Learning-Based Quality Management for Approximate Communication in Network-on-Chips

Yuechen Chen, *Member, IEEE*, and Ahmed Louri, *Fellow, IEEE*

***Abstract*—Current multi/many-core systems spend large amounts of time and power transmitting data across on-chip interconnects. This problem is aggravated when data-intensive applications, such as machine learning and pattern recognition, are executed in these systems. Recent studies show that some data-intensive applications can tolerate modest errors, thus opening a new design dimension, namely, trading result quality for better system performance. In this article, we explore application error tolerance and propose an approximate communication framework to reduce the power consumption and latency of network-on-chips (NoCs). The proposed framework incorporates a quality control method and a data approximation mechanism to reduce the packet size to decrease network power consumption and latency. The quality control method automatically identifies the error-resilient variables that can be approximated during transmission and calculates their error thresholds based on the quality requirements of the application by analyzing the source code. The data approximation method includes a lightweight lossy compression scheme, which significantly reduces packet size when the error-resilient variables are transmitted. This framework results in fewer flits in each data packet and reduces traffic in NoCs while guaranteeing the quality requirements of applications. Our cycle-accurate simulation using the AxBench benchmark suite shows that the proposed approximate communication framework achieves 62% latency reduction and 43% dynamic power reduction compared to previous approximate communication techniques while ensuring 95% result quality.**

***Index Terms*—Accuracy management, approximate communication, network-on-chips (NoCs), reinforcement learning (RL).**

I. Introduction

APPROXIMATE communication leverages the error tolerance of the approximate computing application [1]–[5] to enhance communication efficiency in multicore systems [6]–[15]. A major problem faced by approximate communication is quality management: how can the accuracy of transmitted data be adjusted without missing the quality requirement on the application's results?

To solve this issue, quality management techniques are developed to control the accuracy of transmitted data by providing approximation information [9]–[13]. The approximation information contains an approximation indicator and approximation level. The approximation indicator identifies error-resilient data in the communication traffic; the approximation level specifies the error margin for data approximation. To ensure the application can yield useful results, the quality management technique: 1) predicts the quality loss in the result caused by approximating the data in the network and 2) chooses the approximation level for the data which predicts tolerable quality loss by the application. The key to effective quality management is to accurately predict the quality loss in the result, which requires a careful examination of the relationship between the data accuracy in packets and the result quality.

Current techniques [6]–[12], [14] rely on experienced program designers to manually identify the error-resilient variable in the source code for data approximation during communication. Moreover, they rely on programmers to predict the quality loss in the results and manually assign the approximation level for each error-resilient variable. These techniques face two limitations.

1) *Manually annotating the error-resilient variable limits the opportunity for packet approximation during communication:* Manual annotation can only approximate the variables that are explicitly defined by programmers but not intermediate variables. For example, when executing $d = a+(b\times c)$, a, b, c, and d can be approximated through manual annotation, but the product of b and c is not accessible by programmers. Moreover, as software engineers focus on the algorithm design and possess little information on on-chip traffic, the manually annotated variables only generate few approximable packets. As a result, the fully accurate data packets occupy more than 70% of the total communications, hindering the further improvement of the network performance. Fig. 1 shows the amount of approximated on-chip traffic using the techniques proposed in AxBA [12] and the quality control framework [11]. Manual annotation approximates 27% of the total on-chip traffic on average. However, the rest 73% of the data packets also have the potential of being approximated, which can be fully explored by automatic variable annotation.
2) *Predicting result quality loss involves intensive engineering work and limits the application of approximate communication:* The relation between the accuracy of transmitted data and the application output quality highly depends on the algorithm used by the applications.

Manuscript received April 17, 2020; revised June 12, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported by NSF under Grant CCF-1812495, Grant CCF-1565273, and Grant CCF-1953980. This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Yuechen Chen.)*

The authors are with the Department of Electrical and Computer Engineering, George Washington University Washington, DC, USA (e-mail: yuechen@gwu.edu; louri@gwu.edu).

Digital Object Identifier 10.1109/TCAD.2020.3012235

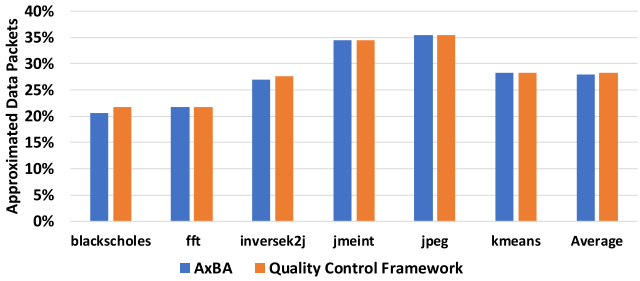

Fig. 1. Percentage of approximated data packets in communication traffic.

Current approximate computing applications contain multiple nonlinear functions, such as artificial neural networks, to meet the demand for solving complicated problems. For nonlinear functions, the relation between the accuracy of transmitted data and the application output quality also becomes nonlinear, which makes it difficult to predict the quality loss in the results. Thus, relying on human engineers to estimate the quality loss in results is neither accurate nor reliable.

Therefore, approximate communication frameworks require a quality management technique to automatically explore the opportunities for data approximation in an approximate computing application. Considering that different approximate computing application allows diverse types of error, in this article, we explore the use of reinforcement learning (RL) to automatically examine the error tolerance of applications and trade the result quality for less network latency and power consumption. The proposed technique includes a static code analyzer and an RL-assisted dynamic code analyzer. The static code analyzer extracts the algorithmic features of variables, which potentially affect the quality of results. The dynamic code analyzer monitors the cache miss to estimate the communication frequency of each variable. The RL evolves the optimal configuration of the approximation levels for variables, which can result in maximized communication efficiency yet ensures result quality. Specifically, the contributions of this work are as follows.

1) We propose an accuracy management technique that exploits RL to automatically annotate variables and adjust accuracy for the variables to improve the performance of on-chip interconnect while ensuring result quality.
2) We implement a static code analyzer and a dynamic code analyzer to extract the algorithmic and communication features of variables.
3) We conduct performance evaluation which shows the proposed framework reduces network latency and dynamic power by 36% and 46%, respectively, compared to AxBA [12], while ensuring a 95% result quality.

II. Motivation and Challenges

In this section, we first detail our motivation to use reinforcement learning (RL) in quality management and then present the challenges of implementing the proposed techniques.

A. Motivation

Use of Reinforcement Learning for Quality Management: The proposed automatic annotation for quality management has two objectives: 1) maximizing the data approximation and 2) ensuring the result accuracy. To achieve these two objectives, the machine learning (ML) algorithm chooses an approximation level for each variable based on the on-chip traffic pattern and the algorithm used in the application. ML algorithms are classified into three basic categories: 1) supervised learning; 2) unsupervised learning; and 3) RL [16].

As a supervised learning model, the algorithm learns on the labeled data set, providing an answer that ML can use to evaluate its accuracy on training data [17]. Therefore, supervised learning requires a labeled training data set to train the ML. Labeling training data requires the user to have the correct answer for a query, such as a query picture showing a car and the label "car" In our case, the query is to find the approximation level for each packet to maximize data approximation and ensure result accuracy. Considering that manual annotation limits the packet approximation and involves intensive engineering work, providing the correct answer to the query is impossible for programmers. Therefore, the supervised learning model is not suitable for quality management.

An unsupervised learning model, in contrast, eliminates the need for human involvement and uses ML to extract features and patterns from unlabeled data [18]. This ML is designed for exploring uncategorized data without a specific desired outcome or correct answer. However, because quality management requires achieving two desired objectives, exploring opportunities for approximation without the desired outcome fails to meet the design requirement.

RL involves training with a reward system, providing feedback when an artificial intelligence agent performs the best action in a particular situation [19]. The RL model explores the design space by randomly selecting an action in the action space. During variable annotation, RL can automatically explore the opportunity for data approximation by choosing an approximation level for each variable. The reward function can jointly analyze the intensity of the data approximation and the quality of the results to provide feedback on each action. The quality of the results can be analyzed by a quality check function, which is included in the approximate computing applications [20]. Therefore, RL can be implemented in quality management and replace inefficient manual annotation.

B. Challenges

Select Representative Features for an Application's Algorithm and Communication: Since RL relies on features to explore the opportunities for data approximation, the selection of representative features is critical to learning-based quality control. The relationship between the approximation level for each variable and the result quality depends on the algorithm implemented in an application. Thus, RL requires the algorithmic features to predict the impact on result accuracy when approximating a variable in an application. RL also needs to analyze on-chip communication pattern to enhance the performance of the interconnect. Considering that RL

annotates error-resilient variables in an application, the communication feature needs to represent the communication intensity for each variable. Therefore, choosing representative features for RL is a challenge for the proposed technique.

III. Features for Reinforcement Learning

A. Algorithmic Features of the Variables

The algorithmic features represent the relationship between the accuracy loss of each variable and the quality loss in the result, enabling RL to differentiate between variables in an application. The proposed variable algorithm features include position, previous operation, and next operation. The position of a variable is defined as the number of calculations that a variable requires to produce a result. The previous operation generates the variable. The next operation uses the variable as an argument.

We first investigate the relationship between the accuracy loss of the variables and the quality loss of the results by taking the operation (e.g., addition, subtraction, multiplication, division, etc.) into consideration. Equation (1) shows the case of $c = a \pm b$, where $\tilde{a}$ and $\tilde{b}$ are approximated and $\tilde{c}$ is the result. The previous operation of variable c is plus or minus

$$\tilde{c} = \tilde{a} \pm \tilde{b}. \tag{1}$$

Suppose that we use (2) (E_{ra} is the relative error) to measure the error of value a

$$E_{ra} = \frac{|a - \tilde{a}|}{a}. \tag{2}$$

From (2), we can obtain

$$E_{rc} = \frac{|c - \tilde{c}|}{c} \tag{3}$$

$$\tilde{a} = a + E_{ra} \times a \tag{4}$$

$$\tilde{b} = b + E_{rb} \times b. \tag{5}$$

Suppose that a and b have the same error ($E_{ra} = E_{rb} = E_{\text{rab}}$). We can obtain (6) by combining (1), (4), and (5)

$$\tilde{c} = (a + E_{\text{rab}} \times a) \pm (b + E_{\text{rab}} \times b). \tag{6}$$

By combining (3) and (6) with $c = a \pm b$, we observe that for $+$ and $-$ operation, the error tolerances for a and b are equal to the quality requirement on c.

Equation (7) describes the multiplication of $\tilde{a}$ and $\tilde{b}$, where $\tilde{a}$ and $\tilde{b}$ are approximated and $\tilde{c}$ is the result

$$\tilde{c} = \tilde{a} \times \tilde{b}. \tag{7}$$

By the same theory, we can obtain (8) for the multiplication operation, where a and b are fully accurate variables and E_{rab} is the relative error

$$\tilde{c} = (a + E_{\text{rab}} \times a) \times (b + E_{\text{rab}} \times b). \tag{8}$$

By combining (3) and (8) with $c = a \times b$, we find that for the multiplication operation, $E_{rc} = (1 + E_{\text{rab}})^2 - 1$. Therefore, $E_{rc} \leq$ error tolerance is ensured when $-1 + \sqrt{1 + E_{\text{rab}}} \leq$ error tolerance for the multiplication operation.

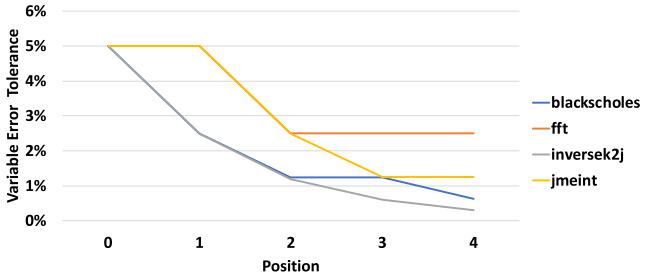

Fig. 2. Average error tolerance of variables versus its position. Application quality loss is 5%. Only variables with the same position value in the application are approximated.

Equation (9) describes the division of $\tilde{a}$ and $\tilde{b}$, where $\tilde{a}$ and $\tilde{b}$ are approximated and $\tilde{c}$ is the result

$$\tilde{c} = \tilde{a}/\tilde{b}. \tag{9}$$

For the division operation, we can derive (10), where a and b are fully accurate variables and E_{ra} and E_{rb} are the relative errors

$$\tilde{c} = (a + E_{ra} \times a)/(b + E_{rb} \times b). \tag{10}$$

By combining (3) and (10) with $c = a/b$, we find that for the division operation, $E_{rc} = |1 - (1 + E_{ra})/(1 + E_{rb})|$. $E_{rc} = 0$ when a and b have the same relative error ($E_{ra} = E_{rb} = E_{\text{rab}}$) for the division operation.

From these analyses, we conclude that for a variable c with different previous operations (e.g., addition, subtraction, multiplication, division, etc.), the error threshold for a and b is different when ensuring the same relative error on c. For example, when c can tolerate 5% relative error and the operation is addition or subtraction, both a and b can tolerate 5% relative error. If the operation is multiplication, both a and b can tolerate 2.5% relative error. Moreover, when a and b have different relative errors or the application uses different error metrics, the operation can affect the policy on adjusting the approximation level for each variable in the application. Therefore, when RL adjusts the approximation level for a variable, the operation can greatly affect the result quality.

Then, we investigate the relationship between the approximation level and result quality by taking the position into consideration. Equation (11) shows a typical function with two operations

$$f(a, b, c) = a + (b \times c) \tag{11}$$

where a needs one operation (addition) to generate the result while b and c require two operations (multiplication and addition). Therefore, the position values for a, b, and c are 1, 2, and 2, respectively. Equation (12) shows the case where a, b, and c are approximated. To make the example simple, we assume that the error is measured using (2) and, for a specific operation that the approximation levels of operands are equal

$$\tilde{f}(a, b, c) = \tilde{a} + (\tilde{b} \times \tilde{c}). \tag{12}$$

Based on the conclusion drawn, when the result error tolerance is E_r, a can tolerate E_r, and b and c can tolerate $-1 + \sqrt{1 + E_r}$. For example, the user can tolerate 5% result error, a can be

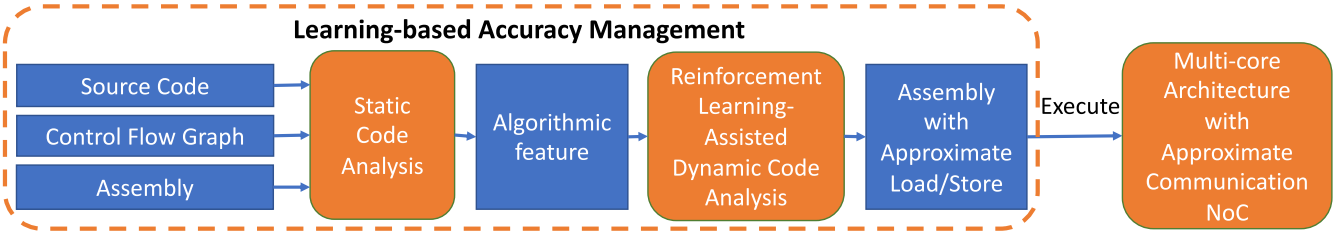

Fig. 3. High-level workflow of learning-based accuracy management.

approximated to 5% error, but b and c can be approximated only to 2.5% error. From this simple example, we observe that some operations (e.g., multiplication and division) can dramatically reduce the approximation level of operands. As the position value grows, the probability of these operations being used increases, which significantly affects the approximation level of the variables.

This effect can be verified by analyzing approximate computing applications. Fig. 2 shows the average error tolerance of the variables relative to its position in an application when the quality loss threshold is 5%. Since only variables with the same position value are approximated, the variable that stores the result (position 0) can be approximated to 5% error when the application can tolerate 5% quality loss. To avoid unsatisfactory result caused by error accumulation during the execution of applications, the variables that store the inputs (position $>$ 3) must attain a low error tolerance. Therefore, in Fig. 2, we observe that the error tolerance of the variables decreases as the position value increases. Fig. 2 also shows that for different applications, the variable error tolerance reduces at a different rate under the same position value interval. Since the position of a variable affects the relationship between the accuracy of the variable and the quality loss of the result, RL can use the position value to categorize variables and choose the approximation level accordingly.

Based on these observations, the relationship between the variable accuracy and the result quality loss depends on the position of the variable and the operation that the variable is involved in. Therefore the position, previous operation, and next operation of a variable are selected as the algorithmic features.

B. Communication Features of Variables

The goal of the proposed quality management technique is to maximize the number of approximated packets in a network to reduce network latency and power consumption. Because the approximated packets are generated by the core as it accesses error-resilient variables, RL needs the communication frequency for each variable to maximize data approximation in communication traffic. For example, in (12), b and c are fetched from two data sets with 100 floating point numbers each, and a is a constant number. Suppose each packet transmits only one floating-point number; 200 packets are required to transmit b and c, whereas a needs to be transmitted only once. Since approximating frequently transmitted variables introduces a large quantity of approximated packets, approximating b and c leads to better network performance compared to approximating a. Thus, the communication frequency for each variable is used as the communication feature in RL to analyze on-chip traffic patterns.

IV. Learning-Based Accuracy Management

The essence of the proposed technique is to automatically exploit the error tolerance of applications and trade data accuracy in the transmitted packets for network performance with an acceptable quality loss. Reducing data accuracy in the packets decreases the number of bits inside a packet, which lowers the power consumption and latency in a network-on-chip (NoC). The high-level workflow of the learning-based accuracy management technique is shown in Fig. 3 and consists of two steps: namely, static code analysis and RL-assisted dynamic code analysis. First, the compiler compiles the source code and generates the assembly code and the control flow graph (CFG), which are delivered to the static code analyzer to extract the algorithmic features of the variables. After that, the dynamic code analyzer extracts the communication feature by counting the cache misses for the variables in the application. Then, based on these features, the RL in the dynamic code analyzer categorizes the load/store operations and automatically chooses the approximation level to maximize the data approximation while ensuring the result quality. The dynamic code analyzer replaces the load/store operations in the assembly code with approximate load/store operations, which contain the approximation level chosen by the RL. Finally, the assembly code with the approximate load/store operations is executed in the multicore architecture with the approximate communication NoC. We present the process of static code analysis in Section IV-A. Then, we discuss the design and the operation of the RL-assisted dynamic code analysis in Section IV-B. In Section IV-C, we discuss the lossy data compression based on a given approximation level.

A. Static Code Analysis

Fig. 4 shows the working process of the static code analyzer with an example. The static code analyzer analyzes the assembly code with CFG and the source code to extract the algorithmic features of the variables. After the source code is compiled, the compiler generates the CFG and the assembly code. Notably, the compiler can break down the complex function in the source code and generate a CFG that describes the function in terms of basic operations, such as addition, subtraction, multiplication, and division. To extract the algorithmic features for each variable, the code analyzer first identifies the

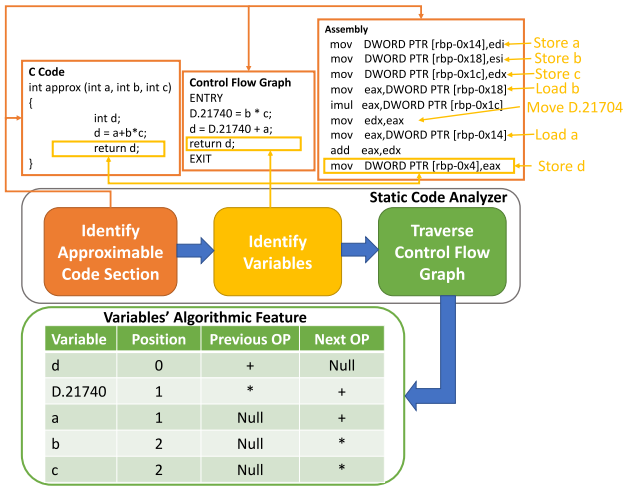

Fig. 4. Static code analyzer workflow. The blue arrow indicates the workflow of the code analyzer. The inputs of the code analyzer are the *C* code, control flow graph, and assembly. The output of the analyzer is the algorithmic features of the variables, which are highlighted in the green box. (The source code is compiled by the GNU C compiler (GCC) for the X86 instruction set architecture. The control flow graph and assembly code are generated by GCC. D.21740 is an intermediate variable created by the compiler.)

approximable code section in the *C* code, CFG, and assembly, which are highlighted in orange in Fig. 4. Then, the code analyzer identifies all variables and results in the source code, CFG, and assembly code, which is highlighted in yellow. In this case, we can see that a new variable is created by the compiler (D.21740), which is also treated as a variable. The code analyzer traverses the CFG and obtains the position value, previous operation, and the next operation for each variable.

B. Reinforcement Learning-Assisted Dynamic Code Analysis

RL is an ML approach wherein the agent acts as a learner and a decision maker by interacting with the environment [19]. Fig. 5 shows the dynamic interaction between the RL agent and the environment. In ①, the agent selects an action a_t from a set of actions, $A = \{1, \ldots, K\}$ at time step t. In ②, the selected action influences the environment by affecting the internal state s_t and the reward r_t. In ③, the effect eventually results in a new state and reward, s_{t+1} and r_{t+1}, respectively, at the next time step $t + 1$.

In RL, the goal of the agent is to interact with the environment by selecting actions in a manner that maximizes the long-term total rewards R, which is the cumulative sum of all future rewards. The future rewards are discounted by a factor γ called the discount factor. As γ approaches 0, the agent begins to consider only the current rewards.

In this article, we use a tabular Q-learning to estimate the long-term reward. A Q-value table is initialized with random values for all possible (s, a) pairs. At each time step, the Q-learning chooses actions based on the current Q so that over many time steps, all actions are taken in all states. In each time step, the action-value table entry $Q(s, a)$ is updated using (13) based on action a, reward R, and new state s'

$$Q(s, a) = Q(s, a) + \alpha\big[R + \gamma * \max Q(s', a) - Q(s, a)\big]. \tag{13}$$

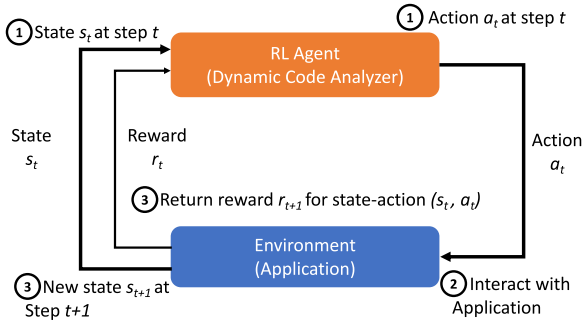

Fig. 5. Agent-environment interaction in RL.

Category	Feature	Description
Algorithmic Feature	1. Position	Number of operation until result
	2. Previous Operation	The operation generated this variable
	3. Next Operation	The next operation of this variable
Communication Feature	4. Communication Frequency	Number of packets generated by this variable
Data Approximation	5. Approximation Level	Variable's current approximation level

Fig. 6. State attributes used in RL.

1) Action Space: The action space consists of different operations on the approximation level that the dynamic code analyzer can choose from. Since approximate communication techniques have different data approximation methods, the range of approximation levels are different for each approximate communication design. The design of action space needs to consider that different approximate communication NoC supports differing ranges of approximation levels. Because a large action space leads to an exponential increase in training time, a scalable design with small action space is required for the proposed quality management technique. Therefore, we design the RL algorithm with three actions $A = \{a_0, a_1, a_2\}$, which represent the reducing approximation level, the maintaining approximation level, and the increasing approximation level, respectively. With the proposed action space, the proposed quality management technique can be applied to approximate communication techniques with different ranges of approximation levels while maintaining a small action space. Since RL requires choosing the approximation level for each variable after dynamic code analysis, a new feature is needed to store the current approximation level.

2) State Space: A state s is a vector of the characteristics of a variable, which consists of the application's algorithmic features, communication feature, and approximation level, as shown in Fig. 6. State attributes 1–3 indicate the relation between variable accuracy and result quality. State attribute 4 indicates the communication frequency of a variable. State attribute 5 indicates the current approximation level when loading or storing the variable. The total number of states depends on the number of variables used in an application.

3) Reward Function: The RL agent uses the reward function to evaluate the benefit due to the action of a given state.

Algorithm 1 Reward Function

```
function R (State s_t, Action a_t)
    if Result Error >Error Threshold and a_t != a_0 then
        return -1
    end if
    if s_t.Approx Level == Max level and a_t == a_2 then
        return -1
    end if
    if s_t.Approx Level == Min Level and a_t == a_0 then
        return -1
    end if
    if a_t == a_0 then
        return N_a × (s_t.Approx Level − 1)
    end if
    if a_t == a_1 then
        return N_a × s_t.Approx Level
    end if
    if a_t == a_2 then
        return N_a × (s_t.Approx Level + 1)
    end if
end function
```

In this article, the goal of the RL agent is to enhance the NoC performance by approximating more packets while ensuring the result quality. Algorithm 1 shows the reward function used to calculate the reward for a given state (s_t) and action (a_t). In this algorithm, N_a and Approx Level represent the total number of approximated packets and the approximation level, respectively. To ensure that the result quality and the selected approximation level are within the acceptable range, the agent receives -1 reward under three circumstances: 1) when the result error exceeds the assigned error threshold and the agent fails to reduce the approximation level (a_0); 2) when the approximation level reaches the upper limit (Max Level) and the agent continues to increase the approximation level (a_2); and 3) when the approximation level reaches the lower limit (Min Level) and the agent continues to decrease the approximation level (a_0). Otherwise, the reward function returns the product of the total number of approximated packets and the selected approximation level after an action.

4) Learning Parameters: In RL, the learning rate α and the discount rate γ can be tuned. The learning rate α can be reduced over time and determines how well Q-learning will converge. It can be shown that for appropriate values of α, Q-learning converges to the optimal action-value function and its corresponding optimal policy [19]. As α approaches 0, the agent focuses more on old information. As a result, the trained model can achieve high accuracy but needs more steps to converge. On the other hand, as α approaches 1, the agent focuses more on the most recent information. Therefore, the RL converges in a small amount of time with low accuracy. Since the RL in the proposed technique is trained offline, the accuracy is more important than the speed. Thus, 0.1 is used as the learning rate (α).

The variable γ (where $0 \leq \gamma \leq 1$) in this equation is the discount rate, which determines the impact of future rewards on the total return: as γ approaches 1, the agent becomes less near sighted by giving more weight to future rewards After extensive experimentation, we selected γ as 0.8 to achieve the maximum network performance while ensuring result accuracy. A detailed discussion of how the parameter γ impacts approximation effectiveness is presented in Section V-F.

5) Working Process of Dynamic Code Analysis: The RL-assisted dynamic code analysis has three working phases: 1) the communication feature extraction phase; 2) the training phase; and 3) the testing phase.

During the communication feature extraction phase, the dynamic code analysis tool runs the application by feeding in part of its query data (training query data). The dynamic code analyzer counts the cache misses for each variable in the assembly code to extract the communication feature.

During the training phase, the dynamic code analysis tool runs the application by feeding in the training query data. When a cache miss occurs during the loading/storing of a variable from/to memory, the dynamic code analyzer sends the variable's features to the RL model. RL selects an action from the action space and updates the approximation level of the variable. Then, the code analyzer continues running the application with the updated approximation level for the variable. When the application finishes processing the training query data, $Q(s, a)$ is updated using Algorithm 1. This process is repeated several times until the Q-learning algorithm converges. A detailed discussion of how various applications effect the training epoch is presented in Section V-F.

After the training phase, the application is fed a small amount of its query data (testing query data). When a cache miss occurs while loading/storing a variable, the dynamic code analyzer replaces the instruction with an approximate load/store and sends the algorithmic feature and current approximation level to the RL model. RL adjusts the approximation level for the approximate load/store operation. The application is run by the code analyzer until no further approximation level adjustment is performed. In other words, RL takes action a_1 (maintaining approximation level) for all error-resilient variables in the application. Finally, the application with the approximate load/store is executed in the multicore architecture with the approximate communication NoC.

Approximate load and store are new types of instructions for approximate communication in NoCs that are used to approximate data packets. We replace the conventional *mov* with an approximate move instruction (*amov* dist, src, and approximation level) in the X86 instruction set so that the network can identify the approximable data in the data packets. In the next section, we discuss how the network approximates packets using the *amove* instructions.

C. Data Approximation Method

Since this work mainly focuses on accuracy management, we use the idea of data truncation from [14] and [8] to perform the data approximation for both the floating point and integer values. Fig. 7 presents a high-level overview of the approximate communication NoC design. We modify the baseline network interface (NI) to include the following additional components: the data approximation logic and the data recovery logic. The data approximation logic truncates the data

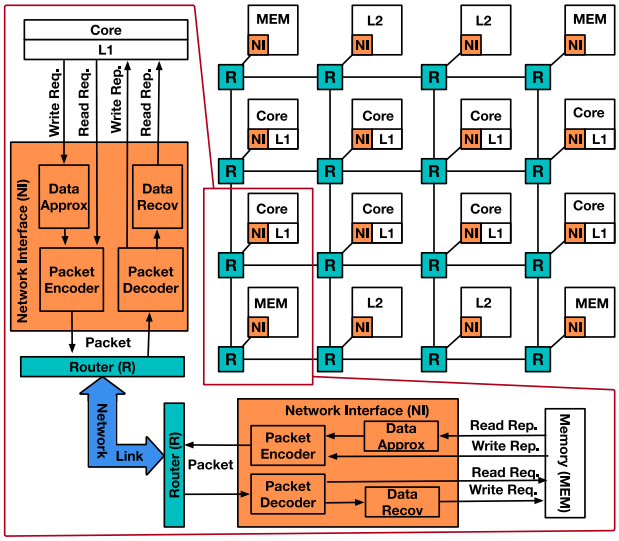

Fig. 7. Multicore architecture with approximate communication NoC. Data approximation logic (Data Approx) truncates the data based on the approximate level before carrying out packet encoding. Data recovery logic (Data Recov) recovers the truncated data.

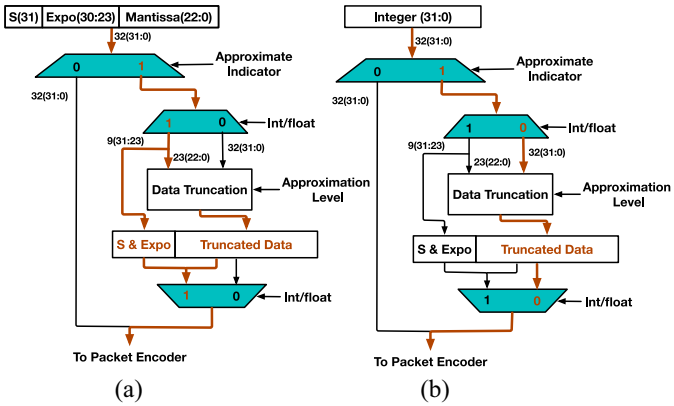

Fig. 8. Data approximation logic design. Data truncation logic removes the least significant bits based on the approximate level and data type (Int/float). Sign (S) Exponent (Expo).

when an approximate load/store misses the cache. Different from conventional load/store instructions, the approximate load/store contains an approximation level for the lossy data compression mechanism to truncate the data. Notably, the proposed quality management technique can also be applied to other approximate communication NoC designs with different approximation level ranges and approximation methods.

When an L1 cache miss is caused by an approximate store operation, a write request is generated by the L1 cache with the approximation information. The approximation information includes the address, data type (int/float), and approximation level. The address and data type identify the error-resilient data in the write request. The data in the write request are truncated by the data approximation logic at the core and L1 cache node based on the approximation level. Then, the write request is encoded by the packet encoder and injected into the network. When the shared cache or memory node receives the write request packet, the data recovery module recovers the truncated data and adds zeros to the truncated part to maintain the original data format. Then, the write request is sent to the shared cache or memory, and a write reply is generated and transmitted to the core and L1 cache node to confirm the transmission.

When an L1 cache miss is caused by approximate load operation, a read request is issued by the L1 cache with the approximation information. Then, the read request is sent to the packet encoder to generate a read request packet. When the shared cache or memory node receives the read request, the approximation information is extracted from the packet and registered at the NI. When the read reply is generated by the shared cache or memory, the data approximation logic checks the approximation information and truncates the data in accordance with the approximation information. Then, the packet encoder prepares the read reply packet and injects it into the network. When the read reply packet arrives at the core and L1 cache node, the data recovery module recovers the data.

TABLE I
RELATIONSHIP BETWEEN THE DATA ERROR THRESHOLD AND THE APPROXIMATION LEVEL

Data Error Threshold	Approximation Level
0.125	10
0.03125	9
0.0078125	8
0.001953125	7
0.000488281	6
0.00012207	5
3.05176E-05	4
3.05176E-05	3
7.62939E-06	2
4.76837E-07	1
0	0

1) Data Approximation Logic: Table I elucidates the relationship between the data error threshold and the approximation level. This data approximation logic supports 11 approximation levels ranging from level 0 to level 10. Level 0 represents the data that are transmitted with 100% accuracy while level 10 represents data that can tolerate a 12.5% relative error. The higher the approximation level is, the lower the data accuracy and the smaller the packet size.

Equation (14) shows the definition of data error threshold, where $\tilde{a}$ is approximated a and E_{ra} is relative error

$$E_{ra} = \frac{|a - \tilde{a}|}{a} \leq \text{data error threshold.} \tag{14}$$

Equations (15) and (16) show the representation of single precision floating point value based on IEEE 754 standard [21]

$$\text{float} = (-1)^S \times \text{mantissa} \times 2^{\text{exp}} \tag{15}$$

$$\text{mantissa} = 2^0 + \sum_{k=1}^{23} X_k 2^{-k} \ (X_k = 0 \text{ or } 1). \tag{16}$$

Based on (15) and (16), the mantissa always starts with one.

According to the IEEE 754 standard [21], when a data point is represented in the floating-point format, the first bit of the mantissa is omitted. We observe that when c bits (of the 23-b mantissa) are protected, the maximum relative error on this floating-point data value will be $\sum_{k=c+1}^{23} 2^{-k}$, which is less

than 2^{-c} according to the sum of the geometric sequence ($\sum_{k=1}^{n} ar^{k-1} = a(1-r^n)/1-r$, where a is the first term, n is the number of terms, and r is the common ratio in the sequence). Therefore, using (16), we can deduce the following expression for the data error tolerance:

$$\text{error tolerance} = 2^{-n} \ (1 \leq n \leq 23). \tag{17}$$

In (17) above, the data error tolerance is a number between 0 and 1, and n is the number of most significant bits (MSBs) in the mantissa of this floating-point value. In a floating-point data value, the 1-b sign and the 8-b exponent (a total of 9 b) are also critical bits, which must be transmitted. Thus, by truncating $23-n$ b, we can ensure the value's relative error is less than 2^{-n}. For example, to satisfy a data error tolerance of 4% (approximation level = 9) for any floating-point value, we can truncate 18 least significant bits (LSBs), resulting in a maximum relative error of 3.12%.

Fig. 8(a) highlights the data truncation process when approximating floating point values. First, the data approximation logic extracts the sign bit and exponent bits from a floating point value using a demultiplexer. Then, the least significant bits of the mantissa are truncated according to the approximation level. After that, the sign and exponent are added to the MSB of the truncated data. Finally, the approximated floating point value is sent to the packet encoder.

Equation (18) shows the representation of a signed integer. In a signed integer, the MSB represents the sign, and the remaining 31 b represent the value

$$\text{int} = \sum_{k=0}^{31} X_k 2^k \ (X_k = 0 \text{ or } 1). \tag{18}$$

We observe that when n bits (of the 31 LSBs) are truncated, the maximum error caused by truncation will be $\sum_{k=0}^{n} X_k 2^k$ ($X_k =$ 0 or 1). Thus, we can use (19) to calculate the number of bits (n) to be truncated for a given error tolerance

$$\text{error tolerance} = \frac{\sum_{k=0}^{n} X_k 2^k}{\sum_{k=0}^{31} X_k 2^k} \ (X_k = 0 \text{ or } 1). \tag{19}$$

Fig. 8(b) highlights the data truncation process when approximating integer values. The data approximation logic directly sends the data to the data truncation module when an integer value needs approximation. Then, the data truncation module eliminates the least significant bits of the integer according to the approximation level. Finally, the multiplexer selects the truncated integer and sends the approximated data to the packet encoder.

2) Data Recovery Logic: Fig. 9 shows the data recovery logic design. Data recovery has two steps. First, the demultiplexer selects approximated data based on the approximate indicator. Then, the data recovery logic recovery the data by filling the truncated least significant bits with zeros. Finally, the recovered data are sent to core or memory.

V. Evaluation and Analysis

A. Experimental Methodology

We evaluate the performance of the learning-based quality management technique using the GEM5 simulator [22] and the AxBench benchmark suite [23]. The GEM5 simulator is modified to support data truncation by integrating the approximate data approximation logic and data recovery logic. We obtain the network latency from the statistics profiled by GEM5 when running AxBench. We use DSENT [24] to capture the dynamic power consumption of the network. The detailed settings for the GEM5 simulator are shown in Table II.

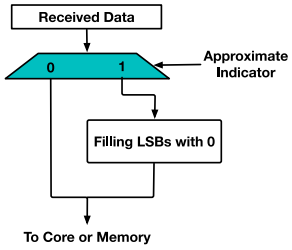

Fig. 9. Data recovery logic design.

TABLE II
Simulation Environment Setup

NoC Parameters	Network Type: Garnet 2.0 Topology: 8 × 8 2D mesh Link Width: 64 bits Routing Algorithm: X-Y Routing
System Parameters	64 on-chip cores @2 GHz 32 kB L1 instruction cache 32 kB L1 data cache 4-way associative 64-bank fully shared 16 MB L2 cache
Target Result Quality	95%
Approximate Communication Techniques Used For Evaluation	AxBA [12] Quality Control Framework [11] Proposed Framework

We implement the proposed technique, including static code analysis and dynamic code analysis, in Python. Table III shows data used for RL training/testing and the evaluation metrics utilized to measure the result quality.

We evaluate the proposed technique by comparing with AxBA [12] and the quality control framework [11] from three perspectives: 1) approximation effectiveness; 2) network latency; and 3) dynamic power consumption. AxBA includes an accuracy management technique that requires the program designer to manually annotate the approximable values and their error tolerances. The quality control framework includes an accuracy management scheme that allows the network to adjust the data accuracy after the result quality is measured. To fairly compare the performances and effectiveness of the quality control techniques, all the approximate communication frameworks use data truncation with 11 approximation levels as the packet approximation method.

B. Approximation Effectiveness

In this article, the effectiveness of the approximate communication framework is defined by

$$E = N_a \times \bar{L}. \tag{20}$$

In (20), E is the approximation effectiveness, N_a represents the percentage of approximated data packets, and $\bar{L}$ shows the

TABLE III
AXBENCH BENCHMARK SUITE [23]

Benchmark	Query Data Size	Training Query Data Size	Testing Query Data Size	Evaluation Metric
blackscholes	64k floating point (fp) values	100 floating point (fp) values	11 floating point (fp) values	Average relative error
fft	5k random fp numbers	128 fp values	16 fp values	Average relative error
inversek2j	100k random (x,y) points	100 (x,y) points	10 (x,y) points	Average relative error
jmeint	10k pairs of 3D triangles	100 Boolean values	10 Boolean values	# of mismatches
jpeg	512 * 512 pixel image	32*32 pixel image	8*8 pixel image	Average pixel diff.
kmeans	512 * 512 pixel image	32*32 pixel image	8*8 pixel image	Average pixel diff.

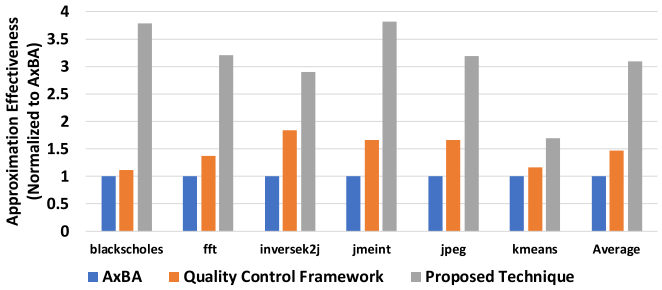

Fig. 10. Effectiveness of the approximate communication framework. The results are normalized with respect to AxBA (higher is better). The quality loss threshold is 5%.

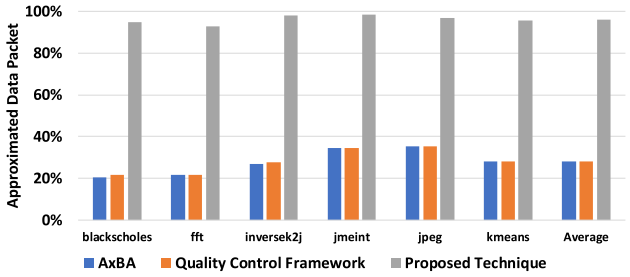

Fig. 11. Approximated data packets. The quality loss threshold is 5%.

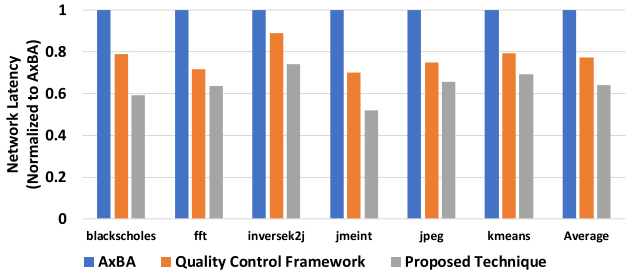

Fig. 12. Network latency. The results are normalized with respect to AxBA (lower is better). The quality loss threshold is 5%.

average approximation level of the packets. An effective quality control method can approximate more packets compared to less effective methods with the same data approximation method. As shown in Fig. 10, the proposed quality control technique can approximate more data packets than AxBA and quality control framework. The quality control methods used in previous approximate communication frameworks rely on a human engineer to identify error-resilient variables, which limits the number of approximated data packets. On the other hand, the proposed technique discovers error-resilient variables through code analysis, making it 3× and 2× more effective than AxBA and the quality control framework, respectively. The largest effectiveness is achieved (3.9×) when the jmeint benchmark is executed with the proposed quality management technique.

The approximation effectiveness can be further verified by analyzing the percentage of approximated data packets. As shown in Fig. 11, the proposed framework can approximate 95% of the data packets on average, whereas AxBA and the quality control framework can approximate only 27% and 28% of the data packet, respectively. The proposed technique achieves the best approximation effectiveness by approximating most of the data packets.

C. Network Latency

Fig. 12 shows the evaluation results for the average network latency normalized with respect to AxBA [12]. The network latency is defined as the number of clock cycles elapsed between the injection of a packet at the source node and the successful delivery of the packet to the destination. The packet injection process includes data truncation and the packet encoding process at the source NI. The packet ejection process includes packet decoding and data recovery at the destination NI. We compare the proposed technique with AxBA and the quality control framework. As shown in Fig. 12, the proposed approximate communication framework achieves an average network latency reduction of 36% compared to AxBA. The largest network latency reduction in the experiment is achieved for the jmeint benchmark (49% reduction) while the smallest network latency improvement is obtained for inversek2j (26%). The proposed technique can achieve this large latency reduction on the jmeint benchmark due to the high approximation effectiveness. With the code analyzer, the proposed technique is 3.9× more effective than AxBA on the jmeint benchmark, as seen from Fig. 10. The main contributor to the propagation of approximated packets is automatic annotation. As a result, the proposed framework is able to achieve latency reductions of 41%, 37%, 35%, and 31% on the blackscholes, fft, jpeg, and kmeans benchmarks, respectively, compared with AxBA. These results show that ML is more efficient in selecting the approximation level to fully utilize the error tolerance of the application than having a programmer select the approximation level.

D. Dynamic Power Consumption

Fig. 13 shows the amounts of dynamic power consumed by the NoC when different accuracy management techniques are

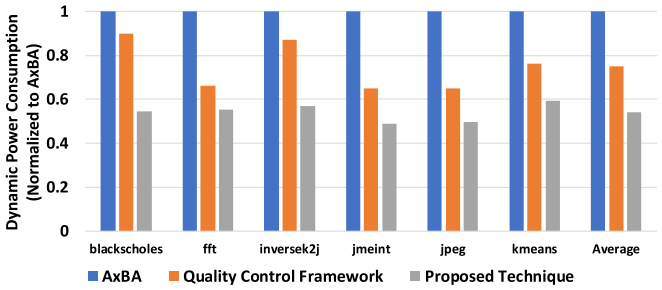

Fig. 13. Dynamic power consumption. The results are normalized with respect to AxBA (lower is better). The quality loss threshold is 5%.

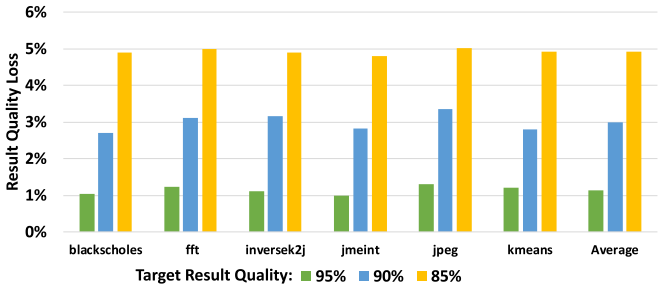

Fig. 14. Application quality loss. The quality loss thresholds are 5%, 10%, and 15%.

(a) (b)

Fig. 15. Jpeg benchmark result comparison. The quality loss threshold is 5%. The result difference is 1.3%.

used. The dynamic power consumption includes the dynamic power consumed by both the NI and NoC. As shown in this figure, the proposed framework achieves an average dynamic power reduction of 46% compared with AxBA [12]. The largest dynamic power reduction in this experiment is achieved on the jmeint benchmark (52% reduction) while the lowest dynamic power improvement is obtained for inversek2j (44% reduction). For the same reason mentioned in the network latency analysis, the proposed framework is able to significantly reduce the number of flits per packet while ensuring the quality of the result. Therefore, the dynamic power consumption of the proposed framework is reduced by 45%, 44%, 50%, and 40% on the blackscholes, fft, jpeg, and kmeans benchmarks, respectively, compared with AxBA.

E. Result Quality

The application quality loss is measured using the application-specific metrics provided in Table III. Fig. 14 shows the application quality loss for different benchmarks with different target result quality. When the target result quality is 95%, the average quality loss is 1.14% and the quality loss for all the benchmarks is less than 5%. The approximate computing application can tolerate these errors caused by approximate communication. Fig. 15 compares the accurate results obtained on the jpeg benchmark with the approximate results yielded by the proposed framework. The difference between the two outputs is negligible and unrecognizable by human vision. When the targeted result quality is reduced to 90% and 85%, the average quality loss is 2.98% and 4.92%, respectively. From Fig. 14, the quality loss for all the benchmarks is less than 10% and 15%, when the target result quality is 90% and 85%, respectively. These results indicate that the proposed quality control technique restrains the quality loss in an acceptable amount for all the benchmarks.

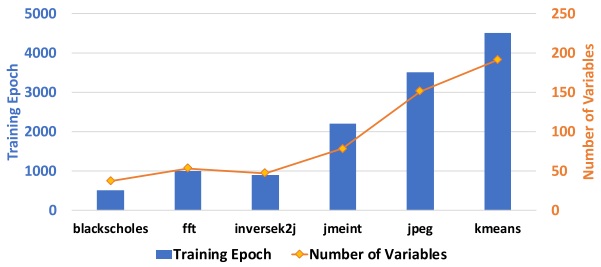

Fig. 16. Number of epochs versus the number of variables in the applications.

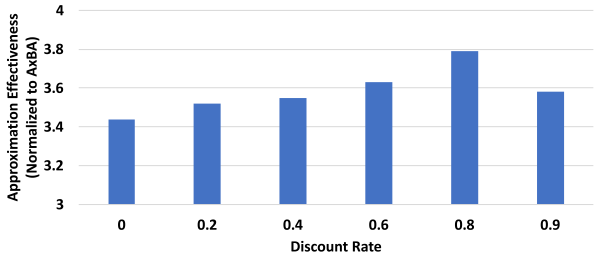

Fig. 17. Impact of discount rate (γ) on approximation effectiveness. The results are normalized to AxBA for the blackscholes benchmark. The quality loss threshold is 5%.

F. Sensitivity Analysis

Application: Fig. 16 shows the number of epochs to train different RL for different applications. The increase of variables in the application expands the Q table, therefore increases the number of epochs. In Fig. 16, we can see that as the number of variables increases in an application, the training epoch also increases. Especially, more training epoch is needed for the benchmarks that have quality metrics other than relative error. However, because training and testing processes are performed before the execution of the application, a high training epoch does not affect the running of the application. Moreover, due to the advancement of the specialized accelerator and heterogeneous architecture, the execution time for training and testing can be significantly reduced.

Impact of Discount Rate (γ): We discuss the impact of discount rate (γ) on the approximation effectiveness. Fig. 17 shows the discount rate and approximation effectiveness for the blackscholes benchmark. The discount rate (γ) determines the impact of future rewards on the total return. As γ approaches 1, the agent becomes less near-sighted by giving more weight to future rewards. Fig. 17 shows that the approximation effectiveness improves initially with a larger γ.

However, aggressively increasing γ can also lead to Q-learning failing to converge, which negatively affects the system performance. The best performance is achieved when γ equals 0.8.

VI. Related Work

Approximate Communication Techniques: Various approximate communication techniques have been proposed to enhance the performance of NoCs [6]–[15], [25], [26]. Betzel *et al.* [13] conducted a survey on three promising techniques, namely, compression, relaxed synchronization, and value prediction, to address communication bottleneck issues in massively parallel systems for approximate computing applications. Boyapati *et al.* [6] and Stevens *et al.* [12] proposed lossy data compression techniques to further reduce the size of error resilience data before packet injection. Wang *et al.* [7] and Xiao *et al.* [10] proposed reducing network congestion by dropping data in a packet before injecting it into the network. Xiao *et al.* [10] explored the application's error threshold and proposed an approximation method for dropping data accordingly.

Quality Control Techniques: In the approximate communication framework, a quality management system ensures that the data error can be tolerated by the approximate computing application [27]–[29]. The proposed approximate communication techniques [6], [7], [10], [12], [14] include a software-based quality management framework, which allows a program designer to assign the error threshold. Significant approximation errors can be eliminated during approximate computations when a lightweight result checking system, such as Rumba [20], is used. ApproxIt [30] proposes a runtime quality calibration scheme to control the quality of an approximate computing application with an iterative method. Approxilyzer [31] provides a solution enabling a quality management system to quantify the quality impact of a single-bit error. Laurenzano *et al.* [32] suggested that the result error can be controlled by managing the input error. However, these quality control frameworks require the program designer to specify the approximable variables, which limits the approximate communication technique in further improving NoC performance. The proposed quality control framework determines the error-resilient value through a code analysis, which further enhances NoC performance with an acceptable quality loss.

Machine Learning Applied to Computer Architecture Designs: The field of computer architectural design using ML has grown significantly in the past few years [33]. Mandal *et al.* [34], [35], Kim *et al.* [36], and Xiao *et al.* [37] used ML to manage on-chip resources, such as the number of active cores, voltage/frequency level, for heterogeneous architecture. Reinforcement learning and imitation learning are used for real-time policy adjustment to achieve better system performance. Xiao *et al.* [37] used RL to map tasks to the specialized accelerator for maximum system performance. ML in an NoC dynamically explores different design spaces, such as links, the topology, the error mitigation policy, etc., and addresses many design tradeoffs [38]–[44]. For example, IntelliNoC [41] uses RL to control the dynamic error mitigation in NoCs. Savva *et al.* [38] used artificial neural networks to control the link between two routers. Approximate communication creates a new dimension for NoC designs, namely, trading data quality for more efficient on-chip communication, raising new design spaces for ML to explore. In this article, we explore the possibility of using RL to manage data quality.

VII. Conclusion

In this work, we proposed a learning-based accuracy management technique for power-efficient and low-latency NoCs. The proposed framework uses RL to automatically explore the error tolerance of an application and automatically adjust the data accuracy in a packet. RL reduces the latency and power consumption of an NoC by approximating variables that require frequent transmission while meeting the quality requirements of the application. We compared the proposed technique with previously proposed accuracy management methods, such as AxBA and the quality control framework. Our detailed evaluation showed that the proposed accuracy management scheme is $3\times$ and $2\times$ better (in terms of the approximation effectiveness) than AxBA and the quality management framework, respectively. The proposed technique reduces the dynamic power consumption and network latency by 46% and 36%, respectively, compared to AxBA.

Acknowledgment

The authors sincerely thank the reviewers for their helpful comments and suggestions.

References

[1] T. Yeh, P. Faloutsos, M. Ercegovac, S. Patel, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Chicago, IL, USA, Dec. 2007, pp. 394–406.

[2] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Vancouver, BC, Canada, 2012, pp. 449–460.

[3] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surveys*, vol. 48, no. 4, pp. 1–33, Mar. 2016.

[4] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb. 2016.

[5] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. 18th IEEE Eur. Test Symp. (ETS)*, Avignon, France, 2013, pp. 1–6.

[6] 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 *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Toronto, ON, Canada, 2017, pp. 666–677.

[7] L. Wang, X. Wang, and Y. Wang, "ABDTR: Approximation-based dynamic traffic regulation for networks-on-chip systems," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, Boston, MA, USA, 2017, pp. 153–160.

[8] M. F. Reza and P. Ampadu, "Approximate communication strategies for energy-efficient and high performance NoC: Opportunities and challenges," in *Proc. Great Lakes Symp. VLSI*, New York, NY, USA, 2019, pp. 399–404.

[9] V. Fernando, A. Franques, S. Abadal, S. Misailovic, and J. Torrellas, "Replica: A wireless manycore for communication-intensive and approximate data," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS'19)*, New York, NY, USA, 2019, pp. 849–863.

[10] S. Xiao, X. Wang, M. Palesi, A. K. Singh, and T. Mak, "ACDC: An accuracy- and congestion-aware dynamic traffic control method for networks-on-chip," in *Proc. IEEE Design Autom. Test Eur. Conf. Exhibit. (DATE)*, Florence, Italy, 2019, pp. 630–633.

[11] Y. Chen and A. Louri, "An online quality management framework for approximate communication in network-on-chips," in *Proc. ACM Int. Conf. Supercomput.*, New York, NY, USA, 2019, pp. 217–226.
[12] J. R. Stevens, A. Ranjan, and A. Raghunathan, "AxBA: An approximate bus architecture framework," in *Proc. Int. Conf. Comput.-Aided Design*, New York, NY, USA, 2018, pp. 1–8.
[13] F. Betzel, K. Khatamifard, H. Suresh, D. J. Lilja, J. Sartori, and U. Karpuzcu, "Approximate communication: Techniques for reducing communication bottlenecks in large-scale parallel systems," *ACM Comput. Surveys*, vol. 51, no. 1, pp. 1–32, Jan. 2018.
[14] Y. Chen, M. F. Reza, and A. Louri, "DEC-NoC: An approximate framework based on dynamic error control with applications to energy-efficient NoCs," in *Proc. IEEE 36th Int. Conf. Comput. Design (ICCD)*, Orlando, FL, USA, Oct. 2018, pp. 480–487.
[15] Y. Chen and A. Louri, "An approximate communication framework for network-on-chips," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 6, pp. 1434–1446, Jun. 2020.
[16] Y. Kodratoff, *Introduction to Machine Learning*. St. Louis, MO, USA: Elsevier, 2014.
[17] M. Kubat, *An Introduction to Machine Learning*, vol. 2. Cham, Switzerland: Springer, 2017.
[18] T. Hofmann, "Unsupervised learning by probabilistic latent semantic analysis," *Mach. Learn.*, vol. 42. nos. 1–2, pp. 177–196, 2001.
[19] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[20] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Portland, OR, USA, Jun. 2015, pp. 554–566.
[21] *IEEE Standard for Floating-Point Arithmetic*, IEEE Standard 754-2008, 2008.
[22] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[23] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design Test*, vol. 34, no. 2, pp. 60–68, Apr. 2017.
[24] C. Sun *et al.*, "DSENT—A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *Proc. IEEE/ACM 6th Int. Symp. Netw. Chip (NOCS)*, Copenhagen, Denmark, May 2012, pp. 201–210.
[25] A. B. Ahmed, D. Fujiki, H. Matsutani, M. Koibuchi, and H. Amano, "AxNoC: Low-power approximate network-on-chips using critical-path isolation," in *Proc. 12th IEEE/ACM Int. Symp. Netw. Chip (NOCS)*, Turin, Italy, Oct. 2018, pp. 1–8.
[26] V. Y. Raparti and S. Pasricha, "DAPPER: Data aware approximate NoC for GPGPU architectures," in *Proc. 12th IEEE/ACM Int. Symp. Netw. Chip (NOCS)*, Oct. 2018, pp. 1–8.
[27] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proc. ACM Int. Conf. Object Orient. Program. Syst. Lang. Appl. OOPSLA '14*, 2014, New York, NY, USA, pp. 309–328.
[28] T. Wang, Q. Zhang, and Q. Xu, "ApproxQA: A unified quality assurance framework for approximate computing," in *Proc. Conf. Design Autom. Test Eur. (DATE '17)*, Leuven, Switzerland, 2017, pp. 254–257.
[29] C. Xu *et al.*, "On quality trade-off control for approximate computing using iterative training," in *Proc. 54th Annu. Design Autom. Conf. DAC '17*, New York, NY, USA, 2017, pp. 1–6.
[30] Q. Zhang and Q. Xu, "Approxit: A quality management framework of approximate computing for iterative methods," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 5, pp. 991–1002, May 2020.
[31] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Taipei, Taiwan, 2016, pp. 1–14.
[32] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, New York, NY, USA, 2016, pp. 161–176.
[33] D. D. Penney and L. Chen, "A survey of machine learning applied to computer architecture design," 2019. [Online]. Available: arXiv:1909.12373.
[34] S. K. Mandal, G. Bhat, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "An energy-aware online learning framework for resource management in heterogeneous platforms," *ACM Trans. Design Autom. Electron. Syst.*, vol. 25, no. 3, p. 28, May 2020.
[35] S. K. Mandal, G. Bhat, C. A. Patil, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "Dynamic resource management of heterogeneous mobile platforms via imitation learning," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 12, pp. 2842–2854, Dec. 2019.
[36] R. G. Kim *et al.*, "Imitation learning for dynamic VFI control in large-scale manycore systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 9, pp. 2458–2471, Sep. 2017.
[37] Y. Xiao, S. Nazarian, and P. Bogdan, "Self-optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 6, pp. 1416–1427, Jun. 2019.
[38] A. G. Savva, T. Theocharides, and V. Soteriou, "Intelligent on/off dynamic link management for on-chip networks," *J. Elect. Comput. Eng.*, vol. 2012, p. 6, Jan. 2012.
[39] H. Zheng and A. Louri, "An energy-efficient network-on-chip design using reinforcement learning," in *Proc. 56th Annu. Design Autom. Conf. (DAC)*, New York, NY, USA, 2019, pp. 1–6.
[40] K. Wang, A. Louri, A. Karanth, and R. Bunescu, "High-performance, energy-efficient, fault-tolerant network-on-chip design using reinforcement learning," in *Proc. Design Autom. Test Eur. Conf. Exhibit. (DATE)*, Florence, Italy, Mar. 2019, pp. 1166–1171.
[41] 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 *Proc. ACM/IEEE 46th Int. Symp. Comput. Archit. (ISCA)*, Phoenix, AZ, USA, 2019, pp. 589–600.
[42] K. Wang and A. Louri, "CURE: A high-performance, low-power, and reliable network-on-chip design using reinforcement learning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2125–2138, Sep. 2020.
[43] K. Wang, H. Zheng, and A. Louri, "TSA-NoC: Learning-based threat detection and mitigation for secure network-on-chip architecture," *IEEE Micro*, early access, Jun. 19, 2020, doi: 10.1109/MM.2020.3003576.
[44] H. Zheng and A. Louri, "Agile: A learning-enabled power and performance-efficient network-on-chip design," *IEEE Trans. Emerg. Topics Comput.*, early access, Jun. 18, 2020, doi: 10.1109/TETC.2020.3003496.

Yuechen Chen (Member, IEEE) received the master's degree in electrical engineering from George Washington University, Washington, DC, USA, in 2016, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering.

His research interests include approximate computing and network-on-chips.

Ahmed Louri (Fellow, IEEE) received the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 1988.

In August 2015, he joined the George Washington University, Washington, DC, USA, is the David and Marilyn Karlgaard Endowed Chair Professor of electrical and computer engineering, where he is also the Director of the High Performance Computing Architectures and Technologies Laboratory (https://hpcat.seas.gwu.edu/Director.html). From 1988 to 2015, he was a Professor of electrical and computer engineering with the University of Arizona, Tucson, AZ, USA. From 2010 to 2013, he served as a Program Director in the National Science Foundation's Directorate for Computer and Information Science and Engineering. He conducts research in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks and network on chips for multicores, and the use of machine learning techniques for energy-efficient, reliable, high-performance, and secure many-core architectures and accelerators.

Dr. Louri was recently selected to be the recipient of the IEEE Computer Society 2020 Edward J. McCluskey Technical Achievement Award, for pioneering contributions to the solution of on-chip and off-chip communication problems for parallel computing and manycore architectures. He is currently serving as the Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTERS.