaches can be differentiated by *what* data is used and *how* that data is used to facilitate learning. Similarly, many appropriate models may exist for a given problem, thus enabling significant diversity in application based on the learning approach, hardware resources, available data, etc. In the following, we introduce these learning approaches and several significant models for each learning approach, focusing on approaches with proven applicability.

2.1 Supervised learning

In supervised learning, the model is trained using input features and output targets, with the result being a model that can predict the output for new, unseen inputs. Common supervised learning applications include regression (predicting a value such as processor IPC) and classification (predicting a label such as the optimal core configuration for application execution).

Supervised learning models can be generalized into four categories: decision trees, Bayesian networks, support vector machines (SVMs), and artificial neural networks [6]. Decision trees use a tree structure where each node represents a feature and branches represent a value (or range of values) for that feature. Inputs are therefore classified by sequentially following branches based on the value of the feature being considered at a given node. Bayesian networks instead embed conditional relationships into a graphical structure; nodes represent random variables and edges represent conditional dependence between these variables. A performance prediction model, for example, can condition prediction for new applications on learned distributions for unobserved variables (i.e., underlying factors affecting performance) from other applications, as in [7]. SVMs are generally known for their function rather than a particular graphical structure (as in

decision tree and Bayesian networks). Specifically, SVMs learn the best dividing line (in 2-D) or hyperplane (in high dimensions) between examples, then uses examples along this hyperplane to make new predictions. SVMs can also be extended to non-linear problems using kernel methods [8] as well as multi-class problems. Finally, artificial neural networks (or simply neural networks) represent a broad category of models that are, again, defined by their structure, which is reminiscent of neurons in the human brain; layers of nodes/neurons are connected using links with learned weights, enabling particular nodes to respond to specific input features. Simple perceptron models contain just one weight layer, directly converting the weighted sum of inputs into an output. More complex DNNs include several (or many) layers of these weighted sums. Additional variants such as convolutional neural networks (CNNs) incorporate convolution operations between some layers to capture spatial locality while recurrent neural networks re-use the previous output to learn sequences and long-term patterns. All these supervised learning models can be used in both classification and regression tasks, although there are some distinct high-level differences. Variants of SVMs and neural networks tend to perform better for high-dimension and continuous features and also when features may be nonlinear [6]. These models, however, tend to require more data compared to Bayesian networks and decision trees.

2.2 Unsupervised learning

Unsupervised learning uses just input data to extract information without human effort. These models can therefore be useful, for example, in reducing data dimensionality by finding appropriate alternative representations or clustering data into classes that may not be obvious for humans.

Thus far, the primary two unsupervised learning models applied to architecture are principal components analysis (PCA) and k-means clustering. PCA provides a method to extract significant information from a dataset by determining linear feature combinations with high variance [9]. As such, PCA can be applied as an initial step towards building a model with reduced dimensionality, a highly desirable feature in most applications, albeit at the cost of interpretability. K-means clustering is instead used to identify groups of data with similar features. These groups may be further processed to generalize behavior or simplify representations for large datasets.

2.3 Semi-supervised learning

Semi-supervised learning represents a mix of supervised and unsupervised methods, with some paired input/output data, and some unpaired input data. Using this approach, learning can take advantage of limited labeled data and potentially significant unlabeled data. We note that this approach has, thus far, not yet found application in architecture. Nevertheless, one work on circuits analysis [10] presents a possible strategy that could be adapted in future work.

2.4 Reinforcement Learning

2.4.1 General Formulation

In reinforcement learning, an agent is sequentially provided with input based on an environment state and learns to perform actions that optimize a reward. For example, in the context of memory controllers, the agent replaces traditional control logic. Input could include pending reads and writes while actions could include standard memory controller commands (row read, write, pre-charge, etc.). Throughput could then be optimized by including it in the reward function. Given this setup, the agent will potentially, over time, learn to choose control actions that maximize throughput.

Reinforcement learning models applied to architecture, as a whole, can be understood using a representation based on states, actions, and rewards. The agent attempts to learn a policy function π, which defines the action a to take at a given state s, based on a received reward r [11]. A learned state-value function, following the policy, is then given as

$$V^{\pi}(s) = \mathbb{E}[\sum_{t \geq 0} \gamma^{\,t} * r_t | s_0 = s, \pi] \tag{2.1}$$

where γ is a discount factor (≤ 1), which dictates how much the model should consider future rewards. The cumulative rewards are then maximized by learning an optimal policy π^* that satisfies

$$\pi^*(s) = arg\max_{\pi} \mathbb{E}[\sum_{t \geq 0} \gamma^{t} * r_t | s_0 = s, \pi]. \tag{2.2}$$

Various models may implement different approaches to learn this optimal policy, but largely address the same problem of maximizing rewards. Q-learning is a noteworthy example that models an action-value function by estimating the value of an individual action, from a given state.

2.4.2 Deep Reinforcement Learning

Breakthroughs in deep learning have spurred researchers to rethink potential applications for deep neural networks (DNNs) in diverse domains. One result is deep reinforcement learning, which synthesizes DNNs and reinforcement learning concepts to address complex problems [12, 13, 14]. This synthesis mitigates data reliance without introducing convergence problems via efficient data-driven exploration based on DNN output. Recently, these concepts have been applied to Go, a grid-based strategy game involving stone placement. In this model, a trained policy DNN learns optimal actions by searching a Monte Carlo tree that records actions suggested by the DNN during training [13, 14]. Deep reinforcement learning can outperform typical reinforcement learning by generating a sequence of actions with better cumulative returns [12, 13, 14].

A Survey of Machine Learning Applied to Computer Architecture Design

Drew Penney and Lizhong Chen

arXiv
1909.12373

Chapter 3: A Survey of Machine Learning Applied to Computer Architecture Design

In the past decade, machine learning (ML) has rapidly become a revolutionary factor in many fields, ranging from commercial applications, as in self-driving cars, to medical applications, improving disease screening and diagnosis. In each of these applications, an ML model is trained to make predictions or decisions without explicit programming by discovering embedded patterns or relationships in the data. Notably, ML models can perform well in tasks/applications where relationships are too complex to model using analytical methods. These powerful learning capabilities continue to enable rapid developments in diverse fields. Concurrently, the exponential growth predicted by Moore's law has slowed, putting increasing burden on architects to supplant Moore's law with architectural advances. These opposing trends suggest opportunities for a paradigm shift in which computer architecture enables ML and, simultaneously, ML improves computer architecture, closing a positive-feedback loop with vast potential for both fields.

Traditionally, the relationship between computer architecture and ML has been relatively imbalanced, focusing on architectural optimizations to accelerate ML algorithms. In fact, the recent resurgence in AI research is, at least partly, attributed to improved processing capabilities. These improvements are enhanced by hardware optimizations exploiting available parallelism, data reuse, sparsity, etc. in existing ML algorithms. In contrast, there has been relatively limited work applying ML to improve architectural design, with branch prediction being one of a few mainstream examples. This nascent work, although limited, presents an auspicious approach for architectural design.

This paper presents an overview of ML applied to architectural design and analysis. As illustrated in Figure 3.1, this field has grown significantly in both success and popularity, particularly in the past few years. These works establish the broad applicability and future potential of ML-enabled architectural design; existing ML-based approaches, ranging from DVFS with simple classification trees to design space exploration via deep reinforcement learning, have already surpassed their respective state-of-the-art human expert and heuristic based designs. ML-based design will likely continue to provide

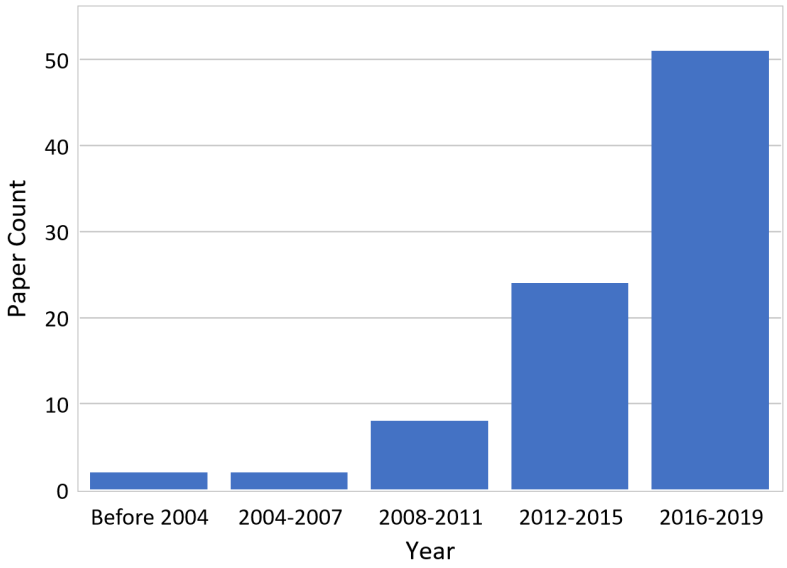


Figure 3.1: Publications on machine learning applied to architecture (for works examined in Section 3.1)

breakthroughs as promising applications are explored.

The paper is organized as follows. Section 3.1 presents existing work on ML applied to architecture. Section 3.2 then compares and contrasts implementation strategies in existing work to highlight significant design considerations. Section 3.3 identifies possible improvements and extensions to existing work as well as promising, new applications for future work. Finally, Section 3.4 concludes.

3.1 Literature Review

This section reviews existing work that applies machine learning to architecture. Work is organized by sub-system (when applicable) or primary objective. We focus on design and optimization, but also introduce general performance prediction work.

3.1.1 System Simulation

Cycle-accurate simulators are commonly used in system performance prediction, but require several orders of magnitude more time than native execution. ML can offset this

penalty through a trade-off between simulation time and accuracy. In general, ML can reduce execution time by 2-3 orders of magnitude with relatively high accuracy (task dependent, typically $> 90\%$). Early work by Ipek et al. [15] modeled architectural design spaces using an ANN ensemble (a group of ANN predictors). Models were trained on approximately 1% of the design space, then predicted CMP performance with 4-5% error for random points, albeit only in that specific configuration space. When combined with SimPoints, predictions exhibit slightly higher error, but the simulated instruction count is further reduced. Ozisikyilmaz et al. [16] additionally predicted SPEC performance for future systems that may be poorly modeled by existing simulators. Evaluation was limited to randomly-sampled data with relatively simple linear regression and neural network models, but nevertheless demonstrated advantages for pruned neural networks compared to standard single-layer models (as in [15]). Several other ML approaches have also been tested. Eyerman et al. [17] proposed a mechanistic-empirical model for processor CPI prediction. In this approach, they used a generic mechanistic model with parameters inferred by a regression model. Their model is limited to single-core performance prediction, but improves accuracy, ease of implementation (compared to purely mechanistic models), and interpretability (compared to purely empirical models). Zheng et al. [18, 19] explored cross-platform prediction from Intel/AMD to ARM processors using linear regression. Their first approach [18] made predictions based on a local neighborhood of examples around the target point to approximate non-linear behavior. They later [19] emphasized phase-level prediction, assuming that phase-level behavior would be approximately linear. Notably, average error for cycle count predictions is less than 1% using phase-level profiling. This approach is, however, restricted to a single target architecture and requires source code for phase-level analysis, leaving significant opportunities for future work. Finally, recent work by Agarwal et al. [20] introduced a method to predict parallel execution speedup using single-threaded execution characteristics. They trained separate models for each thread count using application-level performance counters. Although neural networks were omitted due to limited data, evaluation found that Gaussian process regression still provided promising results, particularly for high thread counts.

3.1.2 GPUs

Design Space Exploration: GPU design space exploration has proven to be a particularly favorable application for ML due to a highly irregular design space; some kernels exhibit relatively linear scaling while others exhibit very complex relationships between configuration parameters, power, and performance [21, 22, 23]. Jia et al. [21] proposed Stargazer, a regression-based framework based on natural cubic splines. Stargazer randomly samples approximately 300 points from a target design space (933K points in evaluation) for each application, then applies stepwise regression on these points. Notably, the framework achieves under 3.8% average performance prediction error. Wu et al. [22] instead explicitly modeled scaling for compute units, core frequency, and memory frequency. Scaling data from training kernels was processed using k-means clustering to group kernels by scaling behavior. An ANN then classifies kernels into these clusters, allowing new kernels to be classified and predictions made using cluster scaling factors. This approach, in contrast to Jia et al. [21], therefore requires just a few samples for new applications. Jooya et al. [23], similar to Jia et al. [21], considered a per-application performance/power prediction model, but additionally proposed a scheme to predict per-application Pareto fronts. Many ANN-based predictors were trained and the most accurate subset was used as an ensemble for prediction. Prediction accuracy was later improved by sampling points within a threshold of the previously predicted Pareto-optimal curve. Lin et al. [24] combined a performance predicting DNN with a genetic search scheme to explore memory controller placement. The DNN was used as a surrogate fitness function, obviating slow system simulations. The resulting placement improves system performance by 19.3%.

Cross-Platform Prediction: Porting applications for execution on GPUs is a challenging task with potentially uncertain benefits over CPU execution. Work has therefore examined methods to predict speedup or efficiency improvements using just CPU execution behavior. Baldini et al. [25] cast the problem as a classification task, training a modified nearest-neighbor and a support vector machine (SVM) model to determine, based on a threshold, whether GPU implementation would be beneficial. Using this approach, they predicted near-optimal configurations 91% of the time. In contrast, Ardalani et al. [26] trained a large ensemble of regression models to directly predict GPU performance for the code segment. Although several code segments exhibit high

error, the geometric mean of the absolute value of the relative error is still just 11.6% and the model successfully identifies several code segments (both beneficial and non-beneficial) that are incorrectly predicted by human experts. Later work by Ardalani et al. [27] introduced a completely static-analysis-based framework using a random forest model for binary classification. This approach eliminates both dynamic profiling and human guidance, instead using features such as instruction mix, branch divergence estimates, and kernel size to provide 94% accuracy for binary speedup classification (using a speedup threshold of 3).

GPU Specific Prediction & Classification: O'Neal et al. [28] presented a methodology for next-generation GPU performance prediction as cycles-per-frame (CPF) for DirectX applications. They focused on Intel GPUs, profiling earlier-generation architectures (e.g., Haswell GT2) to train next-generation predictors. They found that different models (i.e., linear vs non-linear) can produce more accurate results depending on the prediction target (Broadwell GT2/GT3 vs Skylake GT3), with the best performing models achieving less than 10% CPF prediction error. Recent work by Li et al. [29] presented a re-evaluation of commonly accepted knowledge of GPU traffic patterns. They used a CNN and t-distributed stochastic neighbor embedding on heatmap-transformed traffic data, identifying eight unique patterns with 94% accuracy.

Scheduling: GPU processing-in-memory (PIM) architectures can benefit from high memory bandwidth with reduced data movement energy. Despite this benefit, potential limitations on PIM compute capabilities may introduce complex trade-offs between performance and energy when scheduling execution on various resources. For this reason, Pattnaik et al. [30] proposed an approach using a regression model to classify core affinity, thus dividing the workload, and an additional regression model to predict execution time, enabling dynamic task migration. Performance and energy efficiency are improved by 42% and 27%, respectively, over a baseline GPU architecture. Further improvements are possible by improving core affinity classification accuracy (compared to regression).

3.1.3 Memory Systems and Branch Prediction

Caches: Heuristic approaches for caching can incur performance penalties due to dramatic workload variance. ML approaches can learn these intricacies and offer superior performance. Peled et al. [31] proposed a prefetcher exploiting semantic locality (data

structures) using contextual bandits (a simple RL variant), correlating contextual information and candidate addresses for prefetching. Implementation uses a two-level indexing method to dynamically control state information, allowing online feature selection with some additional overhead. Zeng and Guo [32] proposed a long short-term memory (LSTM) model (a recurrent neural network variant) for prefetching based on local history and offset-delta tables. Evaluation showed that the LSTM model enables accurate predictions over longer sequence and higher noise resistance than prior work. Several concerns relating to overhead and warm-up time are addressed, with potential solutions remaining for future work. Similarly, Braun et al. [33] extensively explored LSTM prefetching accuracy under several common access patterns. Experiments considered the impact of lookback size (access history window) and LSTM model size for several noise levels and independent access stream counts. Recent work by Bhatia et al. [34] synthesized traditional prefetchers with a perceptron-based prefetch filter, allowing aggressive predictions without degrading accuracy. Evaluation confirmed substantial coverage and IPC benefits offered by the proposed scheme, with 9.7% IPC speedup over the next best prefetcher when referenced to a no-prefetching four-core baseline. ML has similarly been applied to data reuse policies. For example, Teran et al. [35] predicted LLC reuse with a perceptron model. In this approach, input features are hashed to access saturating weight tables that are incremented/decremented based on correct/incorrect reuse prediction. These features are chosen empirically and shown to significantly impact performance, thus presenting an option for further optimization. Wang et al. [36] predicted reuse prior to cache entry, only storing data in the cache if there was predicted reuse. They used decision trees as a low-cost alternative to ensemble models, achieving 60-80% reduction in writes. Additional research has explored the growing performance bottleneck in translation lookaside buffers (TLBs). Margaritov et al. [37] proposed a scheme for virtual address translation in TLBs based on learned indices [38]. Evaluation showed nearly 100% accuracy for predicted indices, but practical implementation will require dedicated hardware to reduce calculation overhead (and is left for future work).

Schedulers & Control: Controllers for memory and storage systems influence both device performance and reliability, thus representing another strong application for ML models compared with heuristics. Ipek et al. [3] first proposed an RL approach for memory controllers to capture the balance between concurrency, delay, and several other factors. The model predicted optimal actions (precharge, activate, row read/write), im-

proving system performance by 15% (in a two-channel system) over prior work. Mukundan and Martinez [39] later built upon Ipek's work, generalizing the reward function to optimize energy, fairness, etc. They also added power-up and power-down actions to enable a further 8.6% improvement in performance and a significant improvement in energy efficiency. Related work optimizes communication energy between memory/storage and other systems using ML. Manoj et al. [40] proposed a Q-learning method for dynamic voltage swing control in through-silicon-interposer transmission lines. Predictions for power and bit error rate were quantized, then provided as input to the model to determine a new voltage level. Although their approach requires significant quantization to minimize overhead, they still achieved 15.1% energy savings compared to a static voltage baseline. Wang and Ipek [41] reduce data movement energy through online clustering and encoding. Several clusters are continuously updated at a bit-level using majority voting for data in that cluster. The total number of transmitted 1s is then minimized by XORing new data with the closest learned cluster center. Kang and Yoo [42] applied Q-learning to manage garbage collection in SSDs by determining optimal periods of inactivity. Key states are kept in the Q-table using LRU replacement, allowing a vast state space and, ultimately, a 22% average tail latency reduction over the baseline. Many states are, however, observed only once per workload, suggesting potential benefits using deep Q-learning (DQL). Other work directly considered system reliability. For example, Deng et al. [43] proposed a regression-based framework to dynamically optimize performance and lifetime in non-volatile memories. Their approach used phase-based application statistics to manage several conflicting policies for write latency, write cancellation, endurance, etc., guaranteeing a minimum lifetime with modest performance/energy improvements. Xiao et al. [44] proposed a method for disk failure prediction using an online random forest. They trained their model using a disk status window to account for imprecision in recorded failure date, enabling accurate predictions of soon-to-be faulty drives. Comparison against other random forest updating schemes (e.g., updating once a month) highlighted accuracy benefits from consistent training that may be extended to related domains.

Branch Prediction: Branch prediction is a noteworthy example of current ML application in industry, with accuracy surpassing prior state-of-the-art non-ML predictors. The perceptron-based branch predictor was first proposed by Jiménez and Lin [2] as a promising high-accuracy alternative to two-level schemes using pattern history tables.

Later research by St. Amant et al. introduced SNAP [45], a perceptron-based predictor implemented using analog circuitry to enable an efficient and practically feasible design. Perceptron weights and branch history were used to drive current-steering DACs that perform the dot product as the sum of currents. Jiménez [46] further optimized this design using a per-branch history table, dynamic coefficients for history importance, and a dynamic learning threshold. The optimized design achieves 3.1% lower MKPI than L-TAGE. Recent work with perceptron-based predictors by Garza et al. [47] explored bit-level prediction for indirect branches. Possible branch targets are evaluated using their similarity (dot product) with the combined weights from eight feature tables incorporating local and global history, ultimately reducing MKPI by 5% compared to ITTAGE. Currently, state-of-the-art conditional branch predictors (e.g., TAGE-SC-L [48]) still hide significant IPC gains (14.0% for an Intel Skylake architecture) in just a few hard-to-predict (H2P) conditional branches [49]. Tarsa et al. [49] consequently proposed "CNN Helper" predictors that target specific H2Ps using simple two-layer CNNs. Results indicate strong applicability across diverse workloads and present a promising area for future work.

3.1.4 Networks-on-Chip

DVFS & Link Control: Modern computing systems exploit complex power control schemes to enable increasingly parallel architectural designs. Heuristic schemes may fail to exploit all energy-saving opportunities, particularly in dynamic network-on-chip (NoC) workloads, leading to significant benefits through proactive ML-based control. Savva et al. [50] implemented dynamic link control using several ANNs, each of which monitors a NoC partition. These ANNs used just link utilization to learn a dynamic threshold to enable/disable links. Despite energy savings, their approach can cause high latency under dimension-ordered routing. DiTomaso et al. [51] relocated flit buffers to the links and dynamically controlled both link direction and power-gating with per-router classification trees. Using a simple three-level tree to limit overhead, overall NoC power is reduced by 85% and latency is reduced by 14% compared to a concentrated mesh. Winkle et al. [52] explored ML-based power scaling in photonic interconnects. Even a simple linear regression model provided promising results, negligibly reducing throughput (versus no power-gating) while reducing laser power consumption by 42%.

Reza et al. [53] proposed a multi-level ANN control scheme that considered both power and thermal constraints on task allocation, link allocation, and node DVFS. Individual ANNs classified appropriate configurations for local NoC partitions while a global ANN classified optimal overall resource allocation. This scheme identifies the global optimal NoC configuration with high accuracy (88%), but uses complex ANNs that could impact implementation. Clark et al. [54] proposed a router design for DVFS and evaluated several regression-based control strategies. Variants predicted buffer utilization, change in buffer utilization, or a combined energy and throughput metric. This work was expanded by Fettes et al. [55], who introduced an RL control strategy. Both regression and RL models enable beneficial tradeoffs, although the RL strategy is most flexible.

Admission & Flow Control: As with NoC DVFS, both admission and flow control can benefit from proactive prediction. Early work by Boyan and Littman [56] introduced Q-learning based routing in networks using delivery time estimates from neighboring nodes, noting throughput advantages over traditional shortest path routing for high traffic intensity. Several works have expanded upon Q-routing, observing application in dynamically changing NoC topologies [57], improved capabilities in bufferless NoC fault-tolerant routing [58], and high-performance congestion-aware non-minimal routing [59]. More recent works have instead focused on injection throttling and hotspot prevention. For example, Daya et al. [60] proposed SCEPTER, a bufferless NoC using single-cycle multi-hop paths. They controlled injection throttling using Q-learning to maximize multi-hop performance and improve fairness by reducing contending flits. Future work could reduce Q-table overhead which scales with NoC size in their implementation. Wang et al. [61] instead used an ANN to predict optimal injection rates for a standard buffered NoC. Additional preprocessing (to capture both spatial and temporal trends) and node grouping enables high accuracy predictions (90.2%) and reduces execution time by 17.8% compared to an unoptimized baseline. Soteriou et al. [62] similarly explored ANN-based injection throttling to reduce NoC hotspots. The ANN was trained to predict hotspots while recognizing the impact of proposed injection throttling and dynamic routing, providing a holistic mitigation strategy. The model provides state-of-the-art results for throughput and latency under synthetic traffic, but limited improvement under real-world benchmarks, suggesting the potential for further optimization. Another Q-learning approach, proposed by Yin et al. [63], used DQL to arbitrate NoC traffic. They considered a wide range of features and rewards while noting that the proposed

DQL algorithm is impractical due to overhead. Regardless, evaluation exhibited modest throughput improvements over round-robin arbitration.

Topology & General Design: Several works also applied ML to higher-level NoC topology design, involving trade-offs between power and performance, with some further considering thermals. Das et al. [64] used a ML-based *STAGE* algorithm to efficiently explore small-world inspired 3D NoC designs. In this approach, design alternates between base/local search (adding/removing links in a hill-climbing approach) and meta search (predicting beneficial starting points for local search using prior results). The same model was used again by Das et al. [65] to balance link utilization and address TSV reliability concerns. The STAGE algorithm was then enhanced by Joardar et al. [66] to optimize a heterogeneous 3D NoC design. The models explores multi-objective trade-offs between CPU latency, GPU throughput, and thermal/energy constraints. All three works [64, 65, 66] still rely upon hill-climbing for optimization. Recent work by Lin et al. [67] instead explored deep reinforcement learning in routerless NoC design. They used a Monte Carlo tree search to efficiently explore the search space and a deep convolutional neural network to approximate both the action and policy functions, thereby optimizing loop configurations. Further, the proposed deep reinforcement learning framework can strictly enforce design constraints that may be violated by prior heuristic or evolutionary approaches. Rao et al. [68] investigated multi-objective NoC design optimization across a broad SoC feature space (from bandwidth requirements to SoC area). ML models were trained using data from thousands of SoC configurations to predict optimal NoC designs based on performance, area, or both. Limited comparisons against human-expert designs did not consider alternative techniques (e.g., AMOSA [69]), yet exhibited some promising results, motivating research into effective features and models as well as further comparisons against alternative techniques.

Performance Prediction: Existing NoC models based on queuing theory are generally accurate, but rely on assumptions of traffic distribution that may not hold for real applications [70]. Qian et al. [70] emphasized how ML-based approaches can relax the assumptions made by queueing theory models. They constructed a mechanistic-empirical model based on a communication graph, using support vector regression (SVR) to relate several features and queuing delays. Evaluation showed lower error (3% error vs 10% error) than an existing analytical approach. Sangaiah et al. [71] considered both NoC and memory configuration for performance prediction and design space exploration. Follow-

ing a standard approach, they sampled a small portion of the design space, then trained a regression model to predict the resulting system CPI. Evaluation generally showed high accuracy, but lower accuracy for high-traffic workloads (median error of 24%). Additional design space exploration exhibited promising results, reducing the design space from 2.4M points to less than 1000.

Reliability & Error Correction: Overhead introduced by error correction in NoCs can be significant, especially when re-transmission is required. Several works have, therefore, explored ML-based control schemes. DiTomaso et al. [72] trained a decision tree to predict NoC faults using a wide range of parameters including temperature, utilization, and device wear-out. These predictions allow proactive encoding (on top of the baseline cyclic redundancy check) for transmission that are likely to have errors. Wang et al. [73] adopted a similar strategy for dynamic error mitigation, but used an RL-based control policy to eliminate the need for labeled training examples. Their approach provides an average of 46% dynamic power savings (17% better than the decision tree method [72]) compared with a static CRC scheme. In both cases, ML-based proactive control chose a more efficient scheme than CRC only. Wang et al. [74] subsequently proposed a holistic framework for NoC design incorporating dynamic error mitigation, router power-gating, and multi-function adaptive channel buffers (MFAC buffers). They emphasized comprehensive benefits through synergistic integration/control of several architectural innovations, thus achieving substantial improvements in latency (32%), energy-efficiency (67%), and reliability (77% higher Mean Time to Failure) compared to a SECDED baseline.

3.1.5 System-level Optimization

Energy Efficiency Optimization: Significant work has begun to consider systems in which workload execution is constrained by total energy consumption rather than processing resources. Control schemes incorporating ML have shown promise in optimizing energy efficiency with minimal performance reduction, often enabling 60-80% reductions in the energy-delay product compared to race-to-idle schemes. Won et al. [75] introduced a hybrid ANN + PI (proportional-integral) controller scheme for uncore DVFS. They initially trained the ANN offline, then refined predictions online using the PI controller. This hybrid scheme was shown to reduce the energy-delay product by 27% compared

to a PI controller alone, with less than 3% performance degradation compared to the highest V/F level. Pan et al. [76] implemented a power management scheme using a multi-level RL algorithm. Their method propagates individual core states up a tree structure while aggregating Q-learning representations at each level. Global allocation is made at the root, then decisions are propagated back down the tree, enabling efficient per-core control. Bailey et al. [77] addressed power efficiency in heterogenous systems. Similar to Wu et al. [22], they clustered kernels by their scaling behavior to train multiple linear regression models. Runtime prediction used two sample configurations, one from CPU execution and one from GPU execution, to determine the optimal configuration. Lo et al. [78] focused on energy-efficiency optimization for real-time interactive workloads. They used linear regression to model execution time based on annotations and code features, enabling stricter service level guarantees at the cost of applicability when source code is unavailable. Mishra et al. [79] also addressed real-time workloads, combining control theory and several ML-based models. Their framework was realized by offloading learning to a server, allowing low overhead DVFS that reduces energy consumption by 13% compared to the best prior approach. Related work by Mishra et al. [7] applied a comparatively complex hierarchical Bayesian model to combine both offline and online learning. In this approach, they accepted a high execution time penalty (0.8s) in order to provide significantly more accurate predictions than online or offline training alone. This approach therefore targeted longer executing workloads, but can provide more than 24% energy savings over the next best approach. Bai et al. [80] implemented a RL-based DVFS control policy adapted to a novel voltage regulator hierarchy using off-chip switching regulators and on-chip linear regulators. Individual RL agents adapt to a dynamically allocated power budget determined by a heuristic bidding approach. The design was enhanced using adaptive Kanerva coding [81] to limit area/power overhead and experience sharing to accelerate learning. Chen and Marculescu [82] (later Chen et al. [83]) explored an alternative two-level strategy for RL-based DVFS. Similar to Bai et al. [80], they used RL agents at a fine-grain core level to select a V/F level based on an allocated share of the global power budget. They achieved further improvement by allocating power budget using a performance-aware, albeit still heuristic-based, variant that considers relative application performance requirements. Imes et al. [84] explored single-application system energy optimization for a broader range of configurations options including socket allocation, HyperThread usage, and processor DVFS. They

identified several useful models, while noting that further work could optimize models and parameters. Analysis also provided insight into the benefit from single-model multi-resource optimization, particularly for neural networks. Finally, recent work by Tarsa et al. [85] considered an ML framework for post-silicon CPU adaptations using firmware updates to microcontroller-implemented models. Significant accommodations for statistical blindspots limit the rate of service-level-agreement violations while optimizing performance per watt for both general-purpose and application-specific deployment.

Task Allocation and Resource Management: In addition to energy control, ML offers an approach to allocate resources to tasks or tasks to resources by predicting the impact of various configurations on long-term performance. Lu et al. [86] proposed a thermal-aware Q-learning method for many-core task allocation. The agent considered only current temperature (i.e., no application profiling or hardware counters), receiving higher rewards for task assignments resulting in greater thermal headroom. Evaluation indicated an average 4.3°C reduction in peak temperature compared to a heuristic approach. Nemirovsky et al. [87] introduced a method for IPC prediction and task scheduling on a heterogeneous architecture. They predicted IPC for all task arrangements using ANNs, then selected the arrangement with the highest IPC. Evaluation highlighted significant throughput gains ($> 1.3x$) using a deep (but high overhead) neural network, indicating one possible application for pruning (discussed in Section 3.3.2). Recent work has also explored multi-level scheduling in hybrid CPU-GPU clusters. Zhang et al. [88] proposed a deep reinforcement learning (DRL) framework to divide video workloads, first at the cluster level (selecting a worker node) and then at the node level (CPU vs GPU). The two DRL models act separately, but still work together to optimize overall throughput. Allocating resources to tasks is another possible approach. Early work by Bitirgen et al. [89] considered a system with four cores and four concurrent applications. In their approach, per-application ANN ensembles predicted IPC for 2,000 configurations at each interval (500K cycles). IPC predictions were then aggregated to choose the highest performing overall system configuration. Scaling concerns for per-application ensembles and exponentially increasing configuration spaces could be addressed in future work. Recent research has also considered low-level co-optimization involving multiple components/resources. For example, Jain et al. [90] explored concurrent optimization of core DVFS, uncore DVFS, and dynamic LLC partitioning. These options are optimized by individual agents (potentially limiting co-optimization opportunities) at a relatively

large interval (1B instructions). Evaluation nevertheless indicated noteworthy reductions in energy-delay-product through multi-resource optimization. Finally, work by Ding et al. [91] established a somewhat contradictory trend between model accuracy and system optimization goals based on improvements for data scarcity and model bias. Specifically, they found that state-of-the-art models exhibit diminishing returns for accuracy and instead benefit from domain knowledge (e.g., focus sampling on the optimal front).

Chip Layout: Work by Wu et al. [92] demonstrated uses for ML in chip layout, deviating from the common applications including control, prediction, and design space exploration. They used k-means to cluster flip-flops during physical layout, minimizing clock wirelength at the expense of signal wirelength, noting that clock networks can consume more than 40% of chip power. They included constraints on maximum flip-flop displacement and cluster size, generating designs with 28.3% reduced displacement, 3.2% reduced total wirelength, and 4.8% reduced total switching power compared to the prior state-of-the-art approach.

Security: Malware detection, a traditionally software-based task, has been explored using machine learning to enable reliable hardware-based in-execution detection. For example, Ozsoy et al. [93] test both logistic regression (LR) and neural network classifiers trained on low-level hardware counters. Optimization based on reduced precision and feature selection provides high accuracy (100% malware detection and less than 16% false positives) with minimal overhead (0.04% core power and 0.19% core logic area) for the LR model.

3.1.6 ML-Enabled Approximate Computing

Approximate computing has many facets, including circuit level approximation (such as reduced precision adders), control level approximation (relaxing timings, etc), and data level approximation. Methods using ML generally fall within this last category, offering a powerful function/loop approximation technique that commonly provides 2-3 times application speedup and energy reduction with limited impact on output quality. Esmaeilzadeh et al. [94] introduced NPU, a new approach to programmable approximation using neural networks. They developed a framework to realize Parrot transformations that translate annotated code segments into neural networks approximations. Tightly integrating the NPU with the CPU allowed an average 2.3x speedup and 3.0x energy

reduction in studied applications. This framework was later extended by Yazdanbakhsh et al. [95] to implement neural approximation on GPUs. Neural approximation was integrated into the existing GPU pipeline, enabling component re-use and approximately 2.5x speedup and 2.5x reduced energy. Grigorian et al. [96] presented a different approach for a multi-stage neural accelerator. Inputs are first sent through a relatively low accuracy/overhead neural accelerator, then checked for quality; acceptable results are committed, while low quality approximations are forwarded to an additional, more precise, approximation stage. The problem with these works is that error is either constant [94, 95] or requires several stages with potentially redundant approximation [96]. For that reason, Mahajan et al. [97] introduced MITHRA, a co-designed hardware-software control framework for neural approximation. MITHRA implements configurable output quality loss with statistical guarantees. ML classifiers predict individual approximation error, allowing comparison to a quality threshold. Recent work by Oliveira et al. [98] also explored approximation using low-overhead classification trees. Even with software-based execution, they achieved application speedup comparable to an NPU [94] hardware implementation. Finally, ML has also been used to mitigate the impact of faults in existing approximate accelerators. Taher et al. [99] observed that faults tend to manifest in a similar manner across many input test vectors. This observations enables effective error compensation using a classification/regression model to correct output based on predicted faults for a given input.

3.2 Analysis of Current Practice

This section examines varying techniques employed in existing work. These comparisons emphasize potentially useful design practices and strategies for future work.

Work is divided into two categories that represent a natural division in design constraints and operating timescales and therefore correspond to differing design practices. The first category, online ML application, encompasses work that directly applies ML techniques at run-time, even if training is performed offline. Design complexity in this work is therefore inherently limited by practical constraints such as power, area, and real-time processing overhead. The second category, offline ML application, instead applies ML to guide architectural implementation, involving tasks such as design and simulation. Consequently, models for offline ML application can exploit higher complexity and

higher overhead options at the cost of training/prediction time.

3.2.1 Online ML Application

Model Selection: Online ML applications primarily use either decision trees or ANNs, in the case of supervised learning models, and either Q-learning or deep Q-learning, in the case of RL models. Note that tasks for these learning approaches are not necessarily disjoint, particularly for control. Fettes et al. [55] cast DVFS as both a supervised learning regression task and as a reinforcement learning task. The supervised learning approach predicted buffer utilization or change in buffer utilization to determine an appropriate DVFS mode. In contrast, the RL approach directly used DVFS modes as the action space. Both models can perform well, but the RL model is more universally applicable since the energy/throughput trade-off can be tailored to application needs and does not require threshold tuning. This certainly does not mean that RL is a one-model-fits-all solution. Supervised learning models find strong application in function approximation [94, 95, 96, 98] and branch prediction tasks [45, 46], which are far less suitable (if not impossible) to approach using RL since these tasks cannot be represented well as a sequence of actions.

Implementation & Overhead: Implementation of online ML applications highlight limitations in data availability, storage space for models, etc., indicating the need for an efficient, and generally low complexity, model. These limitations will likely become more important to consider as more research moves towards real-world implementation.

Several NoC-based works [50, 60, 75] have applied different methods for global data collection to support ML models. Daya et al. [60] implemented self-learning injection throttling using a separate bufferless starvation network that carries a starvation flag, encoded as a one-hot N-bit vector for a network with N nodes. These starvation vectors are propagated to all nodes, allowing individual node-based Q-learning agents to determine appropriate injection throttling. Soteriou et al. [62] similarly used a dedicated networks to collect buffer utilization and VC occupancy statistics. The ANN-based DVFS control proposed by Won et al. [75] eschewed an additional status/data network by encoding data into unused bits in standard packet headers. Data is opportunistically collected by a central control unit as packets pass through its router. This method introduces potential concerns about data staleness, but prior work [100] observed nearly identical

performance to omniscient data collection for sufficiently large (50K cycle) control windows. Smaller time windows can be accommodated by sending dedicated packets, as done by Savva et al. [50].

Implementation can also consider the use of either hardware or software models. Implementation using dedicated hardware will usually experience lower execution time overhead, but there are other considerations. Esmaeilzadeh et al. [94] implemented a neural processor (NPU) for function approximation using a dedicated hardware module. They also considered a software implementation, but observed a prohibitive increase in instruction count for software execution compared to a baseline x86 function. Later work by Oliveira et al. [98] found that function approximation using a simple classification tree can achieve comparable results to NPU [94] for application speedup and error rate in several applications (albeit somewhat worse on average). Their purely software implementation highlights a trade-off between area/power and accuracy/performance. Won et al. [75] observed a similar trade-off, choosing to implement an ANN in software using an on-die microcontroller rather than dedicated hardware. This implementation consumes several orders of magnitude more cycles (15K cycles for inference), but requires 50mW less average power than a hardware implementation.

Approaches for hardware implementation may also vary based on the task. A "standard" ANN implementation is observed in work by Savva et al. [50]. They incorporated a finite state machine for control, an array of multiply-accumulate (MAC) units for calculation, a register array to load and store results, and a lookup-table-based activation function. Both MAC array width and calculation precision can be adjusted to balance power/area and accuracy/speed. In contrast, St. Amant et al. [45] implemented a perceptron branch predictor using a mixed signal design. They realized dot products in analog circuitry, leveraging transistor sizing and current summing to achieve a feasible overhead. Variance also exists in hardware for RL models. The "standard" Q-learning implementation requires a lookup table to store state-action values. Ipek et al. [3] as well as Mukundan and Martinez [39] instead used $CMAC$ [101], replacing a potentially extensive Q-learning table with multiple coarse-grain overlapping tables. This approach also included hashing, using hashed state attributes to index the CMAC tables. Taken together, these two methods balance generalization and overhead, although may introduce collisions/interference depending on the task. Further pipelining the hashing, CMAC table lookup, and calculation allows more possible action-values to be evaluated

per cycle.

Optimization: Online ML applications with online training benefit from adaptivity to run-time workload characteristics. Despite these benefits, low model accuracy can negatively impact system performance, most notably at the start of execution or during periods of high variance in workload characteristics. Adaptations to control and learning can be considered to avoid these detrimental impacts. Some RL-based work [31] considered mitigating the impact of poor actions during exploration by introducing "shadow" operations. These operations are low confidence actions suggested by the model that are still used in model updates but not executed by the system. Consequently, the model gains feedback on the goodness of the action without negatively impacting the system. In a supervised learning based control task, Won et al. [75] trained an ANN online using control actions made by a PI controller, which exhibits far less start-up delay. Following training, control decisions are made using a hybrid combination based on error and consistency, allowing complementary control. In the simplest case, checking the performance of a default configuration, as in [43], provides a guarantee that the ML model will not perform worse than the default, but can perform better.

In most works, ML models replace existing approaches (commonly a heuristic). Nevertheless, several recent works [34, 49] have demonstrated significant advantages by combining both traditional (non-ML) and ML approaches. These improvements are derived from the orthogonal prediction/decision-making capabilities of the two approaches, thus enabling synergistic performance improvements. This method can also enable lower-cost ML application by focusing on particular shortcomings in traditional approaches. Both recent works [34, 49] consider just branch prediction, thus significant opportunities exist to explore this potential co-design paradigm.

3.2.2 Offline ML Applications

Model/Feature Selection: Offline ML applications generally exhibit substantial model/feature diversity since the model itself is not tied to a particular architecture. Model and feature selection therefore focuses more on maximizing model accuracy while minimizing overall learning/prediction time. Design space exploration, in particular, can be approached using either iterative search methods for direct optimization or supervised learning methods to select optimal points based on the predicted optimality of a design. Several works

[64, 65, 66] used an iterative *STAGE* [102] algorithm that optimizes local search for 3D NoC links by learning an evaluation function to predict local search results from a given starting point. Recent work has instead applied deep reinforcement learning [67] to routerless NoC design. The proposed Monte Carlo tree search, along with actions suggested by a convolutional neural network, provide a highly efficient search process. Parallel threads are also utilized to scale design space exploration with increasing computational resources. System-level design space exploration has favored more standard supervised learning approaches [23, 68, 71]. Specific model choices vary, with linear [23, 68] and non-linear [71] regression models, as well as random forests and neural networks [68] finding implementation. As in online ML applications, discussed in Section 3.2.1, some tasks are naturally limited to supervised learning methods. Cross-architecture prediction is an exemplar [18, 19, 21, 25, 26].

Optimization: The usefulness of an ML model in offline ML applications is largely determined by overhead relative to traditional design approaches. Optimization therefore primarily focuses on improving data efficiency and overall model accuracy.

Ensemble methods have been proposed in online ML applications [43], but primarily find application in offline ML applications as ensembles can be made arbitrarily large (relative to available computation resources). Several optimizations have been suggested to improve efficiency. Jooya et al. [23] trained many neural networks using slightly different configurations and generated an ensemble using a subset of the models that generalized well and were most insensitive to input noise. They further introduced outlier detection by filtering predictions whose performance and/or power predictions differ greatly from the closest configuration in training data. Ardalani et al. [26] instead kept all 100 models that they trained, noting that models may be very strong predictors in one application but weak predictors in another. They remedied this dilemma by selecting only the 60 closest individual predictions to the median prediction.

Sampling method optimization, while not unique to architecture tasks, are nevertheless important to consider in improving model accuracy. Sangaiah et al. [71] considered potential systematic biases in their uncore performance prediction model. Specifically, they observed that uniform random sampling may not adequately capture performance relationships in a non-uniform configuration space (as in cache configurations using powers of two for sizing). They therefore used a low-discrepancy sampling technique, *SOBOL* [103], to remove this systematic bias and prevent performance over-prediction

for low-end configurations.

3.2.3 Domain Knowledge & Model Interpretation

The powerful relationship learning capabilities offered by ML algorithms enable black-box implementation in many tasks (i.e., without consideration for task-specific characteristics), but may fail to capitalize on additional domain knowledge that could improve interpretability or overall model performance. Additionally, in some applications, domain knowledge can help identify aberrant behavior and, again, improve overall model usefulness. These themes are highlighted in several specific works, but can be generally applicable for ML applied to architecture.

One approach uses mechanistic-empirical models, synthesizing a domain knowledge based mechanistic framework with empirical ML based learning for specific parameters. These models simplify implementation compared to purely mechanistic models [17], can avoid incorrect assumptions made in purely mechanistic models [70], and can offer higher accuracy than purely empirical models by avoiding overfitting [17]. Eyerman et al. [17] also demonstrated how these models can be used to construct CPI stacks, allowing meaningful alternative design comparisons.

Deng et al. [43], in their work predicting optimal NVM write strategies, presented a case for tuning ML models based on task specific domain knowledge. Following initial analysis, they discovered how a single configuration parameter (wear quota) can result in higher complexity and sub-optimal prediction accuracy for IPC and system energy, even with quadratic regression and gradient boosting models. Excluding wear quota from the configuration space, then later applying it to the predicted optimal configuration, provided a 2-6% improvement in prediction accuracy. Ardalani et al. [26] similarly examined inherent imperfections in their learning model for cross-platform performance prediction. Some predictions can be easy for learning models and hard for humans, representing an ideal scenario for ML application; the converse can also be true. In both cases, ML application is strengthened by considering task characteristics.

3.3 Future Work

This section synthesizes observations and analysis from Section 3.1 and Section 3.2 to identify opportunities and detail the need for future work. These opportunities may come at the model level, exploiting improved implementation strategies and learning capabilities, or at the application level, addressing the need for generalized tools or exploring altogether new areas.

3.3.1 Investigating Models & Algorithms

Existing works generally apply ML at a single time-scale or level of abstraction. These limitations motivate investigation into models and algorithms that capture the hierarchical nature of architecture, both in terms of system design and execution characteristics.

Perform Phase-level Prediction: Application analysis using basic blocks [104] has long been a useful method for simulation, made possible by identifying unique and representative phases in program execution. Phase-level prediction offers analogous benefits for ML applied to architecture. A few recent works, in particular, have demonstrated promising results, with high accuracy for both performance prediction [19] as well as energy and reliability (lifetime) [43]. In general, most work [7, 23, 71] has not yet adopted phase-level prediction techniques (or does not explicitly mention their methodology). Specifically, future work could explore predictions for control and system reconfiguration based on phase-level behavior, rather than either static windows [89] or application-level behavior [79, 105].

Exploit Nanosecond Scale: Coarse-grain ML, used in many DVFS control schemes, provides significant benefits over standard control-theory based schemes, yet fine grain control can provide even greater efficiency. Specifically, analysis by Bai et al. [80] indicated very rapid changes in energy consumption, on the order of 1K instructions for some applications. Exploiting these brief intervals requires careful consideration for both the model and the algorithm. Future work may optimize existing algorithms such as experience sharing [106] and hybrid/tandem control [75], or consider approaches more suited for novel models (e.g., hierarchical models). These approaches could also enable additional nanosecond-scale co-optimization opportunities, such as dynamic LLC partitioning, to extract further efficiency gains.

Apply Hierarchical & Multi-agent Models: Application execution in computer systems naturally follows a hierarchical structure in which, at the top level, tasks are allocated to cores, then cores are assigned dynamic power and resource budgets (e.g., LLC space), and finally, at the bottom level, data/control packets are sent between cores and memory. Consequently, a single machine learning model may struggle to learn appropriate design/control strategies. Furthermore, in the case of reinforcement learning models, it can be exceedingly difficult to accurately assign credits to specific low-level actions based on their impact on overall execution time, energy efficiency, etc. One promising approach in recent work is hierarchical models [107]. Hierarchical reinforcement learning models enable goal-directed learning that is particularly beneficial in environments with sparse feedback (e.g., task allocation). Applying hierarchical learning to architecture could therefore enable more effective multi-level design and control. Multi-agent models are another promising area in machine learning research. These models tend to focus on problems in which reinforcement learning agents have only partial observability of their environment. Although partial-observability may not be a primary concern in individual computer systems, recent work [108] has applied this concept to internet packet routing and demonstrated convergence benefits via improved cooperation between individual agents.

3.3.2 Enhancing Implementation Strategies

Increasingly complex models require effective strategies and techniques to reduce overhead and enable practical implementation. Model pruning and weight quantization, as discussed below, are two particularly effective techniques with proven benefits in accelerators, while many other promising approaches are also being explored [109].

Explore Model Pruning: Model complexity can be a limiting factor in online ML applications. A standard Q-learning approach requires a potentially extensive table to store action-values. Neural network based approaches for both RL (in Deep Q-Networks) and supervised learning require network weight storage and additional processing capabilities. Neural networks, in particular, are therefore generally constrained to a few layers in existing work, with many using just one hidden layer [50, 75, 89, 97] and some using one or two hidden layers [94, 95].

Recent research on neural networks has demonstrated promising methods to reduce

model complexity through pruning [110, 111]. The general intuition is that many connections are unnecessary and can therefore be pruned. Iteratively pruning a high-complexity network, then re-training from scratch on the sparse architecture achieves good results, with some work demonstrating very high sparsity (>90%) and little accuracy penalty [111].

Pruning applied to neural networks, either in deep Q-learning or supervised learning regression/classification, offers a method to train complex models for high accuracy, then prune for feasible implementation. Deep Q-learning application has, thus far, been limited to two works [55, 63], one of which is currently impractical to implement [63]. Future work may instead consider pruned deep Q-networks as a useful alternative to standard Q-learning approaches. Pruning also provides a substantial opportunity for future work on performance prediction (as in DVFS control) and function approximation (as in ML-enabled approximate computing). System-level approximation (discussed in Section 3.3.4) may particularly benefit from pruning high complexity models.

Explore Quantization: Existing work primarily applies quantization to state values in Q-learning to enable practical Q-table implementation. Similarly, neural networks benefit from potential reduction in execution time, power, and area by reducing multiply-accumulator precision. Recent works, however, suggest a new spectrum of opportunities for alternative hardware implementations based on reduced precision models.

Binary neural networks, for example, quantize weights to be either +1 or -1, enabling computation based on bit-wise operations rather than arithmetic operations [112]. An additional approach considered quantizing neural network weights into finite (but non-binary) subsets in order to replace multiply operations with lookup-table accesses [113], allowing higher precision and lower execution time, albeit with potentially higher area cost. Future work on ML application can exploit similar hardware implementations while exploring optimal quantization levels for various tasks and control schemes.

3.3.3 Developing Generalized Tools

Existing machine learning tools (e.g., scikit-learn [114]) have proven useful for relatively simple ML applications. Nevertheless, complex design and simulation tasks require more sophisticated tools to enable rapid task-specific optimizations using general-purpose frameworks.

Enable Broad Application & Optimization: Purpose-built architectural tools, similar to heuristic design strategies, can be useful in enabling design, exploration, and simulation that satisfies a common use case. These approaches may still be limited in their application to a specific problem, optimization criteria, system configuration, etc. Given the fast-paced nature of architectural research (and machine learning research), there is a need to develop more generalized tools and easily modifiable frameworks to address broader applications and optimization options.

ML-based design tools are especially promising, with recent works demonstrating successful application to immense design spaces (e.g., exceeding 10^{12} in [67]). Opportunities for new design tools are not, however, limited to specific architectural components. Chip layout is a notable example in which even simple clustering algorithms can dramatically outperform existing heuristic approaches [92]. Future work can also continue to develop more broadly applicable tools for performance and power prediction. In particular, recent work on cross-platform performance prediction [27] suggests the possibility for high prediction accuracy with purely static features, thus representing another potential area for additional research.

Enable Widespread Usage: Generalized tools enable additional benefit by facilitating rapid design and evaluation. Using a machine learning approach, one might simply modify training data (in a supervised learning setting) or action/reward representation (in a reinforcement learning setting) rather than exploring models, data representation strategies, search approaches, etc., possibly without *a priori* machine learning experience. For example, recent work [67] envisioned reuse of a deep reinforcement learning framework for diverse NoC-related design tasks involving interposers, chiplets, and accelerators. While the framework might not be compatible with all work, especially in novel areas, it may provide a better foundation for machine learning application to architectures, especially for individuals with limited machine learning background.

3.3.4 Embracing Novel Applications

Opportunities abound for future work to apply ML to both existing and emerging architectures, replace heuristic approaches to enable long-term scaling, and advance capabilities for automated design.

Explore Emerging Technologies: Several proposals [36, 42, 43, 44] establish how

ML can be used to optimize both standard (energy, performance) and non-standard (lifetime, tail-latency) criteria. These non-standard criteria are shown to be particularly problematic in emerging technologies as these technologies cannot easily find widespread implementation without some reliability guarantees. Applying ML to optimize both standard and non-standard criteria therefore provides a method for future work to intelligently balance control strategies dynamically, rather than relying upon a heuristic approach.

Explore Emerging Architectures: ML application to emerging architectures presents a similar benefit by enabling rapid development, even with limited best-practice knowledge, which may take time to develop. Work in long-standing design areas, such as task allocation and branch prediction, may incorporate best-practice domain knowledge to guide approaches, whether applying ML or some other traditional method. Best practices for emerging architectures may not be immediately obvious. For example, ML application to 2D photonic NoCs [52], 2.5D processing-in-memory designs [30], and 3D NoCs [64, 65, 66] have all shown strong performance over existing approaches. Future work can explore ML application to novel concerns such as connectivity and reconfigurability in interposers and domain-specific accelerators.

Expand System-Level Approximate Computing: As discussed in Section 3.1.6, ML applications for approximate computing have been mostly limited to function approximation. However, there are many other facets of approximate computing that have already been implemented in non-ML works, which can be reap additional benefits by utilizing ML. For example, APPROX-NoC [115] reduces network traffic using approximated and encoded data. Another work explored a multi-faceted approximation scheme for a smart camera system [116] using approximate DRAM (lower refresh rate), approximate algorithms (loop skipping) and approximate data (lower sensor resolution). Existing compiler-based work [117] for system-wide approximation enhances prior capabilities to determine approximable code, but relies upon heuristic searches with representative inputs. Consequently, this method does not provide statistical guarantees, such as those in MITHRA [97]. Future work may explore searches based on deep reinforcement learning (or perhaps hierarchical reinforcement learning) to incorporate existing approximation techniques into a scalable framework for high-dimensional approximation and co-optimization.

Implement System-Wide, Component-Level Optimization: Recent work has

begun to explore broader ML-based design and optimization strategies. MLNoC [68] explores a wide SoC feature space for NoC design optimization. Core and uncore DVFS are combined in Machine Learned Machines [90], along with LLC dynamic cache partitioning to explore co-optimization potential at run-time. Related DNN accelerator research [118] proposed co-optimization of hardware-based (e.g., bitwidth) and neural network parameters (e.g., L2 regularization). These works motivate consideration for system-wide, component-level ML application.

Existing system-level optimization schemes (e.g., [84, 87, 105]) consider configuration opportunities at just a single and very high level of abstraction (e.g., task allocation or big.LITTLE core configurations). Although these works may include low-level features such as NoC utilization and DRAM bandwidth in their ML models, they do not account for the impact of component-level optimization techniques such as NoC packet routing, cache prefetching, etc. We instead envision an ML-based system-wide, component-level framework for run-time optimization. In this framework, control decisions would involve a larger hierarchy of both component-level (or lower) features and control options as well as higher-level decisions, allowing a more comprehensive and precise perspective for run-time optimization.

Advance Automated Design: While fully automated design might be the ultimate objective, increasingly automated design is nevertheless an important milestone for future work. Specifically, as more tasks are automated, there is greater potential to enable a positive-feedback loop between machine learning and architecture, providing immense practical benefits for both fields. There are, of course, a number of intervening challenges that must be solved, each of which represents a substantial area for future work.

One challenge involves modeling the hierarchical structure of architectural components. This model would likely benefit from integrating pertinent characteristics across the system stack, from process technology to full-system behavior, thus generating a highly accurate representation for real-world systems. Another research direction could explore methods for machine learning models to identify potential design aspects for improvement. Ideally, this model could explore not just reconfiguration of pre-existing options (as in [119]), but also generate novel configuration options. Integrating these and potentially other capabilities may provide a framework to advance automated design.

3.4 Conclusion

Machine learning has rapidly become a powerful tool in architecture, with established applicability to design, optimization, simulation, and more. Notably, ML has already been successfully applied to many components, including the core, cache, NoC, and memory, with performance often surpassing prior state-of-the-art analytical, heuristic, and human-expert strategies. Widespread application is further facilitated by diverse training methods and learning models, allowing effective trade-offs between performance and overhead based on task requirements. These advancements are likely just the beginning of a revolutionary shift in architecture.

Optimization opportunities at the model level involving pruning and quantization offer broad benefits by enabling more practical implementation. Similarly, opportunities abound to extend existing work using ever-more-powerful ML models, enabling finer granularity, system-wide implementation. Finally, ML may be applied to entirely new aspects of architecture, learning hierarchical or abstract representations to characterize full system behavior based on both high and low level details. These extensive opportunities, along with yet to be envisioned possibilities, may eventually close the loop on highly (or even fully) automated architectural design.

A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study

Ting-Ru Lin*, Drew Penney*, Massoud Pedram, and Lizhong Chen

IEEE International Symposium on High Performance Computer Architecture (HPCA)
February 2020

* Equal contribution

Chapter 4: A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study

4.1 Introduction

Improvements in computational capabilities are increasingly reliant upon advancements in many-core chip designs. These designs emphasize parallel resource scaling and consequently introduce many considerations beyond those in single core processors. As a result, traditional design strategies may not scale efficiently with this increasing parallelism. Early machine learning approaches, such as simple regression and neural networks, have been proposed as an alternative design strategy. More recent machine learning developments leverage deep reinforcement learning to provide improved design space exploration. This capability is particularly promising in broad design spaces, such as network-on-chip (NoC) designs.

NoCs provide a basis for communication in many-core chips that is vital for system performance [120]. NoC design involves many trade-offs between latency, throughput, wiring resources, and other overhead. Exhaustive design space exploration, however, is often infeasible in NoCs and architecture in general due to immense design spaces. Thus, intelligent exploration approaches would greatly improve NoC designs.

Applications include recently proposed routerless NoCs [121, 122]. Conventional router-based NoCs incur significant power and area overhead due to complex router structures. Routerless NoCs eliminate these costly routers by effectively using wiring resources while achieving comparable scaling to router-based NoCs. Prior research has demonstrated up to 9.5x reduction in power and 7x reduction in area compared with mesh [122], establishing routerless NoCs as a promising alternative for NoC designs. Like many novel concepts and approaches in architecture, substantial ongoing research is needed to explore the full potential of the routerless NoC design paradigm and help advance the field. Design challenges for routerless NoCs include efficiently exploring the huge design space (easily exceeding 10^{12}) while ensuring connectivity and wiring resource constraints. This makes routerless NoCs an ideal case study for intelligent

design exploration.

Prior routerless NoC design has followed two approaches. The first, isolated multi-ring (IMR) [121], uses an evolutionary approach (genetic algorithm) for loop design based on random mutation/exploration. The second approach (REC) [122] recursively adds loops strictly based on the NoC size, severely restricting broad applicability. Briefly, neither approach guarantees efficient generation of fully-connected routerless NoC designs under various constraints.

In this paper, we propose a novel deep reinforcement learning framework for design space exploration, and demonstrate a specific implementation using routerless NoC design as our case study. Efficient design space exploration is realized using a Monte-Carlo tree search (MCTS) that generates training data to a deep neural network which, in turn, guides the search in MCTS. Together, the framework self-learns loop placement strategies obeying design constraints. Evaluation shows that the proposed deep reinforcement learning design (DRL) achieves a 3.25x increase in throughput, 1.6x reduction in packet latency, and 5x reduction in power compared with a conventional mesh. Compared with REC, the state-of-the-art routerless NoC, DRL achieves a 1.47x increase in throughput, 1.18x reduction in packet latency, 1.14x reduction in average hop count, and 6.3% lower power consumption. When scaling from a 4x4 to a 10x10 NoC under synthetic workloads, the throughput drop is also reduced dramatically from 31.6% in REC to only 4.7% in DRL.

Key contributions of this paper include:

- Fundamental issues are identified in applying deep reinforcement learning to routerless NoC designs;
- An innovative deep reinforcement learning framework is proposed and implementation is presented for routerless NoC design with various design constraints;
- Cycle-accurate architecture-level simulations and circuit-level implementation are conducted to evaluate the design in detail;
- Broad applicability of the proposed framework with several possible examples is discussed.

The rest of the paper is organized as follows: Section 4.2 provides background on NoC architecture and design space complexity; Section 4.3 describes issues in prior routerless NoC design approaches and the need for a better method; Section 4.4 details the

proposed deep reinforcement learning framework; Section 4.5 illustrates our evaluation methodology; Section 4.6 provides simulation results; and Section 4.7 concludes.

4.2 Background

4.2.1 NoC Architecture

Single-ring NoCs: Nodes in a single-ring NoC communicate using one ring connecting all nodes.[1] Packets are injected at a source node and forwarded along the ring to a destination node. An example single-ring NoC is seen in Figure 4.1(a). Single-ring designs are simple, but have low bandwidth, severely restricting their applicability in large-scale designs. Specifically, network saturation is rapidly reached as more nodes are added due to frequent end-to-end control packets [123]. Consequently, most single-ring designs only scale to a modest number of processors [124].

Router-based NoCs: NoC routers generally consist of input buffers, routing and arbitration logic, and a crossbar connecting input buffers to output links. These routers enable a decentralized communication system in which routers check resource availability before packets are sent between nodes [122]. Mesh (or mesh-based architectures) have become the *de facto* choice due to their scalability and relatively high bandwidth [121]. The basic design, shown in Figure 4.1(b), features a grid of nodes with a router at every node. These routers can incur 11% chip area overhead [125] and, depending upon frequency and activity, up to 28% chip power [126, 127] overhead, although some recent work [128, 129] has shown much smaller overhead using narrow links and shallow/few buffers with high latency cost; this indirectly shows that routers are the main cost in existing NoCs. Hierarchical-ring, illustrated in Figure 4.1(c), instead uses several local rings connected by the dotted global ring. Routers are only needed for nodes intersected by the global ring as they are responsible for packet transfer between ring groups [130]. Extensive research has explored router-based NoC optimization [126, 131, 132], but these solutions only slightly reduce power and area overhead [121].

Routerless NoCs: Significant overhead associated with router-based topologies has motivated routerless NoC designs. Early proposals [131] used bus-based networks in a hierarchical approach by dividing the chip into multiple segments, each with a local

[1]Note that rings and loops are used interchangeably in this paper.

broadcast bus. Segments are connected by a central bus with low-cost switching elements. These bus-based networks inevitably experience contention on local buses and at connections with the central bus, resulting in poor performance under heavy traffic. Recently, isolated multi-ring (IMR) NoCs have been proposed that exploit additional interconnect wiring resources in modern semiconductor processes [121]. Nodes are connected via at least one ring and packets are forwarded from source to destination without switching rings. IMR improves over mesh-based designs in terms of power, area, and latency, but requires significant buffer resources: each node has a dedicated input buffer for each ring passing through its interface, thus a single node may require many packet-sized buffers [121, 122]. Recent routerless NoC design (REC) [122] has mostly eliminated these costly buffers by adopting shared packet-size buffers among loops. REC uses just a single flit-sized buffer for each loop, along with several shared extension buffers to provide effectively the same functionality as dedicated buffers [122].

Both IMR and REC designs differ from prior approaches in that no routing is performed during traversal, so packets in one loop cannot be forwarded to another loop [121, 122]. Both designs must therefore satisfy two requirements: every pair of nodes must be connected by at least one loop and all routing must be done at the source node. Figure 4.2 delineates these requirements and highlights differences between router-based and routerless NoC designs. Figure 4.2(a) depicts an incomplete 4x4 ring-based NoC with three loops. These loops are unidirectional so arrows indicate the direction of packet transfer for each ring. Node F is isolated and cannot communicate with other nodes since no ring passes through its interface. Figure 4.2(b) depicts the NoC with an additional loop through node F. If routers are used, such as at node A, this ring would complete the NoC, as all nodes can communicate with ring switching. Packets from node K, for example, can be transferred to node P using path 3, which combines $paths1$ and $path2$. In a routerless design, however, there are still many nodes that cannot communicate as packets must travel along a *single* ring from source to destination. That is, packets from node K cannot communicate with node P because $path1$ and $path2$ are isolated from each other. Figure 4.2(c) depicts an example 4x4 REC routerless NoC[122]. Loop placement for larger networks is increasingly challenging.

Routerless NoCs can be built with simple hardware interfaces by eliminating crossbars and VC allocation logic. As a result, current state-of-the-art routerless NoCs have achieved 9.5x power reduction, 7.2x area reduction, and 2.5x reduction in zero-load

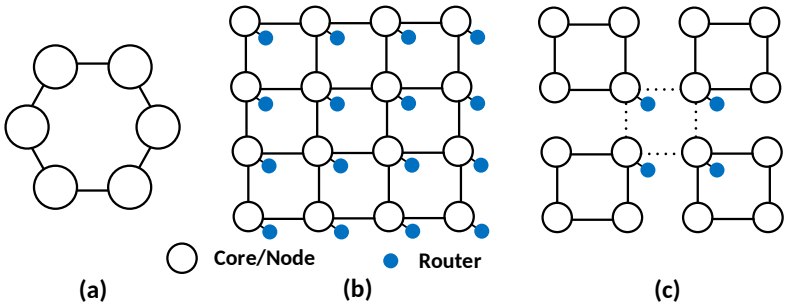


Figure 4.1: NoC Architecture. (a) Single-Ring (b) Mesh (c) Hierarchical Ring

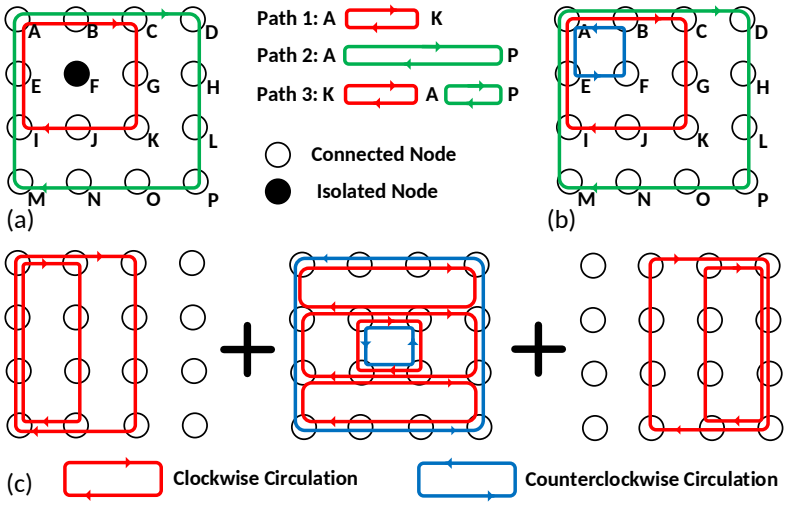


Figure 4.2: A 4x4 NoC with rings. (a) A NoC with one isolated node. (b) A NoC without isolated nodes. (c) A 4x4 routerless NoC with rings.

packet latency compared with conventional mesh topologies [122]. Packet latency, in particular, is greatly improved by single-cycle delays per hop, compared with standard mesh, which usually requires two cycles for the router alone. Hop count in routerless designs can asymptotically approach the optimal mesh hop count using additional loops at the cost of power and area. Wiring resources, however, are finite, meaning that one must restrict the total number of overlapping rings at each node (referred to as node overlapping) to maintain physical realizability. In Figure 4.2 (b), node overlapping at node A, for example, is three, whereas node overlapping at node F is one. Wiring resource restriction is one of the main reasons that make routerless NoC design substan-

tially more challenging. As discussed in Section 3, existing methods either do not satisfy or do not enforce these potential constraints. We therefore explore potential applications and advantages of machine learning.

4.2.2 Design Space Complexity

Design space complexity in routerless NoCs poses a significant challenge requiring efficient exploration. A small 4x4 NoC using 10 loops chosen from all 36 possible rectangular loops has $\binom{36}{10} \approx 10^8$ total designs. This design space increases rapidly with NoC size; an 8x8 NoC with 50 loops chosen from 784 possible rectangular loops has $\binom{784}{50} \approx 10^{79}$ designs. It can be shown that the complexity of routerless NoC designs exceeds the game of Go. Similar to AlphaGo, deep reinforcement learning is needed here and can address this complexity by approximating actions and their benefits, allowing search to focus on high-performing configurations.

4.3 Motivation

4.3.1 Design Space Exploration

Deep reinforcement learning provides a powerful foundation for design space exploration using continuously refined domain knowledge. This capability is advantageous since prior methods for routerless NoC designs have limited design space exploration capabilities. Specifically, the evolutionary approach [121] evaluates generations of individuals and offspring. Selection uses an objective function while evolution relies on random mutation, leading to an unreliable search since past experiences are ignored. Consequently, exploration can be misled and generate configurations with high average hop count and long loops (48 hops) in an 8x8 NoC [122]. The recursive layering approach (REC) overcomes these reliability problems but strictly limits design flexibility. Latency improves as the generated loops pass through fewer nodes on average[122], but hop count still suffers in comparison to router-based NoCs as it is restricted by the total number of loops. For an 8x8 NoC, the average hop count is 5.33 in mesh and 8.32 in the state-of-the-art recursive layering design, a 1.5x increase [122].

Both approaches are also limited by their inability to enforce design constraints, such

as node overlapping. In IMR, ring selection is based solely on inter-core-distance and ring lengths [121] so node overlapping may vary significantly based on random ring mutation. Constraints could be built into the fitness function, but these constraints are likely to be violated to achieve better performance. Alternatively, in REC, loop configuration for each network size is strictly defined. A 4x4 NoC must use exactly the loop structure shown in Figure 4.2 (c) so node overlapping cannot be changed without modifying the algorithm itself. These constraints must be considered during loop placement since an optimal design will approach these constraints to allow many paths for packet transfer.

4.3.2 Reinforcement Learning Challenges

Several challenges apply to deep reinforcement learning in any domain. To be more concrete, we discuss these considerations in the context of routerless NoC designs.

Specification of States and Action: State specification must include all information for the agent to determine optimal loop placement and should be compatible with DNN input/output structure. An agent that attempts to minimize average hop count, for example, needs information about the current hop count. Additionally, information quality can impact learning efficiency since inadequate information may require additional inference. Both state representation and action specification should be a constant size throughout the design process because the DNN structure is invariable.

Quantification of Returns: Return values heavily influence NoC performance so they need to encourage beneficial actions and discourage undesired actions. For example, returns favoring large loops will likely generate a NoC with large loops. Routerless NoCs, however, benefit from diverse loop sizes; large loops help ensure high connectivity while smaller loops may lower hop counts. It is difficult to achieve this balance since the NoC will remain incomplete (not fully connected) after most actions. Furthermore, an agent may violate design constraints if the return values do not appropriately deter these actions. Returns should be conservative to discourage useless or illegal loop additions.

Functions for Learning: Optimal loop configuration strategies are approximated by learned functions, but these functions are notoriously difficult to learn due to high data requirements. This phenomenon is observed in AlphaGo [13] where the policy function successfully chooses from 19^2 possible moves at each of several hundred steps, but requires more than 30 million data samples. An effective approach must consider

this difficulty, which can be potentially addressed with optimized data efficiency and parallelization across threads, as discussed later in our approach.

Guided Design Space Search: An ideal routerless NoC would maximize performance while minimizing loop count based on constraints. Similar hop count improvement can be achieved using either several loops or a single loop. Intuitively, the single loop is preferred to reduce NoC resources, especially under strict overlapping constraints. This implies benefits from ignoring/trimming exploration branches that add loops with suboptimal performance improvement.

4.4 Proposed Scheme

4.4.1 Overview

The proposed deep reinforcement learning framework is depicted in Figure 4.3. Framework execution begins by initializing the Monte Carlo Tree Search (MCTS) with an empty tree and a neural network without *a priori* training. The whole process consists of many exploration cycles. Each cycle begins with a blank design (e.g., a completely disconnected NoC). Actions are continuously taken to modify this design. The DNN (dashed "DNN" box) selects a good initial action, which directs the search to a particular region in the design space; several actions are taken by following MCTS (dashed "MCTS" box) in that region. The MCTS starts from the current design (a MCTS node), and tree traversal selects actions using either greedy exploration or an "optimal" action until a leaf (one of many explored designs) is reached. Additional actions can be taken, if necessary, to complete the design. Finally, an overall reward is calculated ("Evaluation Metrics") and combined with information on state, action, and value estimates to train the neural network and update the search tree (the dotted "Learning" lines). The exploration cycle repeats to optimize the design. Once the search completes, full system simulations are used to verify and evaluate the design. In the framework, the DNN generates coarse designs while MCTS efficiently refines these designs based on prior knowledge to continuously generate more optimal configurations. Unlike traditional supervised learning, the framework does not require a training dataset; instead, the DNN and MCTS gradually train themselves from past exploration cycles.

Framework execution in the specific case of routerless NoCs is as follows: each cycle

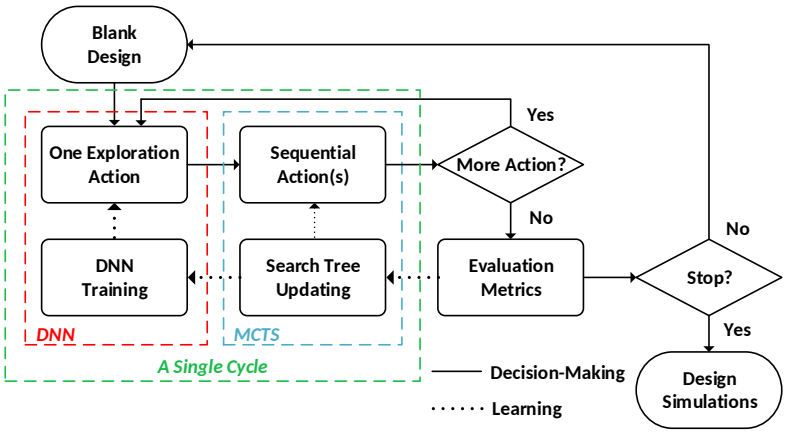


Figure 4.3: Deep reinforcement learning framework.

begins with a completely disconnected routerless NoC; the DNN suggests an initial loop addition; following this initial action, one or more loops are added ("Sequential Action") by the MCTS; rewards are provided for each added loop; the DNN and MCTS continuously add loops until no more loops can be added without violating constraints; the completed routerless NoC configuration is evaluated by comparing average hop count to that of mesh to generate a cumulative reward; overall rewards, along with information on state, action, and value estimates, are used to train the neural network and update the search tree; finally, these optimized routerless NoC configurations are tested.

The actions, rewards, and state representations in the proposed framework can be generalized for design space exploration in router-based NoCs and in other NoC-related research. Several generalized framework examples are discussed in Section 4.6.8. The remainder of this section addresses the application of the framework to routerless NoC design as a way to present low-level design and implementation details. Other routerless NoC implementation details including deadlock, livelock, and starvation are addressed in previous work [121, 122] so are omitted here.

4.4.2 Routerless NoCs Representation

Representation of Routerless NoCs (States): State representation in our framework uses a hop count matrix to encode current NoC state as shown in Figure 4.4. A 2x2 routerless NoC with a single clockwise loop is considered for simplicity. The overall

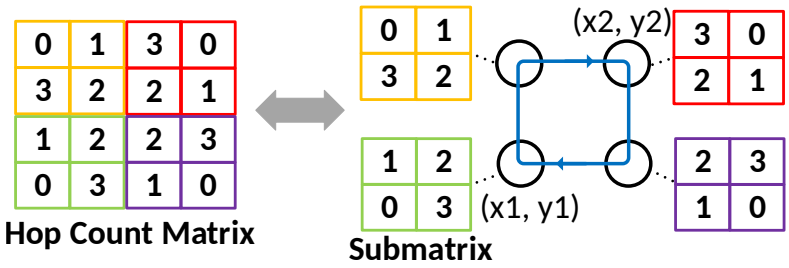


Figure 4.4: Hop count matrix of a 2x2 routerless NoC.

state representation is a 4x4 matrix composed of four 2x2 submatrices, each representing hop count from a specific node to every node in the network. For example, in the upper left submatrix, the zero in the upper left square corresponds to distance from the node to itself. Moving clockwise with loop direction, the next node is one hop away, then two, and three hops for nodes further along the loop. All other submatrices are generated using the same procedure. This hop count matrix encodes current loop placement information using a fixed size representation to accommodate fixed DNN layer sizes. In general, the input state for an $N \times N$ NoC is an $N^2 \times N^2$ hop count matrix. Connectivity is also implicitly represented in this hop count matrix by using a default value of $5 * N$ for unconnected nodes.

Representation of Loop Additions (Actions): Actions are defined as adding a loop to an $N \times N$ NoC. We restrict loops to rectangles to minimize the longest path. With this restriction, the longest path will be between diagonal nodes at the corners of the NoC, as in REC [122]. Actions are encoded as $(x1, y1, x2, y2, dir)$ where $x1, y1, x2$ and $y2$ represent coordinates for diagonal nodes $(x1, y1)$ and $(x2, y2)$ and dir indicates packet flow direction within a loop. Here, $dir = 1$ represents clockwise circulation for packets and $dir = 0$ represents counterclockwise circulation. For example, the loop in Figure 4.4 represents the action $(0, 0, 1, 1, 1)$. We enforce rectangular loops by checking that $x1 \neq x2$ and $y1 \neq y2$.

4.4.3 Returns After Loop Addition

The reward function encourages exploration by rewarding zero for all valid actions, while penalizing repetitive, invalid, or illegal actions using a negative reward. A repetitive

action refers to adding a duplicate loop, receiving a -1 penalty. An invalid action refers to adding a non-rectangular loop, receiving a -1 penalty. Finally, illegal actions involve additions that violate the node overlapping constraint, resulting in a severe $-5 * N$ penalty. The agent receives a final return to characterize overall performance by subtracting average hop count in the generated NoC from average mesh hop count. Minimal average hop count is therefore found by minimizing the magnitude of cumulative returns.

4.4.4 Deep Neural Network

Residual Neural Networks: Sufficient network depth is essential and, in fact, leading results have used at least ten DNN layers [13, 14, 133]. High network depth, however, can cause overfitting for many standard DNN topologies. Residual networks offer a solution by introducing additional shortcut connections between layers that allow robust learning even with network depths of 100 or more layers. A building block for residual networks is shown in Figure 4.5(a). Here, the input is X and the output, after two weight layers, is $F(X)$. Notice that both $F(X)$ and X (via the shortcut connection) are used as input to the activation function. This shortcut connection provides a reference for learning optimal weights and mitigates the vanishing gradient problem during back propagation [133]. Figure 4.5(b) depicts a residual box (Res) consisting of two convolutional (conv) layers. Here, the numbers 3x3 and 16 indicate a 3x3x16 convolution kernel.

DNN architecture: The proposed DNN uses the two-headed architecture shown in Figure 4.5(c), which learns both the policy function and the value function. This structure has been proven to reduce the amount of data required to learn the optimal policy function [14]. We use convolutional layers because loop placement analysis is similar to spatial analysis in image segmentation, which performs well on convolutional neural networks. Batch normalization is used after convolutional layers to normalize the value distribution and max pooling (denoted "pool") is used after specific layers to select the most significant features. Finally, both policy and value estimates are produced at the output as the two separate heads. The policy, discussed in section 4.4.2, has two parts: the four dimensions, $x1, y1, x2, y2$, which are generated by a softmax function following a ReLU and dir, which is generated separately using a tanh function. Tanh output between -1 and 1 is converted to a direction using $dir > 0$ as clockwise and $dir \leq 0$ as

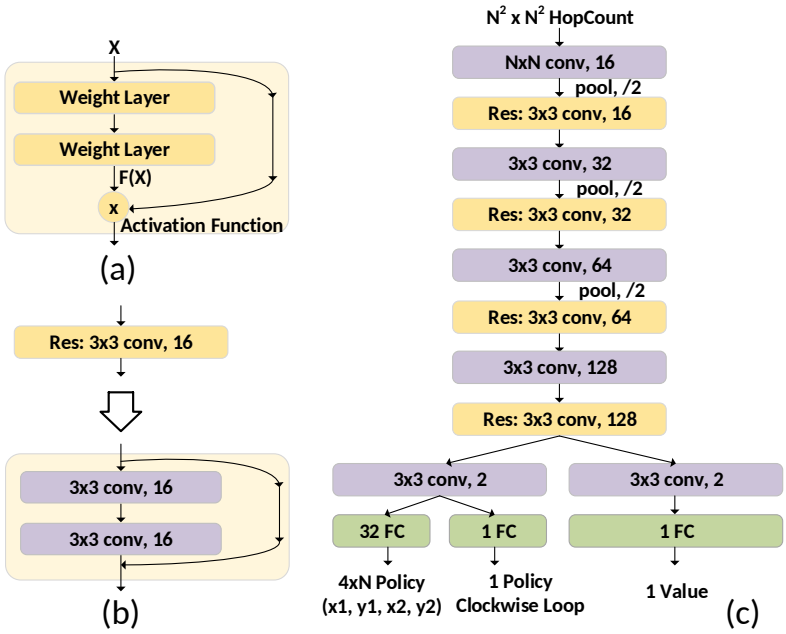


Figure 4.5: Deep residual networks. (a) A generic building block for residual networks. (b) A building block for convolutional residual networks. (c) Proposed network.

counterclockwise. Referring to Figure 4.5(c), the softmax input after ReLU is $\{a_{ij}\}$ where $i = 1, 2, 3, 4$ and $j = 1, ..., N$. Dimensions $x1$ and $y1$ are $max_j(exp(a_{1j})/\sum_j exp(a_{1j}))$ and $max_j(exp(a_{2j})/\sum_j exp(a_{2j}))$. The same idea applies to $x2$ and $y2$. The value head uses a single convolutional layer followed by a fully connected layer, without an activation function, to predict cumulative returns.

Gradients for DNN Training: In this subsection we derive parameter gradients for the proposed DNN architecture.[2] We define τ as the search process for a routerless NoC in which an agent receives a sequence of returns $\{r_t\}$ after taking actions $\{a_t\}$ from each state $\{s_t\}$. This process τ can be described a sequence of states, actions, and returns:

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, ...). \tag{4.1}$$

A given sequence of loops is added to the routerless NoC based on $\tau \sim p(\tau; \theta)$. We

[2]Although not essential for understanding the work, this subsection provides theoretical support and increases reproducibility.

can then write the expected cumulative returns for one sequence as

$$\mathbb{E}_{\tau\sim p(\tau;\theta)}[r(\tau)] = \int_{\tau} r(\tau)p(\tau;\theta)d\tau \tag{4.2}$$

$$p(\tau;\theta) = p(s_0)\prod_{t\geq 0}\pi(a_t;s_t,\theta)P(s_{t+1};s_t,a_t), \tag{4.3}$$

where $r(\tau)$ is a return and θ is DNN weights/parameters we want to optimize. Following the definition of π in section 2.2, $\pi(a_0;s_0,\theta)$ is the probability of taking action a_0 given state s_0 and parameter θ. We then differentiate the expected cumulative returns for parameter gradients

$$\nabla\mathbb{E}_{\tau\sim p(\tau;\theta)}[r(\tau)] = \nabla_\theta \int_{\tau} r(\tau)p(\tau;\theta)d\tau \tag{4.4}$$

$$= \int_{\tau}(r(\tau)\nabla_\theta \log p(\tau;\theta))p(\tau;\theta)d\tau \tag{4.5}$$

$$= \mathbb{E}_{\tau_\theta\sim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau;\theta)]. \tag{4.6}$$

Notice that transition probability $P(s_{t+1,r_t};s_t,a_t)$ is independent of θ so we can rewrite Equation 4.6 as

$$\mathbb{E}_{\tau_\theta\sim p(\tau;\theta)}[r(\tau)\nabla_\theta \log p(\tau;\theta)] \tag{4.7}$$

$$= \mathbb{E}_{\tau_\theta\sim p(\tau;\theta)}[r(\tau)\nabla_\theta \Sigma \log \pi(a_t;s_t,\theta)] \tag{4.8}$$

$$\approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi(a_t;s_t,\theta). \tag{4.9}$$

The gradient in equation 4.9 is proportional to raw returns (a constant value based on the *past* search trajectory). We therefore substitute $r(\tau)$ with A_t as

$$\nabla_\theta\mathbb{E}_{\tau\sim p(\tau;\theta)}[r(\tau)] \approx \sum_{t\geq 0} A_t\nabla_\theta \log \pi(a_t;s_t,\theta) \tag{4.10}$$

$$A_t = \sum_{t'>t} \gamma^{\,t'-t} r_{t'} - V(s_t;\theta), \tag{4.11}$$

where the first term in Equation 4.11 represents the returns from the $future$ trajectory at time t. We also subtract $V(s_t;\theta)$ to reduce the variance when replacing a constant

with a prediction. This approach is known as advantage actor-critic learning where the actor and the critic represent the policy function and value function, respectively [11]. In a two-headed DNN, θ consists of θ_π and θ_v for the policy function and the value function, respectively. Gradients for these two sets of parameters are directly obtained by representing Equation 4.10 as time intervals, rather than as a summation over time. These gradients are then given as

$$d\theta_\pi = (\sum_{t'>t} \gamma^{\,t'-t} r_{t'} - V(s_t;\theta_v))\nabla_{\theta_\pi} \log \pi(a_t; s_t, \theta_\pi) \tag{4.12}$$

$$d\theta_v = \nabla_{\theta_v} (\sum_{t'>t} \gamma^{\,t'-t} r_{t'} - V(s_t;\theta_v))^2. \tag{4.13}$$

The whole training procedure repeats the following equations

$$\theta_\pi = \theta_\pi + \gamma * d\theta_\pi \tag{4.14}$$

$$\theta_v = \theta_v + c * \gamma * \theta_v, \tag{4.15}$$

where γ is a learning rate and c is a constant.

4.4.5 Routerless NoC Design Exploration

An efficient approach for design space exploration is essential for routerless NoC design due to the immense design space. Deep reinforcement learning approaches are therefore well-suited for this challenge as they can leverage recorded states while learning. Some work uses experience replay, which guides actions using random samples. These random samples are useful throughout the entire learning process, so improve collected state efficiency [12], but break the correlation between states. Another approach is the Monte Carlo tree search (MCTS), which is more closely correlated to human learning behavior based on experience. MCTS stores previously seen routerless NoC configurations as nodes in a tree structure. Each node is then labeled with the expected returns for exploration starting from that node. As a result, MCTS can provide additional insight during state exploration and help narrow the scope of exploration to a few promising branches [13] to efficiently learn optimal loop placement.

In our implementation, each node s in the tree represents a previously seen routerless

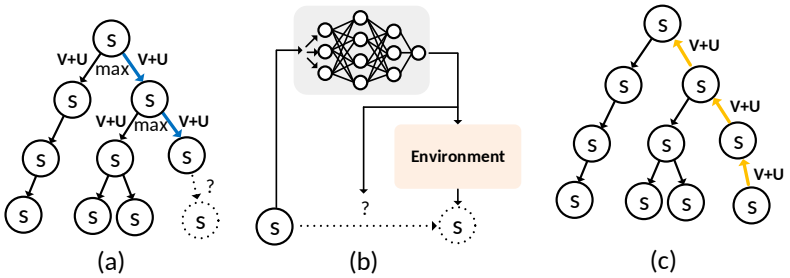


Figure 4.6: Monte Carlo tree search. (a) Search. (b) Expansion+evaluation using DNN. (c) Backup.

NoC and each edge represents an additional loop. Additionally, each node s stores a set of statistics: $\overline{V}(s_{next})$, $P(a_i; s)$, and $N(a_i; s)$. $\overline{V}(s_{next})$ is the mean cumulative return from s_{next} and is used to approximate the value function $V^{\pi}(s_{next})$. $P(a_i; s)$ is the prior probability of taking action a_i based on $\pi(a = a_i; s)$. Lastly, $N(a_i; s)$ is the visit count, representing the number of times a_i was selected at s. Exploration starts from state s, then selects the best action a^* based on expected exploration returns given by

$$a^* = arg\max_{a_i}(U(s, a_i) + \overline{V}(s_{next})) \tag{4.16}$$

$$U(s, a_i) = c * P(a_i; s)\frac{\sqrt{\sum_j N(a_j; s)}}{1 + N(a_i; s)}, \tag{4.17}$$

where $U(s, a_i)$ is the upper confidence bound and c is a constant [134]. The first term in Equation 4.16 encourages broad exploration while the second emphasizes fine-grained exploitation. At the start, $N(a_i; s)$ and $\overline{V}(s_{next})$ are similar for most routerless NoCs so exploration is guided by $P(a_i; s) = \pi(a = a_i; s)$. Reliance upon DNN policy decreases with time due to an increasing $N(a_i; s)$, which causes the search to asymptotically prefer actions/branches with high mean returns [14]. Search is augmented by an ϵ-greedy factor where the best action is ignored with probability ϵ to further balance exploration and exploitation.

There are three phases to the MCTS algorithm shown in Figure 4.6: search, expansion+evaluation, and backup. (1) Search: an agent selects the optimal action (loop placement) by either following Equation 4.16 with probability $1 - \epsilon$ or using a greedy search with probability ϵ. Algorithm 1 details the greedy search that evaluates the benefit

from adding various loops and selects the loop with the highest benefit. $CheckCount()$ returns the total number of nodes that can communicate after adding a loop with diagonal nodes at $(x1, y1)$ and $(x2, y2)$. Next, the $Imprv()$ function returns the preferred loop direction based on the average hop count improvement. The tree is traversed until reaching a leaf node (NoC configuration) without any children (further developed NoCs). (2) Expansion+evaluation: the leaf state is evaluated using the DNN to determine an action for rollout/expansion. Here, $\pi(a = a_i; s)$ is copied, then later used to update $P(a_i; s)$ in Equation 4.17. A new edge is then created between s and s_{next} where s_{next} represents the routerless NoC after adding the loop to s. (3) Backup: After the final cumulative returns are calculated, statistics for the traversed edges are propagated backwards through the tree. Specifically, $\overline{V}(s_{next})$, $P(a_i; s)$, and $N(s, a_i)$ are all updated.

Algorithm 1 Greedy Search

```
Initialization: bestLoop = [0, 0, 0, 0], bestCount = 0, bestImprv = 0, and dir = 0
for x1 = 1;+1;N do
    for y1 = 1:+1;N do
        for x2 = x1+1:+1;N do
            for y2 = y1+1:+1;N do
                count = CheckCount(x1, y1, x2, y2)
                if count > bestCount then
                    bestCount = count
                    bestLoop = [x1, y1, x2, y2]
                    bestImpv, dir = Imprv(x1, y1, x2, y2)
                else if return == bestCount then
                    imprv', dre' = Imprv(x1, y1, x2, y2)
                    if imprv' > bestImprv then
                        bestLoop = [x1, y1, x2, y2]
                        bestImprv = imprv'
                        dir = dir'
                    end if
                end if
            end for
        end for
    end for
end for
return bestRing, dir
```

4.4.6 Multi-threaded Learning

The framework incorporates a multi-threaded approach, in which many threads independently explore the design space while collaboratively updating global parameters [135]. Figure 4.7 depicts the proposed framework with multi-threaded exploration. At the start, thread 0 creates a parent DNN with initial weights/parameters θ, then creates many child threads (1 to n) that create their own child DNNs, each of which acts as an individual learning agent. The parent thread sends DNN parameters to child threads and receives parameter gradients from child threads. Convergence is stabilized by averaging both large gradients and small gradients during training [135]. The parent thread additionally maintains a search tree that records past child thread actions for each MCTS query. While not needed for correctness, the multi-threaded approach facilitates more efficient exploration as shown in evaluation and is very useful in practice.

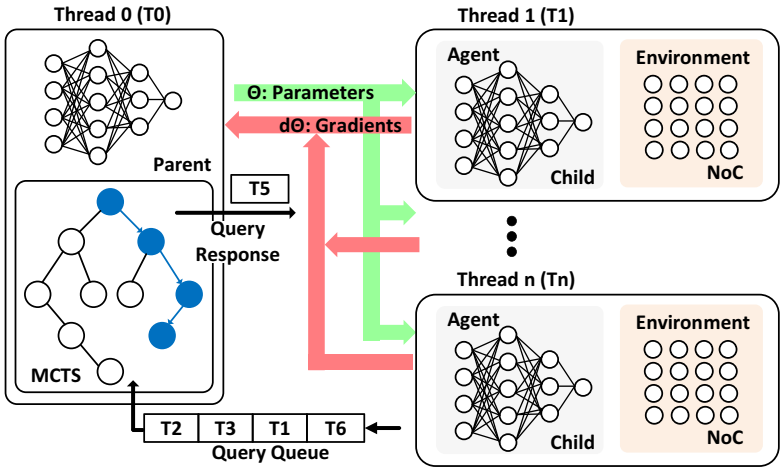


Figure 4.7: Multi-threaded framework.

4.5 Methodology

We evaluate the proposed deep reinforcement learning (DRL) routerless design against the previous state-of-the-art routerless design (REC) [122] and several mesh configurations. All simulations use Gem5 with Garnet2.0 for cycle-accurate simulation [136]. For synthetic workloads, we test uniform random, tornado, bit complement, bit rotation,

shuffle, and transpose traffic patterns. Performance statistics are collected for 100,000 cycles across a range of injection rates, starting from 0.005 flits/node/cycle and incremented by 0.005 flits/node/cycle until the network saturates. Results for PARSEC are collected after benchmarks are run to completion with either sim-large or sim-medium input sizes.[3] Power and area estimations are based on Verilog post-synthesis simulation, following a similar VLSI design flow as in REC that synthesizes the Verilog implementation in Synopsys Design Compiler and conducts place & route in Cadence Encounter under 15nm NanGate FreePDK15 Open Cell Library [137].

We regard node overlapping as a more appropriate measure than link overlapping (i.e., the number of links between adjacent nodes) for manufacturing constraints. REC can only generate NoCs with a single node overlapping value for a given NoC size, whereas DRL designs are possible with many values. Comparisons between REC and DRL therefore consider both equal overlapping (demonstrating improved loop placement for DRL) and unequal overlapping (demonstrating improved design capabilities for DRL).

For synthetic and PARSEC workloads, REC and DRL variants use identical configurations for all other parameters, matching prior testing [122] for comparable results. Results nevertheless differ slightly due to differences between Gem5 and Synfull [138], used in REC testing. In REC and DRL, each input link is attached to a flit-sized buffer with 128-bit link width. Packet injection and forwarding can each finish in a single cycle up to 4.3 GHz. For mesh simulations, we use a standard two-cycle router delay in our baseline (Mesh-2). We additionally test an optimized one-cycle delay router (Mesh-1) and, in PARSEC workloads, an "ideal" router with zero router delay (Mesh-0) leaving only link/contention delays. These mesh configurations use 256-bit links, 2 VCs per link, and 4-flit input buffer. 128-bit links were considered, but exhibited a sub-optimal trade-off between power/area and performance (so would not provide a strong comparison against DRL). Packets are categorized into control and data packets, with 8 bytes and 72 bytes, respectively. The number of flits per packet is then given as packet size divided by link width. Therefore, in REC and DRL simulations, control packets are 1 flit and data packets are 5 flits. Similarly, in mesh simulations, control packets are 1 flit while data packets are 3 flits. For PARSEC workloads, L1D and L1I caches are set to 32 KB with 4-way associativity and the L2 cache is set to 128 KB with 8-way associativity.

[3]Several workloads exhibit compatibility issues with our branch of Gem5, but we include all workloads that execute successfully.

Table 4.1: Hyperparameter Exploration

Epsilon (ϵ)	0.05	0.10	0.20	0.30
# Valid designs	25	27	11	2
Min Hop Count	5.59	5.60	5.61	5.53
SD for Hop Count	0.140	0.065	0.050	0.040

Link delay is set to one cycle per hop for all tests.

4.6 Results & Analysis

4.6.1 Design Space Exploration

Exploration starts without *a priori* experience or training data. Over time, as the search tree is constructed, the agent explores more useful loop configurations, which provide increased performance. Configurations satisfying design criteria can be found in seconds and minutes for 4x4 and 10x10 NoCs, respectively. Figure 4.8 illustrates a 4x4 DRL design. Different from REC [122], the generated topology replaces one inner loop with a larger loop and explores different loop directions. The resulting topology is completely symmetric and far more regular than IMR. We observe similar structure for larger topologies, but omit these due to space constraints.

Multi-threaded exploration efficacy is verified by comparing designs generated using either single or multi-threaded search. For a 10x10 NoC, after a 10 hour period, single-threaded search found 6 valid designs, whereas multi-threaded search found 49 valid designs. Moreover, multi-threaded search generates designs with 44% lower standard deviation (SD) for hop count (decreasing from 0.027 to 0.015). This demonstrates the benefits of multi-threaded search to efficiently achieve more consistent results.

We further evaluate changes in the hyperparameter ϵ, which balances search exploration and exploitation. Results after a five hour period using 8x8 NoCs are summarized in Table 4.1. High values for ϵ can quickly generate more optimal configurations, but may frequently explore invalid actions and thus suffer under strict constraints. We therefore select the best value for ϵ in subsequent evaluations based on the time allocated to exploration as well as the rigor of constraints. In most cases, $\epsilon = 0.1$ generates high-performing designs given adequate time.

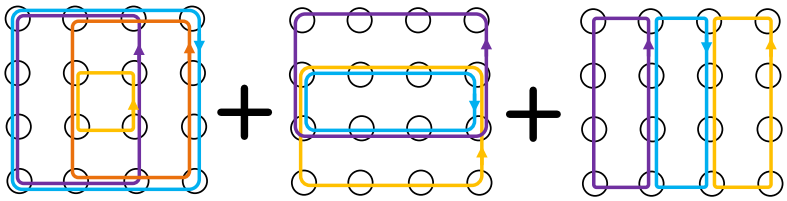


Figure 4.8: A 4x4 NoC topology generated by DRL.

4.6.2 Framework Capabilities

The proposed DRL framework can automatically generate NoC designs under various constraints so can be adapted to available design resources for any NoC size. In contrast, REC generates only a single design for each NoC size and, consequently, cannot be adapted to design goals, thus severely restricting real-world applicability. In the following, we exemplify the broad design capabilities of the DRL framework, none of which are possible with REC.

Generate feasible designs for larger NoCs: REC design does not work if node overlapping is less than $2 * (N - 1)$. Conversely, the proposed DRL framework can generate NoCs with smaller node overlapping across many sizes. For example, with a fixed node overlapping of 18, REC cannot generate NoCs larger than 10x10. Our DRL framework, however, has successfully generated configurations for 12x12, 14x14, 16x16 and 18x18 routerless NoCs. Note that 18x18 is the theoretical max routerless NoC size that can be fully connected with a node overlapping of 18. In a 20x20 NoC, there must be *at least* 19 rectangular loops passing through the bottom left node to connect to all other columns. As summarized in Table 4.2, the average hop count of DRL designs is still close to N, even when N approaches the node overlapping limit, showing the effectiveness of the DRL framework.

Utilize additional wiring resources: DRL is able to exploit additional wiring resources, when available, to improve performance, whereas REC cannot use any wires beyond $2 * (N - 1)$. Table 4.3 and Table 4.4 illustrate the hop count advantage of DRL over REC with various node overlappings. For example, a 10x10 DRL NoC with a node overlapping of 20 achieves a 20.4% reduction in hop count compared with the only possible REC 10x10 NoC.

Facilitate routerless NoC implementation in industry: Routerless NoCs offer

Table 4.2: DRL supports larger NoCs with 18 overlapping.

NoC Size	10x10	12x12	14x14	16x16	18x18
REC Hop Count	9.64	N/A	N/A	N/A	N/A
DRL Hop Count	7.94	12.25	15.11	18.03	21.01

Table 4.3: DRL utilizes additional wiring resources; 8x8.

Topology	REC	DRL	DRL	DRL	DRL
Node overlapping	14	14	16	18	20
Hop count	7.33	6.22	5.94	5.82	5.80
Improve over REC	N/A	15.14%	18.96%	20.60%	20.87%

Table 4.4: DRL utilizes additional wiring resources; 10x10.

Topology	REC	DRL	DRL	DRL	DRL
Node overlapping	18	18	20	22	24
Hop count	9.64	7.94	7.67	7.59	7.55
Improve over REC	N/A	17.64%	20.44%	21.27%	21.68%

a promising approach to achieve multi-fold savings in hardware cost compared with router-based NoCs, but the strict wiring requirements in the previous REC designs may hinder adoption in industry. The proposed DRL framework provides high flexibility to explore many combinations of NoC sizes and constraints that are not possible with REC. This flexibility can greatly aid future NoC research and implementation in industry by adapting to other constraints, such as maximum loop length or maximum hop count, which can also be integrated into the reward function.

4.6.3 Synthetic Workloads

Performance evaluations in this and next subsections use a node overlapping constraint of $2 * (N - 1)$ for both REC and DRL because that is the *only* possible constraint for REC. Alternative DRL configurations, such as those shown in Tables 4.2 to 4.4, can nevertheless provide additional benefits while satisfying various design goals.

Packet Latency: Figure 4.9 plots the average packet latency of four synthetic workloads for a 10x10 NoC. Tornado and shuffle are not shown as their trends are similar to bit rotation. Zero-load packet latency for DRL is the lowest in all workloads. For example, with uniform random traffic, zero-load packet latency is 9.89, 11.67, 19.24, and

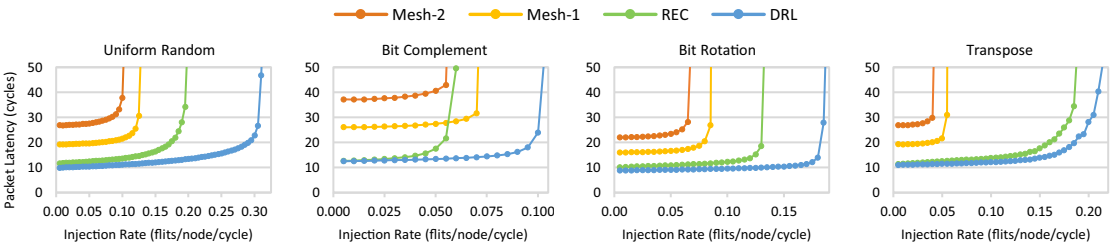


Figure 4.9: Average packet latency for synthetic workloads in 10x10 NoC.

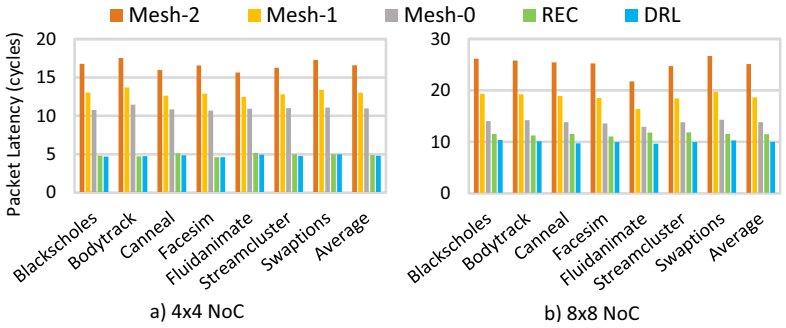


Figure 4.10: Packet latency for PARSEC workloads.

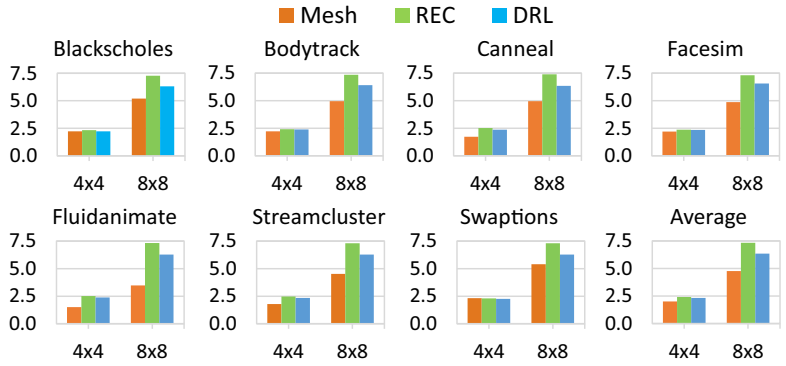


Figure 4.11: Average hop count for PARSEC workloads.

26.85 cycles for DRL, REC, Mesh-1, and Mesh-2, respectively, corresponding to a 15.2%, 48.6%, and 63.2% latency reduction by DRL. Across all workloads, DRL reduces zero-load packet latency by 1.07x, 1.48x and 1.62x compared with REC, Mesh-1, and Mesh-2, respectively. This improvement for both REC and DRL over mesh configurations results from reduced per hop latency (one cycle). DRL improves over REC due to additional

connectivity and better loop placement. For example, in a 10x10 NoC, DRL provides four additional paths.

Throughput: DRL provides substantial throughput improvements for all traffic patterns. For uniform traffic, throughput is approximately 0.1, 0.125, 0.195, and 0.305 for Mesh-2, Mesh-1, REC, and DRL, respectively. Notably, in transpose, DRL improves throughput by 208.3% and 146.7% compared with Mesh-2 and Mesh-1. Even in bit complement where mesh configurations perform similarly to REC, DRL still provides a 42.8% improvement over Mesh-1. Overall, DRL improves throughput by 3.25x, 2.51x, and 1.47x compared with Mesh-2, Mesh-1, and REC, respectively. Again, additional loops with greater connectivity in DRL allow a greater throughput compared with REC. Furthermore, improved path diversity provided by these additional loops allows much higher throughput compared with mesh configurations.

4.6.4 PARSEC Workloads

We compare real-world performance of REC, DRL, and three mesh configurations for 4x4 and 8x8 NoCs on a set of PARSEC benchmarks. We generate Mesh-0 results by artificially reducing packet latency by the hop count for every recorded flit since such a configuration is difficult to simulate otherwise. As a result, performance is slightly worse than an "ideal" zero-cycle-router mesh.

Packet Latency: As shown in Figure 4.10, for the 4x4 network, variations in loop configuration are relatively small, being heavily influenced by full-connectivity requirements. Nevertheless, in the 4x4 NoC, DRL improves performance over REC in all but two applications where performance is similar. For example, DRL reduces packet latency by 4.7% in fluidanimate compared with REC. Improvements over mesh configurations for fluidanimate are greater with a 68.5%, 60.4%, and 54.9% improvement compared with Mesh-2, Mesh-1, and Mesh-0. On average, DRL reduces packet latency by 70.7%, 62.8%, 56.1%, and 2.6% compared with Mesh-2, Mesh-1, Mesh-0, and REC, respectively.

DRL improvements are more substantial in 8x8 NoCs as DRL can explore a larger design space. For example, in fluidanimate, average packet latency is 21.7, 16.4, 12.9, 11.8, and 9.7 in Mesh-2, Mesh-1, Mesh-0, REC, and DRL, respectively. This corresponds to a 55.6%, 41.0%, 25.3%, and 18.2% improvement for DRL compared with Mesh-2, Mesh-1, Mesh-0, and REC. On average, DRL reduces packet latency by 60.0%, 46.2%,

Table 4.5: 8x8 PARSEC workload execution time (ms)

Workload	NoC Type			
	Mesh-2	Mesh-1	REC	DRL
Blackscholes	4.4	4.2	4.0	4.0
Bodytrack	5.4	5.3	5.1	5.1
Canneal	7.1	6.4	6.1	6.0
Facesim	626.0	587.0	515.2	512.3
Fluidanimate	35.3	29.2	25.2	24.4
Streamcluster	11.0	11.0	11.0	11.0

27.7%, and 13.5% compared with Mesh-2, Mesh-1, Mesh-0, and REC, respectively.

Hop Count: Figure 4.11 compares the average hop count for REC, DRL, and Mesh-2 for 4x4 and 8x8 NoCs. Only Mesh-2 is considered as differences in hop count are negligible between mesh configurations (they mainly differ in per-hop delay). For 4x4 networks, REC and DRL loop configurations are relatively similar so improvements are limited, but DRL still provides some improvement in all workloads compared with REC. In streamcluster, average hop count is 1.79, 2.48, and 2.34 for mesh, REC, and DRL, respectively. On average, DRL hop count is 22.4% higher than mesh and 3.8% less than REC. For larger network sizes, we again observe the benefit from increased flexibility in loop configuration that DRL exploits. This optimization allows more loops to be generated, decreasing average hop count compared with REC by a minimum of 12.7% for bodytrack and a maximum of 14.3% in fluidanimate. On average, hop count for DRL is 13.7% less than REC and 35.7% higher than mesh.

Execution Time: Execution times for 8x8 PARSEC workloads are given in Table 4.5. Reductions in hop count and packet latency may not necessarily translate to reduced execution time as applications may be insensitive to NoC performance (notably streamcluster). Nevertheless, in fluidanimate, a NoC sensitive workload, DRL reduces execution time by 30.7%, 16.4%, and 3.17% compared with Mesh-2, Mesh-1, and REC, respectively. Overall, DRL provides the smallest execution time for every workload. Note that NoC traffic for PARSEC workloads is known to be light, so the significant throughput advantage of DRL over mesh and REC (Figure 4.9) is not fully reflected here. Additionally, as mentioned earlier, this evaluation restricts DRL to use the only overlapping value that works for REC. Larger benefits can be achieved with other DRL configurations, as shown next.

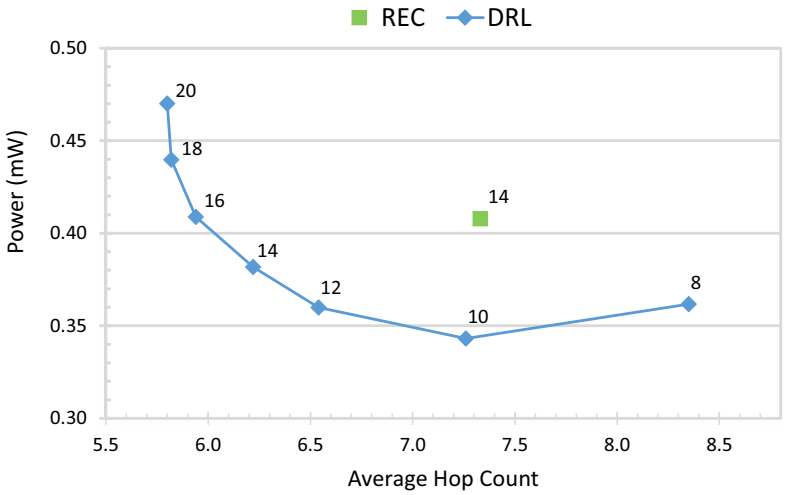


Figure 4.12: Power-performance tradeoffs for 8x8 (labels on the data points are node overlapping caps).

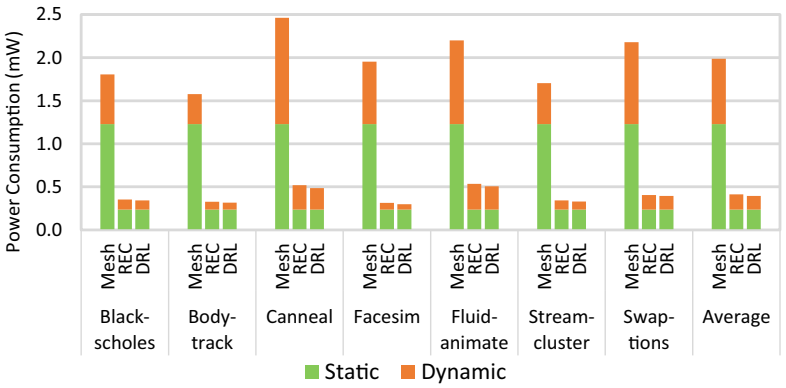


Figure 4.13: Power consumption for PARSEC workloads.

4.6.5 Power

The proposed DRL framework can generate diverse NoCs based on different objectives. Figure 4.12 demonstrates this capability as a tradeoff between power and performance (average hop count) for 8x8 NoCs. Each point represents one possible design and is labeled with the allowed node overlapping; REC therefore represents just a single design point. DRL with a node overlapping of 10 exhibits 1% lower hop count than REC while reducing power consumption by 15.9% due to reduced hardware complexity. Addition-

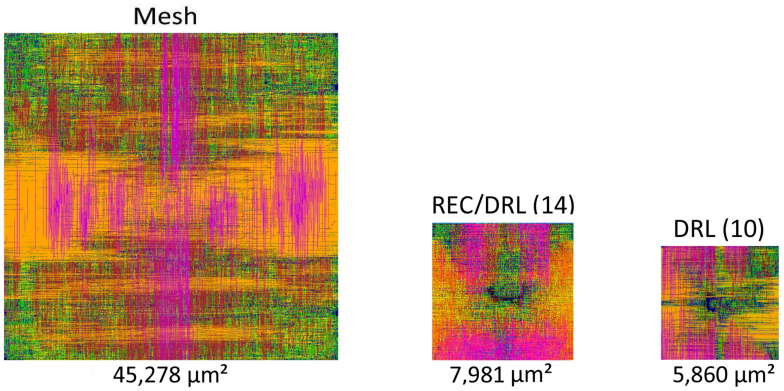


Figure 4.14: Area comparison (after P&R).

ally, DRL with a node overlapping of 16 reduces the average hop count by 18.9% with nearly equal power consumption (within 0.2%) due to more efficient loop placement. Overall, DRL is more flexible and efficient than the fixed REC scheme.

We additionally compare the power consumption of mesh (Mesh-2), REC, and DRL (both with the same node overlapping of 14) across PARSEC workloads. Results are generated after place & route in Cadence Encounter under 15nm technology node [137]. Global activity factor is estimated from link utilization statistics in Gem5 simulations. A clock frequency of 2.0 GHz is used, comparable to commercial many-core processors. As seen in Figure 4.13, static power for REC and DRL is 0.23mW, considerably lower than the 1.23mW of mesh. Dynamic power is the lowest for DRL due to improved loop configuration, leading to lower hop count and therefore lower dynamic power than REC in all workloads. DRL also provides significant savings over mesh due to reduced routing logic and fewer buffers. On average, dynamic power for DRL is 80.8% and 11.7% less than mesh and REC, respectively.

4.6.6 Area

Node area in routerless NoCs is determined by the node overlapping cap. In practice, to reduce design and verification effort, the same node interface can be reused if the node overlapping cap is the same. Figure 4.14 therefore compares the node area for 8x8 mesh (Mesh-2), REC/DRL with an overlapping of 14 (equal area due to equal overlapping),

and DRL with an overlapping of 10. DRL (10) is selected for comparison here because it has very similar hop count to REC, as shown in Figure 4.12. As can be seen, DRL (10) has the smallest area at 5,860 μm^2 due to an efficient design, while providing equivalent performance to REC. Both REC and DRL with an overlapping of 14 have a slightly increased area at 7,981 μm^2. Finally, mesh area is much higher at 45,278 μm^2. This difference is mainly attributed to routerless NoCs eliminating both crossbars and virtual channels. Note that results for REC and DRL already include the small look-up table at source. This table is needed to identify which loop to use for each destination (if multiple loops are connected), but each entry has only a few bits [122]. Area for the table and related circuitry is 443 μm^2, equivalent to only 0.9% of the mesh router (power is 0.028mW or 1.13% of mesh). We also evaluated the additional repeaters necessary to support DRL. Total repeater area is 0.159 mm^2 for DRL (14), so overhead compared with REC represents just 1.1% of mesh.

4.6.7 Discussion

Comparison with IMR: Evaluation by Alazemi et al. [122] showed that REC is superior to IMR in all aspects. In synthetic testing, REC achieves an average 1.25x reduction in zero-load packet latency and a 1.61x improvement in throughput over IMR. Similarly, in real benchmarks, REC achieves a 41.2% reduction in average latency. Both static and dynamic power are also significantly lower in REC due to reduced buffer requirements and more efficient wire utilization. Finally, REC area is just 6,083 μm^2 while IMR area is 20,930 μm^2, corresponding to a 2.4x increase. Comparisons between REC and DRL were therefore the primary focus in previous subsections since REC better represents the current state-of-the-art in routerless NoCs. The large gap between IMR and REC also illustrates that traditional design space search (e.g., genetic algorithm in IMR) is far from sufficient, which calls for more intelligent search strategies.

Reliability: Reliability concerns for routerless NoC stem from the limited path diversity since wiring constraints restrict the total number of loops. For a given node overlapping, DRL designs provide more loops and thus more paths between nodes as more nodes approach the node overlapping cap. In the 8x8 NoC, there are, on average, 2.77 paths between any two nodes in REC. This increases to 3.79 paths, on average, between any two nodes in DRL (using equal overlapping). DRL can therefore tolerate

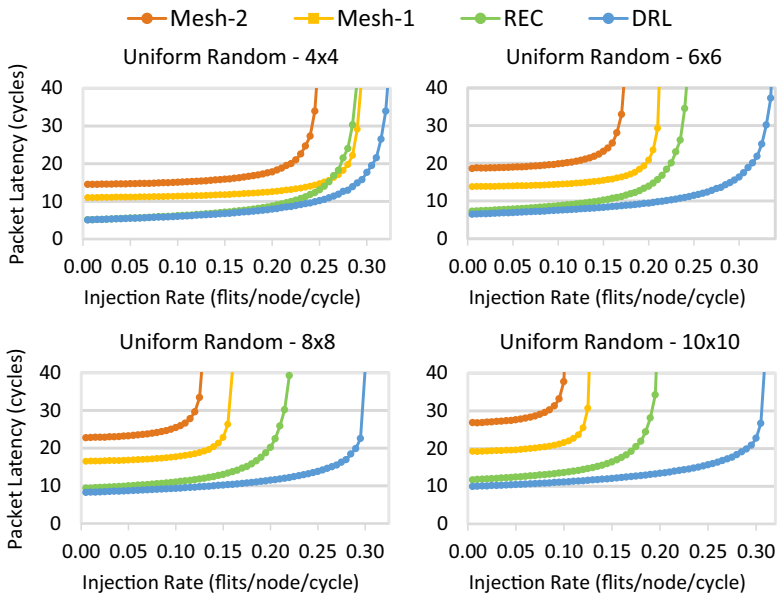


Figure 4.15: Synthetic Scaling for NoC Configurations.

more link failures before the NoC fails.

Scalability: DRL scales well compared with both REC and mesh configurations. For PARSEC workloads, shown in Figure 4.10, DRL exhibits 2.6% lower packet latency than REC for a 4x4 NoC, improving to a 13.5% reduction for an 8x8 NoC. Average hop count, shown in Figure 4.11, exhibits a similar trend. DRL improves average hop count by 3.8% in a 4x4 NoC and 13.7% in an 8x8 NoC. Scaling improvements are more evident in synthetic workloads. Figure 4.15, for example, shows scaling results for 4x4 to 10x10 NoC sizes with uniform random workloads. Note that the same axis values are used for all NoC sizes to emphasize scaling performance. Whereas REC throughput decreases from 0.285 flits/node/cycle to 0.195 flits/node/cycle, corresponding to a 31.6% decrease, the throughput for DRL only changes slightly from 0.32 to 0.305 flits/node/cycle, corresponding to a 4.7% reduction. This shows a significant improvement in scalability from REC to DRL. Increasing the NoC size also allows more flexibility in loop exploration, and thus more effective use of wiring resources for a given node overlapping constraint. Additionally, loop design for $N \times M$ NoCs using DRL is straightforward to implement, only requiring modifications to the DNN for dimension sizes.

4.6.8 Broad Applicability

Routerless NoC design represents just one possible application for the framework presented in this paper. This framework, with modifications to state/action representations, could also be applied to router-based NoC designs. Specifically, one related application is in 3-D NoCs where higher dimensionality encourages novel design techniques. Prior work has explored small-world router-based designs [64, 139] using a relatively limited learning-based approach. The design space exploration would be more effective using our framework. Specifically, state representation using hop count remains compatible with the current DNN structure by concatenating matrices for each 2D layer. Actions can involve adding links between nodes in the same layer (intra-layer links) or different layers (inter-layer links). One DNN can be used for each action type to achieve an efficient deep reinforcement learning process with a smaller design space. A significant advantage of our framework is that strict constraints can be enforced on link addition, such as 3-D distance, to meet timing/manufacturing capabilities.

The proposed framework can also be generalized for other NoC-related research problems. While detailed exploration is beyond the scope of this paper, we briefly mention a few promising examples that can benefit from our framework. Future work may exploit underutilized wiring resources in silicon interposers [140, 141] and explore better ways to connect CPU cores and stacked memories. The framework could similarly be used to improve the latency and throughput of chiplet networks [142, 143] by exploring novel interconnects structures that are non-intuitive and hard for human to conceive. NoCs for domain-specific accelerators (e.g., TPU [144], Eyeriss [145], and others) are another possible application. Due to their data-intensive nature, accelerators can benefit from high-performance [146] and possibly reconfigurable [147] NoCs, where the framework can explore connectivity among processing elements (PEs) and between PEs and memory.

4.7 Conclusion

Design space exploration using deep reinforcement learning promises broad application to architectural design. Current routerless NoC designs, in particular, have been limited by their ability to search design space, making routerless NoCs an ideal case study to demonstrate our innovative framework. The proposed framework integrates deep

learning and Monte Carlo search tree with multi-threaded learning to efficiently explore large design space under constraints. Full system simulations shows that, compared with state-of-the-art routerless NoC, our proposed deep reinforcement learning NoC can achieve a 1.47x increase in throughput, 1.18X reduction in packet latency, 1.14x reduction in average hop count, and 6.3% lower power consumption. The proposed framework has broad applicability to diverse NoC design problems and enables intelligent design space exploration in future work.

Intelligent Resource Optimization for Edge Networks Using Machine Learning

Drew Penney, Bin Li, and Lizhong Chen

Not yet published

Chapter 5: Intelligent Resource Optimization for Edge Networks Using Machine Learning

5.1 Introduction

High-priority (HP) latency-sensitive applications at the network edge, such as packet processing and machine learning inference, typically require resource provisioning for the peak load. This practice ensures satisfactory performance at all times, but results in over-provisioning during periods of low load. Service providers (SPs) have sought to ameliorate these adverse effects by opportunistically co-scheduling best effort (BE) applications, thereby enabling higher average resource utilization and, subsequently, reducing total cost of ownership (TCO)[1]. Naive co-scheduling can, however, introduce resource contention between HP and BE applications that may compromise QoS (e.g., drop rate or throughput) and lead to service level agreement violations.

Prior work has demonstrated practical co-scheduling by strictly managing the resources for each application, thus limiting contention. One approach involves static allocation, in which case the HP application is granted sufficient resources to handle peak load, while remaining resources are given to the BE application. In practice, this scheme proves highly inefficient since average load can deviate significantly from peak load. Further improvement is possible through extensive profiling, thereby enabling allocation that follows historic load curves on a time-of-day basis. Nevertheless, transient yet significant deviations from historic load averages can still compromise QoS guarantees. Overall, these static allocation methods leave much to be desired.

More recent work has explored dynamic resource allocation as a more effective and adaptable solution. Dynamic resource allocation requires dynamic feedback, usually in the form of QoS measurements (e.g., request latency), to find acceptable resource configurations and to determine when re-allocation is necessary due to changing workload demands. In particular, this feedback is assumed to be both highly accurate and fre-

[1]Industry practice observed across the spectrum of providers, including clouds running Google Gmail, Microsoft Bing, and AliCloud AliExpress.

quently updated, with many works assuming a one second sampling period. We find, however, that this one second interval is extremely inadequate for some operating environments, especially in packet processing workloads where natural fluctuations in resource demands can introduce an order of magnitude variations in QoS metrics. The search-based allocation methods adopted by most prior works further exacerbate this problem since they can require 20-40 samples [148], resulting in unacceptably long periods of QoS violations. Moreover, these works must undergo an extended re-sampling period every time the workload changes. This fundamental reliance upon QoS samples, and consequently *reactive* approach to dynamic resource allocation, necessitates a more intelligent resource allocation strategy that can predict appropriate configurations and guarantee high utilization without sacrificing QoS. Specifically, we investigate machine learning as a tool for dynamic resource allocation to provide *proactive* QoS guarantees.

Here, we propose a novel machine learning framework for dynamic resource allocation. Proactive QoS predictions are provided by several machine learning predictors, each of which is trained on a variety of best-effort workloads to accurately predict the effects of diverse resource contention behaviors. These QoS predictions, along with additional workload performance indicators, are passed to a reinforcement learning model that intelligently allocates resources to meet QoS goals. Together, these components enable rapid resource adjustments that safely exploit periods of low workload demands while minimizing QoS violations.

5.2 Practical Co-scheduling Opportunities

5.2.1 Resource Management Options

In general, prior works are limited in scope and allocate just one or two resources, most commonly the number of cores and core frequency. Average resource utilization can improve dramatically [149, 150] compared with static allocation, yet remains far from optimal since co-scheduling may benefit from dynamic allocation of resources that were not considered. As an example, consider a data-intensive BE application. If the only resource management options are core allocation and core frequency, then it may be difficult (or even impossible) to adequately control memory bandwidth contention. Furthermore, prior work primarily adopts search-based methods that require many samples

(20-40), so stop searching once an "acceptable," not necessarily optimal, allocation is found. These methods can also fail to identify acceptable allocations when configuration complexity increases.

Increasing adoption of workload co-scheduling in industry applications has led to the development of several powerful, new tools for fine-grained resource management. Intel® Resource Director Technology (RDT) is an exemplar that enables monitoring and management of cache and memory bandwidth on a per-thread basis. Fine-grained management of these resources, along with conventional resources (e.g., core allocation and core frequency), provides a comprehensive platform for workload co-scheduling, even with workloads that heavily contend for resources.

5.2.2 Workload Characterization

We begin by characterizing the resource scaling and contention behavior for a number of workloads. In particular, our selected HP workload is the virtual Broadband Network Gateway (vBNG), which represents a critical packet-processing workload at the network edge. We test a variety of BE workloads from several standard benchmarks (PARSEC and SPEC CPU) along with several broadly representative workloads (machine learning and java server). Platform and workload configuration is provided in Table 5.2. For the vBNG workload, performance is measured in terms of dropped packets per second. For the purposes of this discussion, we consider an acceptable fraction of dropped traffic to be on the order of 10^{-5}. For all other workloads, performance is measured as either execution time or a workload-specific metric.

First, we consider the isolated performance of the HP workload and observe its behavior as we scale cache and memory bandwidth resources. The results are shown in Figure 5.1-5.3. At high loads (i.e., high traffic injection rates), packet loss can vary significantly with resource allocation. Specifically, at an injection rate of 150 Gbps, the fraction of dropped traffic ranges from 10^{-1} to just higher than than 10^{-5}. At a lower load (e.g., an injection rate of 75 Gbps), a packet loss ratio of 10^{-6} can be achieved with practically all resource configurations. Overall, until injection rate exceeds 140-150 Gbps, the ratio of dropped traffic can be made acceptable with proper resource management, thus motivating workload co-scheduling.

		Memory Bandwidth Allocation							
		10	20	30	40	50	60	90	100
Cache	2	4962953	252528	13859	3873	4525	3321	3236	3037
Allocation	3	4818207	93710	9196	1730	1511	1347	1376	1407
	4	4668729	60662	5771	958	1051	794	763	804
	5	4555254	50661	4738	551	692	588	601	591
	6	4477018	42260	2950	570	488	483	470	496
	7	4472753	39026	2642	474	440	401	384	454
	8	4459026	35471	2221	486	412	518	426	425
	9	4281783	33879	2525	472	567	459	457	444

Figure 5.1: Isolated vBNG performance (packet drop rate) at 150 Gbps injection rate.

		Memory Bandwidth Allocation							
		10	20	30	40	50	60	90	100
Cache	2	23834	243	71	52	57	55	36	66
Allocation	3	15174	249	65	73	39	39	35	37
	4	16891	216	68	53	65	51	39	53
	5	13866	188	50	54	46	48	47	45
	6	13646	186	79	84	36	44	47	40
	7	13777	216	181	71	45	28	66	57
	8	13284	195	116	57	69	45	77	32
	9	13588	284	102	44	48	57	46	48

Figure 5.2: Isolated vBNG performance (packet drop rate) at 113 Gbps injection rate.

		Memory Bandwidth Allocation							
		10	20	30	40	50	60	90	100
Cache	2	90	17	1	1	7	1	3	9
Allocation	3	181	13	4	5	0	0	1	0
	4	86	12	4	5	1	7	10	3
	5	83	10	0	1	3	4	3	6
	6	80	14	3	4	11	3	2	3
	7	102	19	4	2	12	5	5	6
	8	102	15	2	10	5	4	1	0
	9	121	16	5	4	0	4	3	4

Figure 5.3: Isolated vBNG performance (packet drop rate) at 75 Gbps injection rate.

		Memory Bandwidth Allocation							
		10	20	30	40	50	60	90	100
Cache	1	15.96	11.53	9.35	7.62	7.46	7.39	7.31	7.30
Allocation	2	15.92	11.54	9.40	7.65	7.49	7.39	7.33	7.29
	3	15.91	11.56	9.39	7.66	7.48	7.43	7.37	7.31
	4	15.91	11.51	9.36	7.65	7.47	7.37	7.32	7.28
	5	15.90	11.53	9.42	7.69	7.47	7.38	7.34	7.31
	6	15.88	11.51	9.36	7.61	7.45	7.36	7.33	7.26
	7	15.88	11.54	9.37	7.64	7.48	7.37	7.31	7.27
	8	15.87	11.52	9.32	7.67	7.46	7.37	7.29	7.25
	9	15.87	11.52	9.32	7.68	7.45	7.35	7.27	7.26

Figure 5.4: Isolated bwaves (BE) execution time.

		Memory Bandwidth Allocation							
		10	20	30	40	50	60	90	100
Cache Allocation	1	6.18	5.98	5.91	5.88	5.87	5.89	5.87	5.86
	2	5.92	5.66	5.59	5.57	5.58	5.55	5.55	5.57
	3	5.82	5.58	5.52	5.48	5.46	5.46	5.47	5.44
	4	5.77	5.48	5.41	5.37	5.42	5.40	5.38	5.35
	5	5.73	5.43	5.36	5.33	5.31	5.32	5.30	5.30
	6	5.65	5.40	5.31	5.26	5.26	5.25	5.25	5.25
	7	5.62	5.33	5.26	5.21	5.21	5.21	5.24	5.21
	8	5.58	5.30	5.22	5.19	5.18	5.18	5.17	5.16
	9	5.57	5.28	5.34	5.15	5.13	5.14	5.13	5.13

Figure 5.5: Isolated omnetpp (BE) execution time.

		Memory Bandwidth Allocation							
		10	20	30	40	50	60	90	100
Cache Allocation	1	27.57	25.69	24.81	23.33	23.25	22.53	22.76	22.79
	2	26.18	24.07	23.44	21.70	21.81	21.54	21.46	20.93
	3	25.03	22.77	22.42	20.89	20.76	20.37	20.49	20.16
	4	24.13	22.46	21.27	20.20	19.67	19.74	19.20	19.48
	5	23.28	21.49	20.72	19.53	19.23	19.22	18.80	18.98
	6	22.86	21.20	20.42	19.09	18.87	18.79	18.67	18.53
	7	22.74	20.85	20.16	19.00	18.79	18.56	18.41	18.40
	8	22.78	20.87	20.08	18.95	18.88	18.34	18.40	18.37
	9	22.64	20.80	20.08	18.96	18.62	18.52	18.44	18.13

Figure 5.6: Isolated cactusADM (BE) execution time.

Next, we study the isolated performance of many BE workloads, again observing behavior as cache and memory bandwidth resources are scaled. As shown in Figures 5.4-5.6, these workloads exhibit diverse scaling behaviors. In particular, Bwaves (shown in Figure 5.4) from SPEC CPU2006 scales heavily with memory bandwidth yet is almost entirely unaffected by changes in cache allocation. This exclusive dependence on memory bandwidth could present a significant challenge for some prior works since cache restrictions alone would almost certainly be insufficient when managing contention. In contrast, some workloads such as Omnetpp (shown in Figure 5.5) from SPEC CPU2017 primarily scale with cache allocation and are only affected by memory bandwidth allocation when it is restricted to the lowest possible value(s). Naturally, many workloads scale with both cache and memory bandwidth allocation. CactusADM (shown in Figure 5.6) from SPEC CPU2006 is one such example that scales almost identically with both resources. In practically all of these BE workloads, we also observe that changes in performance occur most rapidly at lower resource values (e.g., 10/20/30% memory

vBNG Drop Rate

HP Cache Allocation	BE Memory Bandwidth Allocation: 10	20	30	40	50	60	90	100
2	1244	5580	11924	16041	16596	16371	15943	15471
3	162	254	363	579	609	658	629	626
4	71	69	105	171	167	198	178	147
5	32	71	118	146	215	170	152	162
6	44	38	71	78	59	61	103	112
7	48	54	70	67	99	101	93	141
8	48	55	71	108	85	119	106	109

BE Bops (Business Operations per second)

HP Cache Allocation	BE Memory Bandwidth Allocation: 10	20	30	40	50	60	90	100
2	291356	367039	394475	404315	406538	406086	406447	406650
3	288940	359865	383305	393066	392931	393102	393038	393010
4	288138	351409	370666	377765	378000	377517	377996	377304
5	286846	351149	370068	377681	378515	378168	378280	378176
6	279033	341335	358336	366375	365402	365538	365261	365712
7	276732	331554	345014	350740	351079	350928	351205	351288
8	273103	322921	335527	340663	340968	341645	340893	341525

Figure 5.7: Co-scheduled behavior of vBNG and jbb2005. The upper table shows the vBNG packet drop rate while the bottom table shows performance for jbb2005. HP memory bandwidth allocation is set to 100%. BE cache allocation is given all system ways not used by the HP workload.

bandwidth allocation and 2,3,4 cache ways) where these additional resources are actively utilized by the workload. Consequently, these "critical" regions of resource allocation tend to be important when co-scheduling.

Finally, we examine performance when co-scheduling HP and BE workloads. For these experiments, HP memory bandwidth allocation is set to a constant 100% while HP/BE cache and BE memory bandwidth allocation are changed, thus focusing on contention caused by increased BE resources. HP load is also held constant at 70%. Figure 5.7 presents results when co-scheduling the vBNG workload with SPEC jbb2005. Interestingly, jbb2005 introduces among the lowest contention of all tested workloads, resulting in a broad range of acceptable resource configurations (achieving a drop rate fraction less than 10^{-5}). We do, however, observe a QoS "cliff" where drop rate changes substantially between adjacent resource configurations. Here, we observe a cliff between

vBNG Drop Rate

HP Cache Allocation	BE Memory Bandwidth Allocation: 10	20	30	40	50	60	90	100
2	5915	42101	124316	225432	234465	242444	244552	254105
3	1006	7318	50542	113877	117531	121823	122083	127790
4	400	2027	41109	94383	96484	98450	104376	105619
5	286	778	31133	81326	81664	85690	86628	88950
6	175	461	36671	91429	92744	95479	97545	101489
7	162	339	37282	88238	86548	89984	91678	92793
8	153	290	39549	94045	99562	100755	104417	100653

BE Steps/sec

HP Cache Allocation	BE Memory Bandwidth Allocation: 10	20	30	40	50	60	90	100
2	1.89	2.43	2.79	3.14	3.17	3.17	3.21	3.19
3	1.89	2.44	2.79	3.15	3.18	3.20	3.24	3.21
4	1.87	2.42	2.77	3.08	3.11	3.12	3.14	3.17
5	1.87	2.42	2.78	3.11	3.13	3.17	3.20	3.20
6	1.89	2.41	2.78	3.10	3.16	3.17	3.19	3.20
7	1.86	2.40	2.76	3.05	3.08	3.08	3.08	3.11
8	1.86	2.38	2.73	3.02	3.04	3.06	3.09	3.07

Figure 5.8: Co-scheduled behavior of vBNG and Resnet50. The upper table shows the vBNG packet drop rate while the bottom table shows performance for Resnet50. HP memory bandwidth allocation is set to 100%. BE cache allocation is given all system ways not used by the HP workload.

two and three cache ways. Other BE workloads introduce substantially greater contention. As shown in Figure 5.8, resources for a machine learning workload (specifically, training a Resnet50 model) must be highly restricted in order to meet desired QoS levels. In this case, providing more than a few cache ways or more than 10% memory bandwidth allocation for the BE workload leads to unstable HP performance. QoS cliffs for the HP application also extend far into the range of configuration options. The potential complexity in co-scheduling BE workloads alongside our HP vBNG workload requires an intelligent approach that can avoid QoS violations while still maximizing BE performance.

5.3 Proactive Control Framework

5.3.1 Overview

The proposed resource control framework adopts a novel approach to directly predict workload QoS, rather than waiting for measurements. In doing so, we address a fundamental limitation in prior work and enable stronger QoS guarantees in more stringent operating environments.

The overall resource allocation procedure in illustrated in Figure 5.9. First, the telemetry module collects runtime information (e.g., general-purpose performance counters) for all workloads via the Linux *perf* tool. We provide a detailed discussions of these performance counters in Section 5.3.2. This information is then passed to the QoS prediction model, which predicts the QoS value (e.g., packet drop rate) that will be achieved by each HP workload, assuming no change in resource allocation. These QoS predictions, along with all runtime information, are provided to a reinforcement-learning-based controller, which decides an appropriate resource allocation for the next interval. In general, resource allocation can be performed between any number of HP and BE workloads provided that there are sufficient resources to meet QoS goals for all HP workload.

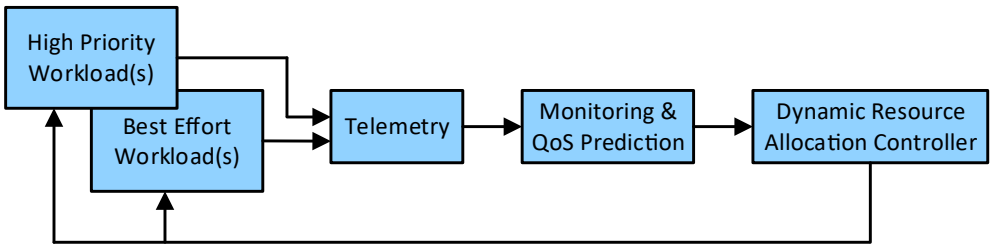


Figure 5.9: Resource control framework.

5.3.2 QoS Prediction

Transient, yet significant, fluctuations in workload demands make direct QoS measurements an unreliable source of information in some operating environments. Further, any attempts to stabilize measurements by averaging over longer periods may lead to

unacceptably long periods of QoS violations. We address these problem by predicting, rather than measuring, the worst-case QoS value (e.g., packet drop rate) that will be achieved by all high priority workloads. In doing so, resource allocation can take into consideration any expected performance fluctuations and thereby avoid many potential QoS violations.

Predictor Setup: In our framework, QoS prediction is cast as a supervised learning regression task. Input consists of several hardware performance counters while the prediction target is the QoS value. This data is gathered across a broad range of co-scheduling configurations (i.e., BE workloads) and resource configurations, with the target value being the worst-case QoS value that is measured during each sampling period. In particular, we found that approximately 100 individual QoS measurements are required to achieve satisfactory statistical guarantees. For comparison, prior work using online sampling would require approximately 100 second sampling intervals (assuming one measurement per second as allowed by our QoS measurement tool[2]) to achieve similar QoS guarantees, rendering an online approach undesirable for many real-world applications.

We found that highly-accurate QoS prediction can be a difficult task when generalizing across a wide variety of co-scheduled applications. In particular, the contention caused by individual workloads can vary significantly with their resource allocation due to changes in execution bottlenecks. These differences become particularly apparent when HP workload intensity is concurrently varied across a wide range. We address these issues via a two-level QoS prediction setup. Hardware performance readings are first passed to a supervised learning classifier that determines whether the QoS level will be above or below a desired threshold. This prediction is then used to select between two possible supervised learning regressor (that provide the actual QoS prediction). One regressor is trained on the "critical" range of QoS values in which the framework will ideally operate (e.g., zero up to the threshold value). A second regressor is instead trained on the full range of QoS values. In doing so, the regressor trained on the smaller range can focus on contention behaviors that are most likely to occur during typical framework execution, while the full-range regressor can still predict the impact of coarse-grained contention behaviors during periods of rapid fluctuations in workload demands.

[2]For our HP vBNG workload, we use the collectd [151] dpdk_telemetry plugin, which has a minimum update frequency of 1 Hz.

Table 5.1: Selected Features

Feature		Description
Fixed Performance Counters	inst_retired.any	Counts retired instructions
	cpu_clk_unhalted.thread	Counts cycles when the core is not halted
General Performance Counters	frontend_retired.latency_ge_2	Counts retired instructions following a period of ≥ 2 cycles with no uops delivered by the frontend
	l1d_pend_miss.fb_full	Counts cases when a fill buffer entry was requested, but unavailable
	offcore_requests.all_requests	Counts off-core memory transactions
	offcore_requests_buffer.sq_full	Counts cases when the off-core requests buffer was full

Feature Selection: Hardware performance counters for all these predictors are selected via a combination of automated analysis and human-expert domain knowledge. This procedure is meant to guide feature selection towards performance counters that are both informative and easily explainable, thus more likely to generalize to new applications. An initial list of approximately 1000 potential counters is first reduced by eliminating counters with low variance. Next, we perform a series of permutation tests. In each test, all feature columns are copied and then the individual values within each column are randomly shuffled. The importance of the original feature columns are then compared against the importance of their randomly shuffled equivalents to provide a robust estimate of true feature importance, even in the presence of pairwise (or higher order) relationships between features. At this point, a small number of features are eliminated based on human-expert knowledge. Finally, the remaining features are reduced to the desired number via recursive feature elimination on a random forest model.[3] The resulting set of hardware performance counters (shown in Table 5.1) maintains strong prediction accuracy (equivalent to any larger set of counters) while maintaining low overhead.

[3]We find that performance counter multiplexing (required when more than four general purpose counters are measured per thread) can introduce substantial overhead for some HP applications. As such, we strictly limit our list to meet this criteria.

5.3.3 Dynamic Resource Allocation Controller

Most prior works strictly avoid offline training for resource controllers to allow for immediate adaptation to new operating environments. Although this approach can be beneficial in some circumstances, it remains fundamentally incompatible when QoS samples alone are not enough to avoid transient QoS violations and offline QoS predictor training is required. Consequently, we adopt a reinforcement-learning-based resource controller in our framework in order to exploit the benefits from offline learning. Whereas purely online approaches must re-sample configurations when workload demands change (commonly introducing QoS violations), the trained reinforcement learning model can directly select appropriate resource configurations and can therefore be applied to applications where faster resource re-configuration becomes necessary.

Model Architecture: The reinforcement-learning-based resource controller in our framework uses a branching dueling Q-network (BDQ) architecture (depicted in Figure 5.10). This architecture features a shared network module, followed by distinct action branches, one for each control knob (e.g., cache, memory bandwidth, etc.). Splitting these action dimensions allows the number of network outputs to grow linearly, rather than combinatorially, with respect to the action space, thus simplifying control complexity significantly when allocating more than a few distinct resources. The proposed reinforcement learning resource controller integrates several additional techniques (target network, prioritized replay buffer, and a cyclic learning rate) to improve training stability and shorten overall training time.

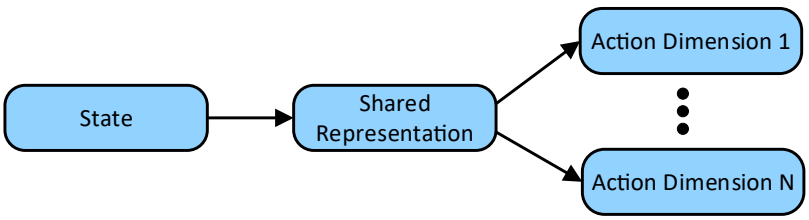


Figure 5.10: Branching dueling Q-network (BDQ) model architecture.

State/Action/Reward Representation: Input to the reinforcement learning resource controller includes IPC (calculated from the fixed performance counters in Table 5.1), the general-purpose performance counters in Table 5.1, and the predicted QoS

from selected regressor model. IPC and the general-purpose counters act as indicators for architectural pressures, essentially indicating whether key execution processes (e.g., prefetching), are performing as expected or are likely to begin affecting the QoS. Similarly, the predicted QoS provides both "worst-case" QoS estimate and a higher-level perspective on architecture pressures.

HP resource allocation is selected directly by the reinforcement learning model while BE resource allocation is derived from HP resource allocation. This distinction allows the resource controller to be trained for a more general-purpose operating environment where BE workloads may change frequently, contrasting with prior work (e.g., Twig [152]) that consider only a fixed setup with only HP workloads. Cache allocation is straightforward since LLC ways can be strictly disjoint for each application, thus all cache ways not selected for the HP workload are given to the BE workload(s). Memory bandwidth allocation, on the other hand, is not a strictly divided resource; Intel® Resource Director Technology currently implements memory bandwidth control separately for each class of service (i.e., workload in this paper). Regardless, we can still allocate memory bandwidth as if it were strictly divided by selecting from an extended range (e.g., 0-200%) with the reinforcement learning controller. Values in the range of 0-100% limit HP memory bandwidth while leaving BE memory bandwidth unlimited. Conversely, values in the range of 100-200% limit BE memory bandwidth while leaving HP memory bandwidth unlimited. A shorter range (e.g., 0-120%) could allow an overlap region where both workloads are moderately limited.

Reward setup is given in Equations 5.1 - 5.3. Constants (written in all caps) allow for user defined resource and QoS targets.[4] Note that, unlike prior work, rewards are based on the *predicted* not the *sampled* QoS. In almost all cases, the predicted maximum QoS will be closer to the true maximum QoS than the current reading, even with prediction error. As such, rewards based on current samples are far more likely to grant positive rewards for improper resource allocations that would allow transient QoS violations. Further, rewards given by current samples may be inconsistent due to natural fluctuations, thus hindering learning. With this setup, positive rewards are given when the predicted QoS is less than the target while negative penalties are given when the

[4]We select a minimum and maximum of 2 and 9 cache ways, respectively, for each application. Minimum and maximum values for memory bandwidth allocation (MBA) are set to 10% and 100%. Both the desired QoS buffer and target QoS are application specific.

predicted QoS is greater than the target. Positive rewards involve a separate component for cache and memory, both of which encourage HP workload to use fewer resources while still meeting QoS targets. Negative penalties are based strictly on the difference between the predicted and desired QoS.

$$r = \begin{cases} \dfrac{cache\_reward + memory\_reward}{2} & \text{if predicted\_qos} \leq \text{TARGET\_QOS} \\ -\min[\,\dfrac{1}{2} * (\dfrac{predicted\_qos}{TARGET\_QOS} - 1),\, 1\,] & \text{if predicted\_qos} > \text{TARGET\_QOS} \end{cases} \tag{5.1}$$

$$cache\_reward = \frac{MIN\_CACHE}{chosen\_cache} - \frac{MIN\_CACHE}{MAX\_CACHE} \tag{5.2}$$

$$memory\_reward = \frac{MIN\_MBA}{chosen\_memory} - \frac{MIN\_MBA}{MAX\_MBA} \tag{5.3}$$

5.4 Initial Results

5.4.1 Setup

Platform and workload setup is specified in Table 5.2. Cores used for these experiments are isolated to ensure that there is no interference from the OS. Resource controllers are pinned to additional isolated cores to ensure consistent results and limit interference with the HP and BE workloads. All cores are set to 2.7 GHz (base frequency) and boosting is disabled. HP vBNG workload demand (i.e., network packet injection rate) in all tests

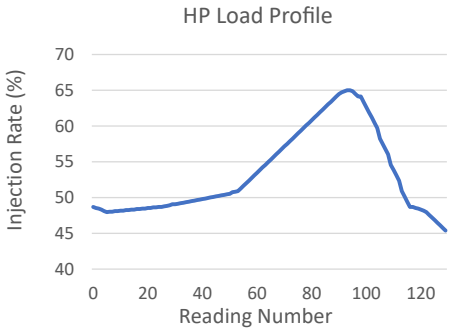
Table 5.2: Platform & Workload Configuration

Item		Description
Evaluation Platform		Intel Xeon Platinum 8280 (38.5MB LLC, 11 ways) All workloads pinned to socket 1
Workload Setup	SPEC CPU2006	12 copies, 12 threads, test/train input sizes
	SPEC CPU2017	12 copies, 12 threads, test/train input sizes
	SPEC jbb2005	24 warehouses, 24 threads
	PARSEC	16 threads, native input size
	Resnet50 (train)	24 threads, CIFAR-10 dataset

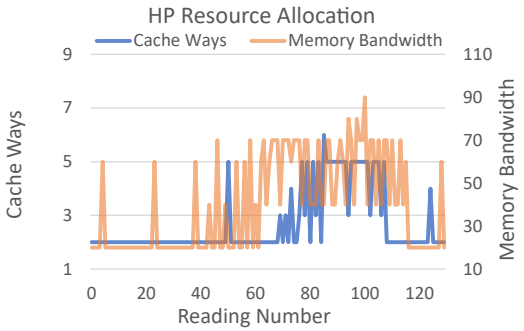
follows realistic traffic profiles gathered from real-world operating environments.

5.4.2 Evaluation

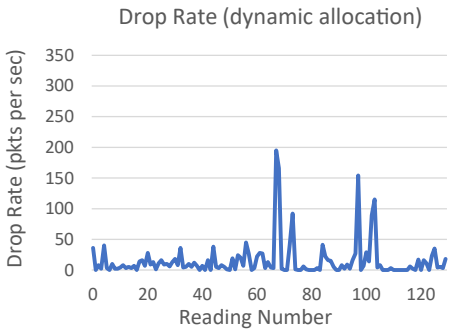
For initial testing, we evaluate performance when co-scheduling one HP workload (vBNG) and one BE workload (SPEC jbb2005). HP workload demand follows the curve shown in Figure 5.11a. During the initial period of lower demand (reading 1-60), the proposed framework appropriately identifies 2 cache ways and 20% memory bandwidth (shown in Figure 5.11b) as sufficient to meet the target QoS, with the measured QoS (shown in Figure 5.11c) fluctuating around 0-50 packets dropped per second. As demand increases,



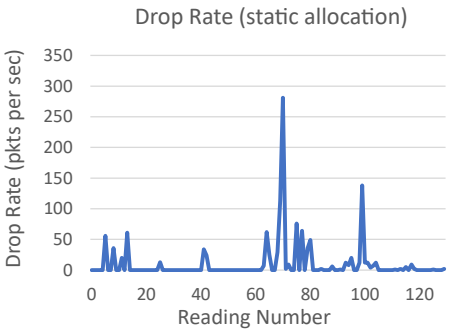
(a) Dynamic workload demand for the HP (vBNG) application.



(b) HP (vBNG) resource allocation selected by proposed framework.



(c) Measured vBNG drop rate using resource allocation in Figure 5.11c.



(d) Measured vBNG drop rate using static resource allocation of 5 cache ways and 70% memory bandwidth.

Figure 5.11: Resource allocation comparison.

the controller determines that 2 cache ways is no longer sufficient and begins allocating roughly five cache ways, along with higher memory bandwidth, to the HP vBNG workload. Several substantial spikes in the drop rate are observed, with one approaching 200 packets per second, but still well under the specified drop rate target of 10^{-5} (approximately 300 packets per second). Finally, after demand peaks and begins to decrease, the controller likewise decreases resource allocation for the HP vBNG workload. We additionally plot the drop rate when statically allocation the maximum amount of resource selected by the reinforcement learning model (Figure 5.11d). During periods of lower demand, these additional resources can provide marginal benefits in terms of lower and more stable drop rate, but provide no practical benefits since two cache ways and 20% memory bandwidth were already sufficient to prevent QoS violations.

5.5 Conclusion

Co-scheduling of high-priority and best-effort workloads, enabled by dynamic resource allocation, has become a widely adopted technique to improve machine utilization and reduce total cost of ownership. Existing approaches based on QoS sampling provide acceptable guarantees when workload demand is stable, yet fail to address transient QoS violations that may become critical when targeting stricter QoS guarantees. Our proposed framework solves these problems by directly predicting the worst-case QoS level that may occur. These QoS predictions are combined with an intelligent reinforcement-learning-based resource controller to enable proactive, rather than reactive, resource allocation that mitigates transient QoS violations. Evaluation shows that the proposed framework successfully minimizes resource allocation for the HP workload while providing equivalent QoS guarantees as a static allocation scheme based on oracle knowledge.

Chapter 6: General Conclusion

Increasing complexity in modern computing systems, combined with the slowing of technology-based advances, has prompted a new, machine-learning-based paradigm for computer architecture design.

In Chapter 3, we observed how a growing number of works are applying machine learning to practically all major components, including the core, cache/memory, NoC, and GPUs. Notably, these applications often provide state-of-the-art results, surpassing long-standing design strategies based solely on human-expert knowledge, exhaustive search, and heuristic approximations. Next, Chapter 4 highlighted the immense potential for design space exploration via deep reinforcement learning. In particular, the NoC design case study highlighted how a design space exceeding 10^{392} can be successfully navigated, even under strict design constraints. Finally, Chapter 5 introduced an innovative strategy for machine-learning-based resource management under strict QoS guarantees. The proposed framework, based on proactive QoS prediction, demonstrated how machine learning can effectively mitigate contention between applications, thereby enabling new opportunities for performance optimization and/or power saving. Taken together, these works present a promising future for machine-learning-based architectural design.

Bibliography

[1] Q. Xie, M.-T. Luong, E. Hovy, and Q. V. Le, "Self-training with noisy student improves imagenet classification," 2019. arXiv:1911.04252.

[2] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 2001.

[3] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *International Symposium on High-Performance Computer Architecture (HPCA)*, June 2008.

[4] D. Penney and L. Chen, "A Survey of Machine Learning Applied to Computer Architecture Design," Sep. 2019. arXiv:1909.12373.

[5] T.-R. Lin, D. Penney, M. Pedram, and L. Chen, "A Deep Reinforcement Learning Framework for Architectural Exploration: A Routerless NoC Case Study," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2020.

[6] S. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real World AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pp. 3–24, 2007.

[7] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffman, "A probabilistic graphical model-based approach for minimizing energy under performance constraints," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2015.

[8] V. N. Vapnik, "An overview of statistical learning theory," *IEEE Transactions on Neural Networks*, vol. 10, Sep. 1999.

[9] J. Shlens, "A tutorial on principal component analysis," 2014. arXiv:1404.1100.

[10] M. Alawieh, F. Wang, and X. Li, "Efficient hierarchical performance modeling for integrated circuits via bayesian co-learning," in *Design Automation Conference (DAC)*, June 2017.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, USA: MIT Press, 2nd ed., 1998.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," in *NIPS Deep Learning Workshop*, Dec. 2013.

[13] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," in *Nature*, Jan. 2016.

[14] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," in *Nature*, Oct. 2017.

[15] E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana, "Efficiently exploring architectural design spaces via predictive modeling," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006.

[16] B. Ozisikyilmaz, G. Memik, and A. Choudhary, "Machine learning models to predict performance of computer system design alternatives," in *International Conference on Parallel Processing (ICPP)*, Sept. 2008.

[17] S. Eyerman, K. Hoste, and L. Eeckhout, "Mechanistic-empirical processor performance modeling for constructing cpi stacks on real hardware," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2011.

[18] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, July 2015.

[19] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," in *Design Automation Conference (DAC)*, June 2016.

[20] N. Agarwal, T. Jain, and M. Zahran, "Performance prediction for multi-threaded applications," in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, June 2019.

[21] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based gpu design space exploration," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2012.

[22] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2015.

[23] A. Jooya, N. Dimopoulos, and A. Baniasadi, "Multiobjective gpu design space exploration optimization," in *International Conference on High Performance Computing & Simulation (HPCS)*, July 2016.

[24] T.-R. Lin, Y. Li, M. Pedram, and L. Chen, "Design space exploration of memory controller placement in throughput processors with deep learning," in *IEEE Computer Architecture Letters*, vol. 18, Mar. 2019.

[25] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2014.

[26] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *International Symposium on Microarchitecture (MICRO)*, June 2015.

[27] N. Ardalani, U. Thakker, A. Albarghouthi, and K. Sankaralingam, "A static analysis-based cross-architecture performance prediction using machine learning," in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, June 2019.

[28] K. O'Neal, P. Brisk, E. Shriver, and M. Kishinevsky, "Halwpe: Hardware-assisted light weight performance estimation for gpus," in *Design Automation Conference (DAC)*, June 2017.

[29] Y. Li, D. Penney, A. Ramamurthy, and L. Chen, "Characterizing on-chip traffic patterns in general-purpose gpus: A deep learning approach," in *International Conference on Computer Design (ICCD)*, Nov. 2019.

[30] A. Pattnaik, X. Tang, A. Jog, O. Kayıran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2016.

[31] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *International Symposium on High-Performance Computer Architecture (HPCA)*, June 2015.

[32] Y. Zeng and X. Guo, "Long short term memory based hardware prefetcher," in *International Symposium on Memory Systems (MemSys)*, Oct. 2017.

[33] P. Braun and H. Litz, "Understanding memory access patterns for prefetching," in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, June 2019.

[34] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *International Symposium on Computer Architecture (ISCA)*, June 2019.

[35] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *International Symposium on Microarchitecture (MICRO)*, Oct. 2016.

[36] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou, "Efficient ssd caching by avoiding unnecessary writes using machine learning," in *International Conference on Parallel Processing (ICPP)*, Aug. 2018.

[37] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Virtual address translation via learned page tables indexes," in *Conference on Neural Information Processing Systems (NeurIPS)*, Dec. 2018.

[38] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *International Conference on Management of Data (SIGMOD)*, June 2018.

[39] J. Mukundan and J. F. Martinez, "Morse: Multi-objective reconfigurable self-optimizing memory scheduler," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2012.

[40] S. Manoj, H. Yu, H. Huang, and D. Xu, "A q-learning based self-adaptive i/o communication for 2.5d integrated many-core microprocessor and memory," *IEEE Transactions on Computers*, vol. 65, June 2015.

[41] S. Wang and E. Ipek, "Reducing data movement energy via online data clustering and encoding," in *International Symposium on Microarchitecture (MICRO)*, Oct. 2016.

[42] W. Kang and S. Yoo, "Dynamic Management of Key States for Reinforcement Learning-assisted Garbage Collection to Reduce Long Tail Latency in SSD," in *Design Automation Conference (DAC)*, June 2018.

[43] Z. Deng, L. Zhang, N. Mishra, H. Hoffman, and F. T. Chong, "Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms," in *International Symposium on Microarchitecture (MICRO)*, Oct. 2017.

[44] J. Xiao, Z. Xiong, S. Wu, Y. Yi, H. Jin, and K. Hu, "Disk failure prediction in data centers via online learning," in *International Conference on Parallel Processing (ICPP)*, June 2018.

[45] R. S. Amant, D. A. Jiménez, and D. Burger, "Low-power, high-performance analog neural branch prediction," in *International Symposium on Microarchitecture (MICRO)*, Nov. 2008.

[46] D. A. Jiménez, "An optimized scaled neural branch predictor," in *International Conference on Computer Design (ICCD)*, Oct. 2011.

[47] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez, "Bit-level perceptron prediction for indirect branches," in *International Symposium on Computer Architecture (ISCA)*, June 2019.

[48] A. Seznec, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions: Championship Branch Prediction, held in conjunction with ISCA*, 2016.

[49] S. J. Tarsa, C.-K. Lin, G. Keskin, G. Chinya, and H. Wang, "Improving branch prediction by modeling global history with convolutional neural networks," in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, June 2019.

[50] A. G. Savva, T. Theocharides, and V. Soteriou, "Intelligent on/off dynamic link management for on-chip networks," in *Journal of Electrical and Computer Engineering - Special issue on Networks-on-Chip: Architectures, Design Methodologies, and Case Studies*, Jan 2012.

[51] D. DiTomaso, A. Sikder, A. Kodi, and A. Louri, "Machine learning enabled power-aware network-on-chip design," in *Design, Automation and Test in Europe (DATE)*, Mar. 2017.

[52] S. V. Winkle, A. Kodi, R. Bunescu, and A. Louri, "Extending the power-efficiency and performance of photonic interconnects for heterogeneous multicores with machine learning," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2018.

[53] M. F. Reza, T. T. Le, B. De, M. Bayoumi, and D. Zhao, "Neuro-noc: Energy optimization in heterogeneous many-core noc using neural networks in dark silicon era," in *International Symposium on Circuits and Systems (ISCAS)*, May 2018.

[54] M. Clark, A. Kodi, R. Bunescu, and A. Louri, "Lead: Learning-enabled energy-aware dynamic voltage/frequency scaling in nocs," in *Design Automation Conference (DAC)*, June 2018.

[55] Q. Fettes, M. Clark, R. Bunescu, A. Karanth, and A. Louri, "Dynamic voltage and frequency scaling in nocs with supervised and reinforcement learning techniques," *IEEE Transactions on Computers*, vol. 68, Mar. 2019.

[56] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: a reinforcement learning approach," *Advances in Neural Information Processing Systems*, vol. 6, pp. 671–678, 1994.

[57] M. Majer, C. Bobda, A. Ahmadinia, and J. Teich, "Packet routing in dynamically changing networks on chip," in *International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2005.

[58] C. Feng, Z. Lu, A. Jantsch, J. Li, and M. zhang, "A reconfigurable fault-tolerant deflection routing algorithm based on reinforcement learning for network-on-chip," in *International Workshop on Network on Chip Architectures (NoCArc), held in conjunction with MICRO*, Dec. 2010.

[59] M. Ebrahimi, M. Daneshtalab, and F. Farahnakian, "Haraq: Congestion-aware learning model for highly adaptive routing algorithm in on-chip networks," in *International Symposium on Networks-on-Chip (NOCS)*, June 2012.

[60] B. K. Daya, L.-S. Peh, and A. P. Chandrakasan, "Quest for high-performance bufferless nocs with single-cycle express paths and self-learning throttling," in *Design Automation Conference (DAC)*, June 2016.

[61] B. Wang, Z. Lu, and S. Chen, "Ann based admission control for on-chip networks," in *Design Automation Conference (DAC)*, June 2019.

[62] V. Soteriou, T. Theocharides, and E. Kakoulli, "A holistic approach towards intelligent hotspot prevention in network-on-chip-based multicores," *IEEE Transactions on Computers*, vol. 65, May 2015.

[63] J. Yin, Y. Eckert, S. Che, M. Oskin, and G. H. Loh, "Toward more efficient noc arbitration: A deep reinforcement learning approach," in *International Workshop on AI-assisted Design for Architecture (AIDArc), held in conjunction with ISCA*, June 2018.

[64] S. Das, J. R. Doppa, D. H. Kim, P. P. Pande, and K. Chakrabarty, "Optimizing 3d noc design for energy efficiency: A machine learning approach," in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2015.

[65] S. Das, J. R. Doppa, P. P. Pande, and K. Chakrabarty, "Energy-efficient and reliable 3d network-on-chip (noc): Architectures and optimization algorithms," in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2016.

[66] B. K. Joardar, R. G. Kim, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "Learning-based application-agnostic 3d noc design for heterogeneous manycore systems," *IEEE Transactions on Computers*, vol. 68, June 2019.

[67] T.-R. Lin, D. Penney, M. Pedram, and L. Chen, "Optimizing routerless network-on-chip designs: An innovative learning-based framework," May 2019. arXiv:1905.04423.

[68] N. Rao, A. Ramachandran, and A. Shah, "Mlnoc: A machine learning based approach to noc design," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Sept. 2018.

[69] S. Bandyopadhyay, S. Saha, U. Maulik, and K. Deb, "A simulated annealing-based multiobjective optimization algorithm: Amosa," *IEEE Transactions on Evolutionary Computation*, vol. 12, May 2008.

[70] Z. Qian, D.-C. Juan, P. Bogdan, C.-Y. Tsui, D. Marculescu, and R. Marculescu, "Svr-noc: A performance analysis tool for network-on-chips using learning-based support vector regression model," in *Design, Automation and Test in Europe (DATE)*, Mar. 2013.

[71] K. Sangaiah, M. Hempstead, and B. Taskin, "Uncore rpd: Rapid design space exploration of the uncore via regression modeling," in *International Conference on Computer-Aided Design (ICCAD)*, Nov. 2015.

[72] D. DiTomaso, T. Boraten, A. Kodi, and A. Louri, "Dynamic error mitigation in nocs using intelligent prediction techniques," in *International Symposium on Microarchitecture (MICRO)*, Oct. 2016.

[73] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "High-performance, energy-efficient, fault-tolerant network-on-chip design using reinforcement learning," in *Design, Automation and Test in Europe (DATE)*, Mar. 2019.

[74] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "Intellinoc: A holistic design framework for energy-efficient and reliable on-chip communication for manycores," in *International Symposium on Computer Architecture (ISCA)*, June 2019.

[75] J.-Y. Won, X. Chen, P. Gratz, J. Hu, and V. Soteriou, "Up by their bootstraps: Online learning in artificial neural networks for cmp uncore power management," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2014.

[76] G.-Y. Pan, J.-Y. Jou, and B.-C. Lai, "Scalable power management using multi-level reinforcement learning for multiprocessors," in *ACM Transactions on Design Automation of Electronic Systems*, Aug. 2014.

[77] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. de Supinski, "Adaptive configuration selection for power-constrained heterogeneous systems," in *International Conference on Parallel Processing (ICPP)*, Sept. 2014.

[78] D. Lo, T. Song, and G. E. Suh, "Prediction-guided performance-energy trade-off for interactive applications," in *International Symposium on Microarchitecture (MICRO)*, Dec. 2015.

[79] N. Mishra, J. D. Lafferty, and H. Hoffman, "Caloree: Learning control for predictable latency and low energy," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2018.

[80] Y. Bai, V. W. Lee, and E. Ipek, "Voltage regulator efficiency aware power management," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2017.

[81] M. Allen and P. Fritzsche, "Reinforcement learning with adaptive kanerva coding for xpilot game ai," in *IEEE Congress of Evolutionary Computation*, June 2011.

[82] Z. Chen and D. Marculescu, "Distributed reinforcement learning for power limited many-core system performance optimization," in *Design, Automation and Test in Europe (DATE)*, Mar. 2015.

[83] Z. Chen, D. Stamoulis, and D. Marculescu, "Profit: Priority and power/performance optimization for many-core systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, pp. 2064–2075, Oct. 2018.

[84] C. Imes, S. Hofmeyr, and H. Hoffman, "Energy-efficient application resource scheduling using machine learning classifiers," in *International Conference on Parallel Processing (ICPP)*, Aug. 2018.

[85] S. J. Tarsa, R. B. R. Chowdhury, J. Sebot, G. Chinya, J. Gaur, K. Sankaranarayanan, C.-K. Lin, R. Chappell, R. Singhal, and H. Wang, "Post-silicon cpu

adaptation made practical using machine learning," in *International Symposium on Computer Architecture (ISCA)*, June 2019.

[86] S. J. Lu, R. Tessier, and W. Burleson, "Reinforcement learning for thermal-aware many-core task allocation," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, May 2015.

[87] D. Nemirovsky, T. Arkose, N. Markovic, M. Nemirovsky, O. Unsal, and A. Cristal, "A machine learning approach for performance prediction and scheduling on heterogeneous cpus," in *International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2017.

[88] H. Zhang, B. Tang, X. Geng, and H. Ma, "Learning Driven Parallelization for Large-Scale Video Workload in Hybrid CPU-GPU Cluster," in *International Conference on Parallel Processing (ICPP)*, Aug. 2018.

[89] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *International Symposium on Microarchitecture (MICRO)*, Nov. 2008.

[90] R. Jain, P. R. Panda, and S. Subramoney, "Machine learned machines: Adaptive co-optimization of caches, cores, and on-chip network," in *Design, Automation and Test in Europe (DATE)*, Mar. 2016.

[91] Y. Ding, N. Mishra, and H. Hoffmann, "Generative and multi-phase learning for computer systems optimization," in *International Symposium on Computer Architecture (ISCA)*, June 2019.

[92] G. Wu, Y. Xu, D. Wu, M. Ragupathy, Y. yen Mo, and C. Chu, "Flip-flop clustering by weighted k-means algorithm," in *Design Automation Conference (DAC)*, June 2016.

[93] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, Mar. 2016.

[94] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *International Symposium on Microarchitecture (MICRO)*, Dec. 2012.

[95] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for gpu throughput processors," in *International Symposium on Microarchitecture (MICRO)*, Dec. 2015.

[96] B. Grigorian, N. Farahpour, and G. Reinman, "Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2015.

[97] D. Mahajan, A. Yazdanbaksh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *International Symposium on Computer Architecture (ISCA)*, June 2016.

[98] G. F. Oliveira, L. R. Goncalves, M. Brandalero, A. C. S. Beck, and L. Carro, "Employing classification-based algorithms for general-purpose approximate computing," in *Design Automation Conference (DAC)*, June 2018.

[99] F. N. Taher, J. Callenes-Sloan, and B. C. Schafer, "A machine learning based hard fault recuperation model for approximate hardware accelerators," in *Design Automation Conference (DAC)*, June 2018.

[100] X. Chen, Z. Xu, H. Kim, P. Gratz, J. Hu, M. Kishinevsky, and U. Ogras, "In-network monitoring and control policy for dvfs of cmp networks-on-chip and last level caches," in *International Symposium on Networks-on-Chip (NOCS)*, May 2012.

[101] R. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *Conference on Neural Information Processing Systems (NeurIPS)*, June 1996.

[102] J. A. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *The Journal of Machine Learning Research*, Sep. 2001.

[103] P. Bratley and B. L. Fox, "Algorithm 659: Implementing sobol's quasirandom sequence generator," *ACM Transactions on Mathematical Software*, vol. 14, Mar. 1988.

[104] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2001.

[105] W. Wang, J. W. Davidson, and M. L. Soffa, "Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Mar. 2016.

[106] R. M. Kretchmar, "Reinforcement learning algorithms for homogenous multi-agent systems," in *Workshop on Agent and Swarm Programming*, 2003.

[107] T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Conference on Neural Information Processing Systems (NeurIPS)*, Dec. 2016.

[108] H. Mao, Z. Gong, Z. Zhang, Z. Xiao, and Y. Ni, "Learning multi-agent communication under limited-bandwidth restriction for internet packet routing," Feb. 2019. arXiv:1903.05561.

[109] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," Aug. 2017. arXiv:1703.09039.

[110] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," Oct. 2015. arXiv:1506.02626.

[111] D. C. Mocanu, E. Mocanu, P. Stone, P. H. Nguyen, M. Gibescu, and A. Liotta, "Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science," *Nature Communications*, vol. 9, June 2018.

[112] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," Mar. 2016. arXiv:1602.02830.

[113] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "Looknn: Neural network with no multiplication," in *Design, Automation and Test in Europe (DATE)*, Mar. 2017.

[114] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[115] R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. J. Kim, "Approx-noc: A data approximation framework for network-on-chip architectures," in *International Symposium on Computer Architecture (ISCA)*, June 2017.

[116] A. Raha and V. Raghunathan, "Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart camera system," in *Design Automation Conference (DAC)*, June 2017.

[117] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing," *University of Washington Technical Report*, vol. 1, Jan. 2015.

[118] B. Reagen, J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Wahtmoug, G.-Y. Wei, and D. Brooks, "A case for efficient accelerator design space exploration via bayesian optimization," in *International Symposium on Low Power Electronics and Design (ISLPED)*, July 2017.

[119] A. Vallero, A. Savino, G. Politano, S. D. Carlo, A. Chatzidimitriou, S. Tselonis, M. Kaliorakis, D. Gizopoulos, M. Riera, R. Canal, A. Gonzalez, M. Kooli, A. Bosio, and G. D. Natale, "Cross-layer system reliability assessment framework for hardware faults," in *International Test Conference (ITC)*, Nov. 2016.

[120] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference*, June 2001.

[121] S. Liu, T. Chen, L. Li, X. Feng, Z. Xu, H. Chen, F. Chong, and Y. Chen, "Imr: High-performance low-cost multi-ring nocs," in *IEEE Transactions on Parallel Distributed Systems*, June 2016.

[122] F. Alazemi, A. Azizimazreah, B. Bose, and L. Chen, "Routerless networks-on-chip," in *IEEE International Symposium on High Performance Computer Architecture*, Feb. 2018.

[123] T. W. Ainsworth and T. M. Pinkston, "On characterizing performance of the cell broadband engine element interconnect bus," in *International Symposium on Networks-on-Chip*, May 2007.

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

[125] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger, "Implementation and evaluation of on-chip network architecture," in *International Conference on Computer Design*, Nov. 2007.

[126] L. Chen and T. M. Pinkston, "Nord: Node-router decoupling for effective power-gating of on-chip routers," in *International Symposium on Microarchitecture*, Dec. 2012.

[127] Y. Hoskote, S. Vangal, A. Singh, H. Borkar, and S. Borkar, "A 5-ghz mesh interconnect for a teraflops processor," in *IEEE Micro*, Nov. 2007.

[128] M. McKeown, A. Lavrov, M. Shahrad, P. J. Jackson, Y. Fu, J. Balkind, T. M. Nguyen, K. Lim, Y. Zhou, and D. Wentzlaff, "Power and energy characterization of an open source 25-core manycore processor," in *International Symposium on High Performance Computer Architecture*, Feb 2018.

[129] J. Balkind, M. McKeown, Y. Fu, T. M. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Feb 2016.

[130] R. Ausavarungnirun, C. Fallin, X. Yu, K. K.-W. Chang, G. Nazario, R. Das, G. H. Loh, and O. Mutlu, "Design and evaluation of hierarchical rings with deflection routing," in *International Symposium on Computer Architecture and High Performance Computing*, Oct. 2014.

[131] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian, "Towards scalable, energy-efficient, bus-based on-chip networks," in *International Symposium on High-Performance Computer Architecture*, Jan. 2010.

[132] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. V. D. Wijngaart, "A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling," in *IEEE Journal of Solid-State Circuits*, Jan. 2011.

[133] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, June 2016.

[134] C. D. Rosin, "Multi-armed bandits with episode context," in *Annals of Mathematics and Artificial Intelligence*, Mar. 2011.

[135] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, June 2016.

[136] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basil, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *Computer Architecture News*, vol. 39, pp. 1–7, 2011.

[137] Nangate Inc., "Nangate freepdk15 open cell library." [Online]. Available: http://www.nangate.com.

[138] M. Badr and N. Enright Jerger, "Synfull: synthetic traffic models capturing cache coherent behaviour," in *International Symposium on Computer Architecture*, 2014.

[139] S. Das, J. R. Doppa, P. P. Pande, and K. Chakrabarty, "Energy-efficient and reliable 3d network-on-chip (noc): Architectures and optimization algorithms," in *International Conference on Computer-Aided Design*, Nov. 2016.

[140] N. Enright Jerger, A. Kannan, Z. Li, and G. H. Loh, "Noc architectures for silicon interposer systems," in *International Symposium on Microarchitecture*, Dec. 2014.

[141] A. Kannan, N. Enright Jerger, and G. H. Loh, "Enabling interposer-based disintegration of multi-core processors," in *International Symposium on Microarchitecture*, Dec. 2015.

[142] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. S. B. Altaf, N. Enright Jerger, and G. H. Loh, "Modular routing design for chiplet-based systems," in *International Symposium on Computer Architecture*, June 2018.

[143] G. H. Loh, N. Enright Jerger, A. Kannan, and Y. Eckert, "Interconnect-memory challenges for multi-chip, silicon interposer systems," in *International Symposium on Memory Systems*, Oct. 2015.

[144] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, and J. Ross, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, Dec. 2017.

[145] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *International Symposium on Computer Architecture*, June 2016.

[146] H. Kwon, A. Samajdar, and T. Krishna, "Rethinking nocs for spatial neural network accelerators," in *International Symposium on Networks-on-Chip*, Oct. 2017.

[147] B. Hong, Y. Ro, and J. Kim, "Multi-dimensional parallel training of winograd layer on memory-centric architecture," in *International Symposium on Microarchitecture*, Oct. 2018.

[148] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *International Conference on Archi-*

tectural Support for Programming Languages and Operating Systems (ASPLOS), Apr. 2019.

[149] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2014.

[150] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2017.

[151] F. Forster and S. Harl, "collectd." https://collectd.org/.

[152] R. Nishtala, V. Petrucci, P. Carpenter, and M. Själander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2020.