Machine Learning Approaches for Efficient Design Space Exploration of Application-Specific NoCs

YONG HU, MARCEL METTLER, and DANIEL MUELLER-GRITSCHNEDER, Chair of EDA, Technical University of Munich
THOMAS WILD and ANDREAS HERKERSDORF, Chair of Integrated Systems, Technical University of Munich
ULF SCHLICHTMANN, Chair of EDA, Technical University of Munich

In many Multi-Processor Systems-on-Chip (MPSoCs), traffic between cores is unbalanced. This motivates the use of an application-specific Network-on-Chip (NoC) that is customized and can provide a high performance at low cost in terms of power and area. However, finding an optimized application-specific NoC architecture is a challenging task due to the huge design space.

This article proposes to apply machine learning approaches for this task. Using graph rewriting, the NoC Design Space Exploration (DSE) is modelled as a Markov Decision Process (MDP). Monte Carlo Tree Search (MCTS), a technique from reinforcement learning, is used as search heuristic. Our experimental results show that—with the same cost function and exploration budget—MCTS finds superior NoC architectures compared to Simulated Annealing (SA) and a Genetic Algorithm (GA). However, the NoC DSE process suffers from the high computation time due to expensive cycle-accurate SystemC simulations for latency estimation. This article therefore additionally proposes to replace latency simulation by fast latency estimation using a Recurrent Neural Network (RNN). The designed RNN is sufficiently general for latency estimation on arbitrary NoC architectures. Our experiments show that compared to SystemC simulation, the RNN-based latency estimation offers a similar speed-up as the widely used Queuing Theory (QT). Yet, in terms of estimation accuracy and fidelity, the RNN is superior to QT, especially for high-traffic scenarios. When replacing SystemC simulations with the RNN estimation, the obtained solution quality decreases only slightly, whereas it suffers significantly when QT is used.

CCS Concepts: • **Hardware → Software tools for EDA**; **Network on chip**; • **Networks → Network on chip**; • **Computing methodologies → Neural networks**;

Additional Key Words and Phrases: Application-specific networks-on-chip, design space exploration, Monte-Carlo-tree search, recurrent neural network

Authors' addresses: Y. Hu, M. Mettler, D. Mueller-Gritschneder, and U. Schlichtmann, Chair of EDA, Technical University of Munich, Arcisstr. 21, Munich, Germany, 80333; emails: {yong.hu, marcel.mettler, daniel.mueller, ulf.schlichtmann}@tum.de; T. Wild and A. Herkersdorf, Chair of Integrated Systems, Technical University of Munich, Arcisstr. 21, Munich, Germany, 80333; emails: {thomas.wild, herkersdorf}@tum.de.

1084-4309/2020/08-ART44 $15.00
https://doi.org/10.1145/3403584

ACM Reference format:
Yong Hu, Marcel Mettler, Daniel Mueller-Gritschneder, Thomas Wild, Andreas Herkersdorf, and Ulf Schlichtmann. 2020. Machine Learning Approaches for Efficient Design Space Exploration of Application-Specific NoCs. *ACM Trans. Des. Autom. Electron. Syst.* 25, 5, Article 44 (August 2020), 27 pages.
https://doi.org/10.1145/3403584

1 INTRODUCTION

With an increasing device density, more and more processing elements (PEs) can be integrated in one Multi-Processor System-on-Chip (MPSoC). On the one hand, this brings opportunities for implementing more complex applications; but, on the other hand, it also brings challenges for the design of the on-chip interconnect. To guarantee quality of service (QoS), interconnects are required to provide not only high bandwidth to support the heavy on-chip communication but also low latency for real-time system response. Facing such challenges, Networks-on-Chip (NoCs) have emerged as an attractive solution to replace traditional bus systems due to their better scalability. PEs are connected with NoC routers and communicate through data packets on the network. NoCs can be used for both general-purpose and application-specific MPSoCs. General-purpose MPSoCs are designed to support a wide range of different applications, and communication demand between PEs is mostly unknown at design time. Fair access among PEs is desired, and homogeneous NoCs with regular topologies are beneficial, e.g., the mesh topology used in both the Teraflops research chip of Intel [41] and the RAW processor from MIT [5]. However,application-specific MPSoCs, e.g., for smartphone chips, make use of a large number of fixed-function PEs such as Video Decoders or DMAs. For such systems, the network traffic is highly unbalanced. This motivates to customize the NoC according to the needs of the application to obtain a better performance, e.g., lower latency, while requiring less power and area.

Generally, there exist two types of approaches for the synthesis of application-specific NoCs: top-down approaches and iterative approaches. The top-down approaches start from the system specification and make decisions step-by-step based on heuristics. E.g., if two PEs communicate with each other heavily, they will be connected to the same router. However, top-down approaches have a common drawback: The top-level decisions are made before obtaining the complete NoC architecture, so performances and costs can only be estimated with high-level models, for which the estimation accuracy cannot be guaranteed. Iterative design space exploration (DSE) approaches start from a given or a population of given NoC architectures. They iteratively explore new NoC architectures in the design space and search for an optimal tradeoff between performance and cost. The huge size of the NoC design space challenges these approaches, hence, calling for efficient exploration heuristics. Besides, designers often want to be able to track and control the changes made during the DSE process. But existing heuristics such as Genetic Algorithms (GAs) make it hard for the designer to track the decisions applied.

This article applies machine learning approaches in an iterative NoC DSE. We use the Monte Carlo Tree Search (MCTS) heuristic to explore the NoC design space. MCTS is widely used in reinforcement learning. It structures the explored space as a tree, so modifications from the initial NoC to the optimal NoC can be back-traced step-by-step. More importantly, MCTS balances exploration and exploitation very well using the Upper Confidence Bound for Trees (UCT) function. This is important to improve exploration efficiency, avoid local minima, and identify delayed rewards, which occur when the beneficial effect of a modification does not show up immediately but only becomes visible after several other modifications. However, even when using highly efficient MCTS, the NoC DSE still suffers from the high computational cost of expensive cycle-accurate SystemC simulations, which are required to estimate the NoC latencies. To eliminate this bottleneck,

this article proposes to use a Recurrent Neural Network (RNN) to replace the SystemC simulation used to estimate the latency of NoC traffic flows. The designed RNN is trained using the TensorFlow framework [16] and Scikit-Learn library [36]. The trained RNN has sufficient generality to be used for latency estimation for arbitrary new NoC architectures. To summarize, the contributions of this article are the following:

- We model the NoC DSE as Markov Decision Process (MDP) using graph rewriting for modifying NoC architectures. MCTS is applied as the search heuristic upon the MDP. Our experimental results show that—with the same cost function and exploration budget—MCTS finds superior NoC architectures compared to Simulated Annealing (SA) and a Genetic Algorithm (GA).
- An RNN is developed to replace the SystemC simulation for estimating the latency of NoC flows. It is well designed for generalization. So it is trained only once and can then be used on any arbitrary NoC architecture. Experiments compare the RNN with SystemC simulation (SC) and an analytical model based on Queuing Theory (QT). In terms of estimation accuracy and fidelity, the RNN is superior to QT, especially for high-traffic scenarios. The fidelity measures the difference between the ordering of estimated values and the ordering of actual values. It is an important metric for guiding a DSE. Our RNN-based DSE is as fast as QT-based DSE and both are much faster than SC-based DSE. Meanwhile, the quality of the result of the RNN-based DSE is less than SC-based DSE, but much better than the quality of the result of QT-based DSE. RNN, hence, allows to trade in less quality for same speed-up compared to QT, making it a very competitive latency estimation method for NoC DSE.

The article is organized as follows: Section 2 discusses related work. Section 3 introduces the background on MCTS and RNNs. Section 4 describes our MCTS-based NoC DSE with graph rewriting. Section 5 describes the design of the RNN for NoC latency estimation and its training process. Section 6 presents our NoC DSE framework with multiple evaluation tools and DSE algorithms. Section 7 gives the experimental results on our proposed MCTS heuristic and the trained RNN.

2 RELATED WORK

Application-specific NoC synthesis has been the focus of several research works, which we classify either as an iterative or top-down approach. Besides, this section also introduces existing NoC evaluation models as well as related machine learning approaches.

Iterative approaches: The method in Reference [26] uses Tabu search for application-specific NoC synthesis. It starts from an initial solution and iteratively explores its neighborhood. Explored designs are marked Tabu to avoid cycles in the search. The approach in Reference [22] proposes a multi-commodity flow (MCF)-based scheme to find the optimal NoC architecture that minimizes power consumption under communication latency constraints. It also provides an approximation algorithm, which is reported to be much faster than the commercial LP solver CPLEX. The approach in Reference [11] applies a GA to explore the NoC design space. It models the connections in the topology as endpoint pairs and these endpoints are regarded as genes. The connections are modified randomly to generate new designs by mutation of these genes. The work in Reference [39] uses an SA algorithm to explore the NoC design space. The approach filters out designs with large delay to support hard timing constraints in QoS systems. The work in Reference [32] also applies SA in the design space exploration and it further models the memory system including caches.

Top-down approaches: The method in Reference [40] uses spectral clustering to assign PEs to routers. In a subsequent step, the routing between PEs is determined by the A* algorithm. A similar approach with a different clustering algorithm was presented in Reference [42]. The assignment

of PEs to routers is a key decision for application-specific NoCs. In these approaches, this decision is done based on a cost function that combines latency, bandwidth, and floorplan information to find the best cluster of PEs. Yet, no simulation can be used at this level of abstraction to test the decision.

NoC evaluation models: the methods in References [33] and [27] apply Queuing Theory to estimate flit latency and buffer usage for each router. They model each output channel as a server and its corresponding input channels as clients. Using the application communication graph and routing paths, they obtain the flit arrival rate at each input channel, which corresponds to the request rate of each client. Then, Queuing Theory is used to calculate average queue length and average waiting time of each request, which is the estimated latency of each flit. However, Queuing Theory can not capture the burstiness and data dependency in the communication, which limits its accuracy. The research in Reference [29] proposes Max-plus Algebra to estimate end-to-end delay in NoCs. Alternatively, Reference [37] suggests Network Calculus to estimate worst-case end-to-end delay, and Reference [8] proposes stochastic Network Calculus theory for latency estimation. In addition to analytical models, many SystemC-based NoC simulators have been developed, including the Booksim simulator [31] by Stanford and the Noxim [7] by University of Catania. They are cycle-accurate, but also require a lot of computation time. Besides, they provide limited support for application-specific NoC architectures.

Machine learning approaches in NoC synthesis: The Orion 3.0 tool [25] uses least-square regression to learn the post-P&R power and area from parametric models that are automatically created under different router configurations. This tool is widely used to estimate the power and area of various NoC architectures. The work in Reference [28] uses regression methods to estimate the NoC latency. But it can only be used for mesh or torus NoCs. It reuses the Booksim simulator configurations as its input parameters and also uses Booksim to determine the golden network latency. The work in Reference [12] uses the STAGE machine learning algorithm to optimize the placement of planar and vertical communication links in 3D-NoCs for energy efficiency. It iteratively applies a base search and a meta search. The base search runs in a greedy way to find local optima and generate new training data. The meta-search learns from features of the training data and tries to explore a good start state for the base search. The features consist of average hop count, bandwidth-hop product, and clustering coefficient. The work in Reference [30] uses reinforcement learning to optimize the task allocation in MPSoCs.

3 BACKGROUND

This section introduces the background of MCTS and RNNs, which form the basis of the presented NoC DSE approach.

3.1 MDPs and MCTS

MCTS is widely used in many game engines, including the famous AlphaGo [14] and AlphaGo Zero [15]. It is a heuristic search algorithm used for solving Markov decision processes (MDPs) [6]. An MDP is a process description model for sequential decision problems. The complete description consists of the following components:

- S: a finite set of states s that represents the design space;
- $Q(s)$: a reward function describing the quality of a state s;
- $A(s)$: the set of available actions, which consists of all actions that can be applied on state s;
- $f(s, a)$: a state transition function, which returns one output state $s' = f(s, a)$ when applying the action $a \in A(s)$ on the state s.

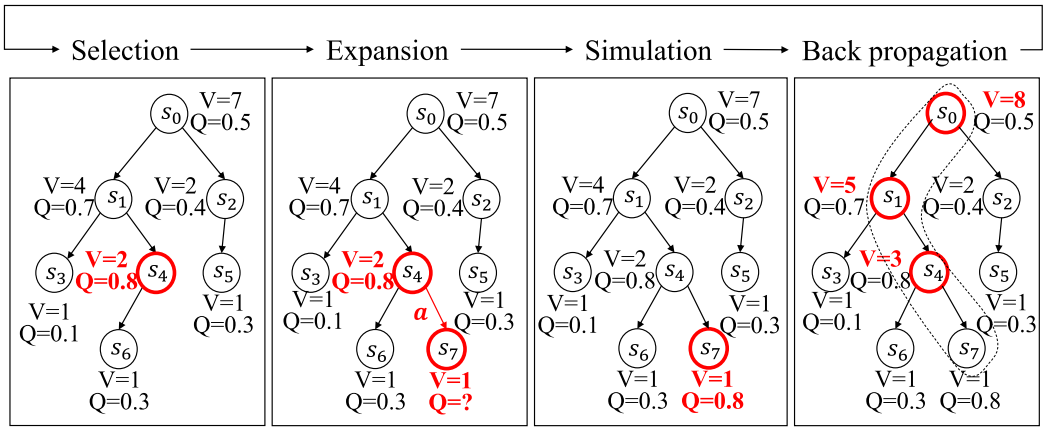


Fig. 1. MCTS iterative exploration.

The solution to an MDP is the sequence of actions on the state that yields the highest reward given a specific initial state s_0. MCTS is an efficient method to achieve this goal. It structures the exploration of the MDP as a search tree T. Each state s is a node on the tree. The MCTS heuristic explores the MDP space iteratively. It always investigates a sub-tree that is rooted in the so-called root node. At the beginning, the root node is set to be the initial node s_0 and the tree is empty. When an action $a \in A(s)$ is applied on s, the output $s' = f(s, a)$ will be added as a child of node s and the action a is stored on the directed edge from parent node s to the child node s'. Besides, each node s also stores the reward function $Q(s)$ and its visit count $V(s)$. $V(s)$ has an initial value of 1. Each time a new child is added as a successor, $V(s)$ will increase by one. Figure 1 shows a sub-tree that already has several explored nodes. A new child is added to the sub-tree following four basic steps:

(1) **Selection:** The node that is most urgent for expansion will be selected. The urgency of a node s is defined by the Upper Confident Bound for Trees (UCT) as

$$UCT(s) = Q(s) + 2C_p\sqrt{\frac{\ln(V(s_0))}{V(s)}}. \tag{1}$$

Larger UCT values indicate higher urgency. The parameter C_p is used to adjust the tradeoff between exploration and exploitation. By default, $C_p = \frac{1}{\sqrt{2}}$. In the example in Figure 1, the node s_4, which has the highest UCT value, is selected for expansion.

(2) **Expansion:** One of the available actions $a \in A(s_4)$ is applied on the node s_4 selected above. This creates the new node $s_7 = f(s_4, a)$. The newly created node is added as a child to the selected node. Then, $V(s_7)$ is initialized as one, but the quality $Q(s_7)$ is still unknown.

(3) **Simulation:** The newly created node is evaluated to compute its quality, e.g., in the example $Q(s_7) = 0.8$.

(4) **Backpropagation:** The reward of the newly created node is back-propagated following the search tree till the root node is reached. This step triggers all predecessor nodes to increase their visit count by one. E.g., the set of predecessor nodes of s_7 consists of s_4, s_1, and s_0.

The MCTS keeps adding and exploring new child nodes for the current root node following the above steps until a certain termination condition is fulfilled. The goal of the root node exploration

is to find the one action on the root node that yields the highest future reward. This is equal to finding the root node's direct child node that yields the highest future reward. There exist multiple approaches to compare the future reward. One is based on the visited count, and the direct child with the highest $V(s)$ is then returned. A second approach is based on $Q(s)$. We return the direct child node, which is the parent of the node with the overall highest reward $Q(s)$ encountered during the exploration of the root node. Using this approach, MCTS is able to discover late reward gains, which are not obtained in direct children (one action applied) of the root node but in grandchildren and even their successors. At some point, the search should focus on the more rewarding parts of the search tree. This is achieved by applying the action with the highest future reward on the root node, which then leads to updating the root node as the direct child node with highest future reward. Nodes that belong to children of other direct children, hence, are not further explored, but the exploration only focuses on the children in the sub-tree of the new root node.

3.2 Recurrent Neural Networks (RNNs)

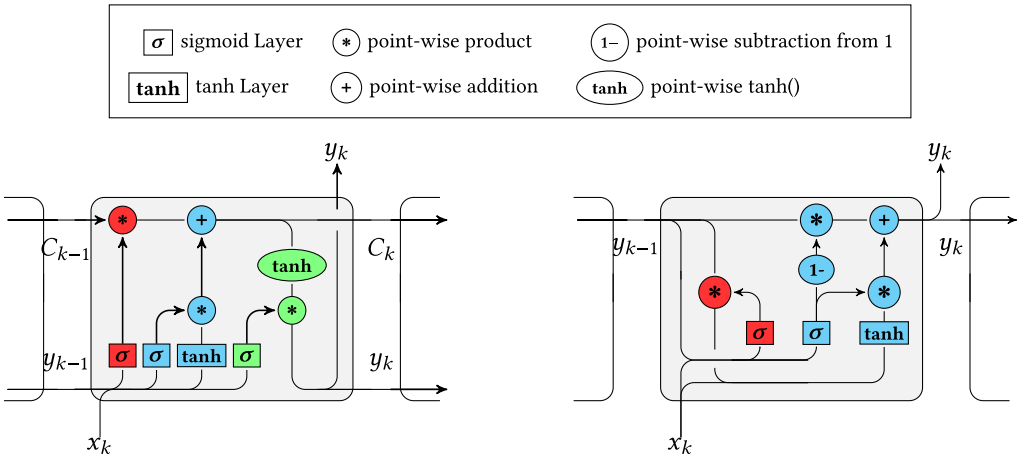
In machine learning, sequences of data are typically processed by Recurrent Neural Networks (RNNs). The outputs of the RNN neurons depend not only on the current inputs but also on their previous outputs. Hence, RNNs can be trained to process the dependency of an input sequence of vectors $\mathbf{X} = [\mathbf{x}_1 \quad \cdots \quad \mathbf{x}_k \quad \cdots \quad \mathbf{x}_K]$, in which $\mathbf{x}_k$ is the kth input vector. For each input vector $\mathbf{x}_k$, the RNN generates one scalar as output Y_k. Depending on the application, either the complete sequence of outputs is used or only one element of the output sequence is used, e.g., the last output Y_K.

3.2.1 RNN Training. To train an RNN, cycles in the connections of the RNN must be removed by unrolling RNN layers. Each RNN layer is unrolled to K layers, where K is the RNN input sequence length. Those K layers can be treated as a feed-forward neural network that has equal weights and biases. Then, the training methods for feed-forward networks can be reused for RNNs. The details of those training methods are available in Reference [35], which also addresses the vanishing and exploding gradient problem. This is a well-known problem in machine learning, which occurs in the training of neural networks with a large number of hidden layers. For this reason, it is difficult to train RNNs with long input sequences. To address this problem, the Long Short Term Memory (LSTM) architecture [18] and the Gated Recurrent Unit (GRU) architecture [9] are proposed. Both architectures outperform the traditional RNN and achieve competitive results for different applications [10]. They are briefly outlined in the following:

3.2.2 Long Short Term Memory (LSTM). The LSTM network, which is introduced in Reference [18], is a special RNN architecture for the learning of long-term dependencies. In contrast to traditional RNNs, the LSTM network replaces the network layers by LSTM cells. The structure of an unrolled LSTM cell is illustrated in Figure 2(a), where the rectangles represent the network layers, the circles represent the point-wise operations, the joining arrows represent the concatenation of data, and the diverging arrows represent the duplication of data.

The core idea of the LSTM cell is about the data flow path, illustrated as the horizontal arrow at the top of Figure 2(a), which goes through every unrolled cell without passing a network layer. On this path, the number of network layers in an unrolled LSTM cell is independent of the input sequence length K. As a result, the vanishing and exploding gradient problem will not occur even for long input sequences, which allows the cell to learn long-term dependencies.

Additionally, three more gates have been added: the forget gate, the input gate, and the output gate. As shown in Figure 2(a), the forget gate, illustrated in red, evaluates the amount of data that should be discarded from the cell state. The input gate, illustrated in blue, evaluates the amount



(a) The structure of an unrolled LSTM cell, where the forget gate is illustrated in red, the input gate is illustrated in blue and the output gate is illustrated in green [34].

(b) The structure of an unrolled GRU, where the reset gate is illustrated in red and the update gate is illustrated in blue [34].

Fig. 2. Structures of an unrolled LSTM cell and an unrolled GRU cell.

of data that should be added to the current cell state. Furthermore, the output gate, illustrated in green, was added to extract the part of the cell state that is relevant for the cell output.

3.2.3 Gated Recurrent Unit (GRU). An alternative architecture that prevents the vanishing and the exploding gradient problem is the GRU. As introduced in Reference [9], the GRU is a variation of the LSTM cell. In this architecture, the cell state and the output gate, as well as the input and the forget state, are combined. This leads to a simple cell structure shown in Figure 2(b). A GRU consists of two gates: the reset gate and the update gate. The reset gate, illustrated in red, specifies how much information of the previous output state is relevant for the processing of the current input $\mathbf{x}_k$. The update gate, illustrated in blue, combines the operations of the input and the forget gate of the LSTM cell. Here, the Sigmoid-layer simultaneously defines the amount of information that will be added and the amount of information that will be discarded from the output state y_{k-1}. Thus, the cell only discards information from the output when new information is added.

4 MCTS-BASED NOC DESIGN SPACE EXPLORATION

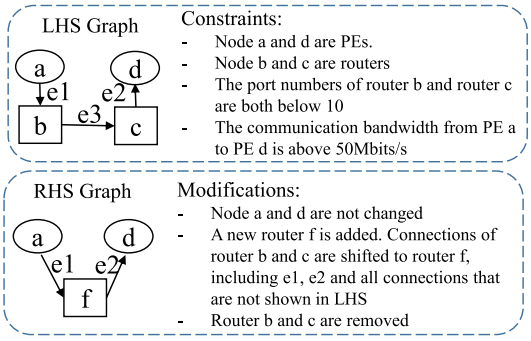
This section introduces graph rewriting, which is applied to model the NoC DSE process as an MDP. MCTS is applied as the search heuristic to solve the MDP.

4.1 Traffic Specification and Latency Constraints

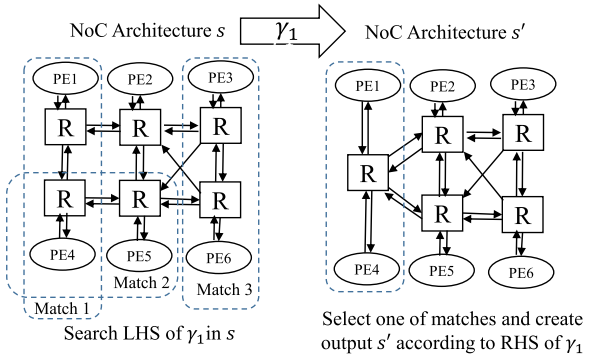
The specification of the traffic requirements for a specific MPSoC is a set of 4-tuples as follows:

$$(PE_{s,n}, PE_{d,n}, BW_n, Y_{ub,n}) \text{ with } n = 1 \ldots N, \tag{2}$$

where $PE_{s,n}$ is the source PE of flow n, $PE_{d,n}$ the destination PE, BW_n the bandwidth, and $Y_{ub,n}$ the upper bound of the average latency of flow n. $Y_{ub,n}$ is optional and only set for a subset of critical flows $n \in f_{crit}$ that have strict latency requirements. For example, often flows from processors need to be serviced fast to avoid performance penalties resulting from the processor waiting on data.



(a) Graph Rewriting Rule γ_1



(b) Apply the Rule γ_1 on NoC architecture s

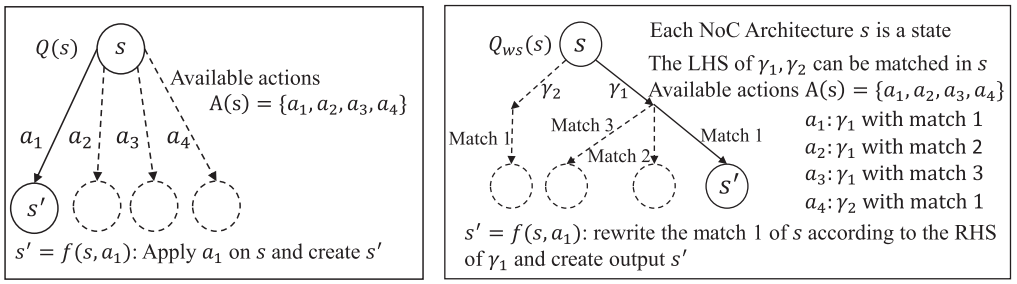
Fig. 3. Illustration of graph rewriting for NoCs.

4.2 Graph Rewriting of NoCs

The NoC architecture can be well described with a graph: PEs and routers are represented by nodes, and links are represented by directed edges between nodes, as shown in Figure 3(b). A modification of the NoC architecture can be formulated using the theory of *graph rewriting* [23]. Graph rewriting is based on rules, which consist of a left-hand-side (LHS) graph and a right-hand-side (RHS) graph. The LHS graph describes a region of interest (ROI), where to modify the graph. The LHS can be defined not only based on the graph structure, but also with some property constraints such as node degree, the distance between two nodes, the name or the property of a node. The RHS describes how to modify the ROI. The modifications can also be made on both the graph structure and its properties. A rule execution on a so-called design graph is a three-step process:

(1) In the *match* step, we search for the LHS in the design graph. All matches between LHS and design graph are stored. For a single rule there can be several matches.
(2) In the *selection* step, one match is chosen.
(3) In the *apply* step, the selected matching sub-graph in the design graph is replaced by the RHS graph of the rule.

Simply stated, we replace a sub-graph of the design graph by a new sub-graph. One example of the graph rewriting rule γ_1 for a NoC modification is illustrated in Figure 3. The LHS of γ_1 includes two PEs a and d and two routers b and c. There needs to exist a connection from a to b, b to c, and c to d for the LHS to match. Besides, router b and c are constrained to have less than 10



(a) General MDP (b) NoC DSE with graph rewriting

Fig. 4. MDP model for NoC DSE using graph rewriting.

ports, and the communication from a to d should be above 50 $Mbits/s$. Since two PEs communicate heavily and the connected routers are still not big, we can merge two routers into one to reduce communication latency. This modification is described in the RHS of γ_1 in Figure 3(a). The rule γ_1 is applied on an example NoC design graph in Figure 3(b). During the match step, we look for the LHS of γ_1 in the design graph using graph isomorphism. In this example, three matches for γ_1 are marked in the design graph. Match 1 is chosen, which consists of the PE1, PE4, and their connected routers. In the apply step, the design graph is modified according to the RHS of γ_1 to generate a new design graph representing a new NoC architecture.

4.3 MDP Model for NoC DSE using Graph Rewriting

As shown in Figure 4(a), a general MDP model consists of four components: a set S of states s, a set of available actions for each state $A(s)$, the state transition function $f(s, a)$, and a reward function $Q(s)$ for each state. For our NoC DSE, each NoC architecture represents one state s. The reward function $Q(s)$ describes the quality of the state s, which depends on the MDP goal. The goal of NoC DSE is to minimize multiple objectives, including the NoC latency $g_1(s)$, power $g_2(s)$, and area $g_3(s)$ while keeping the upper bound average latency constraint on the critical flows $n \in f_{crit}$: The reward function $Q_{ws}(s)$ combines all these factors together using the weighted-sum method with a penalty term for the constraints, where Y_n corresponds to the latency of a critical flow $n \in f_{crit}$:

$$Q_{ws}(s) = -1 \cdot \left(\lambda_1 \cdot \frac{g_1(s)}{g_1(s_0)} + \lambda_2 \cdot \frac{g_2(s)}{g_2(s_0)} + \lambda_3 \cdot \frac{g_3(s)}{g_3(s_0)} + \lambda_4 \cdot \max_{n \in f_{crit}} \frac{\max(Y_n - Y_{ub,n}, 0)}{1 \text{ cycle}} \right). \quad (3)$$

The three objectives are normalized by the respective values for the initial solution s_0. However, the reduction of NoC latency is usually at the expense of increasing power and area, so it is not possible to have minimal values of $g_1(s)$, $g_2(s)$, and $g_3(s)$ at the same time. Hence, the weight parameters λ_1, λ_2, λ_3 allow to give certain priority to the different objectives, depending on the designer choice. Additionally, the penalty term for violating the upper average latency constraint of critical flows is included directly into the reward function. A constraint is violated when the term $\max(Y_n - Y_{ub,n}, 0) > 0$. Normalization by 1 cycle just removes the unit. To be independent of the number of critical flows, we only evaluate the worst-case violation found by the second max function over all critical flows. The fourth weight factor λ_4 is used to set the priority between the optimization of the three main objectives and the penalty for constraint violations.

The two remaining MDP components, available actions $A(s)$ and the state transition function $f(s, a)$, are implemented using graph rewriting. As shown in Figure 4(b), each NoC architecture s is a state. A graph rewriting rule γ can be applied on the NoC architecture s only if there exists a

match between the LHS of the rule γ and the NoC architecture s. So an action $a \in A(s)$ is defined as a graph rewriting rule γ together with one selected match in the NoC architecture s. Meanwhile, the state transition function with $s' = f(s, a)$ is implemented as applying the selected match in the NoC architecture s with the RHS of the rule γ and creating the output NoC architecture s'.

In this work, we encode basic NoC modifications using graph rewriting rules, including shifting a PE to a different router, adding/removing a router, and adding/removing a directed link between a pair of routers. Any other kind of modification can always be achieved with a combination of these basic rules.

ALGORITHM 1: MCTS Based NOC DSE.

Input: Initial NoC architecture s_0
Output: Optimal NoC architecture s^*

$s_{root} \leftarrow s_0$;
Simulation: evaluate s_0 area, power and latency of flows and calculate $Q_{ws}(s_0)$
for l = 1 to L **do**
  while within computation budget **do**
    Selection: $s \leftarrow \arg\max_{s \in T_{s_{root}}} UCT(s)$ // $T_{s_{root}}$ is the tree where the root is s_{root}
    Expansion: apply $a \in A(s)$ on s and creates $s' \leftarrow f(s, a)$
      the s' is added as a child of s and $V(s') \leftarrow 1$
    Simulation: evaluate s' area, power and latency of flows and calculate $Q_{ws}(s')$
    $s_p \leftarrow s'$
    while $s_p \neq s_{root}$ **do**
      $s_p \leftarrow$ the parent of s_p
      $V(s_p) \leftarrow V(s_p) + 1$
    end while
  end while
  $s_{dc} \leftarrow \arg\max_{(s \in T_{s_{root}}) \textbf{and} s \neq s_{root}} Q_{ws}(s)$
  while the parent of $s_{dc} \neq s_{root}$ **do**
    $s_{dc} \leftarrow$ the parent of s_{dc}
  end while
  $s_{root} \leftarrow s_{dc}$
end for

Backpropagation

Root Update: find the direct child s_{dc} of root s_{root} that yields highest future reward

return $s^* \leftarrow \arg\max_{s \in T_{s_0}} Q_{ws}(s)$

After modelling the NoC DSE as an MDP, MCTS is applied. As shown in Algorithm 1, the MCTS-based NoC takes an initial NoC architecture as the input state s_0. In the first step, the initial NoC architecture is evaluated and used as the root of the search tree. The space is then iteratively explored following the four MCTS steps: selection, expansion, simulation, and back-propagation. As introduced in Section 3.1, MCTS explores successor nodes at deeper layers of the search tree, but its target is to find the direct child node that yields the highest future reward. So, after running out of computation budget, MCTS performs a root node update: It searches all nodes on the tree $T_{s_{root}}$ to find the node s_{dc} that has the highest quality Q_{ws}. Then the node s_{dc} iteratively switches to its parent node until reaching a direct child of the root node s_{root}. The edge between s_{dc} and s_{root} shows the best action that should be applied on the current root node s_{root}, because s_{dc} is the best direct child that yields the highest future reward. The root node is updated to be s_{dc} and the MCTS steps are re-started to find the next best action on the new root node. When the overall

exploration budget runs out, the algorithm searches among all so-far explored NoC architectures in the complete search tree T_{s_0} starting from s_0. The one with the maximal $Q_{ws}(s)$ is returned as the optimal NoC architecture s^*.

Following the search tree, we can trace back all applied actions from s^* to the initial design given by initial root s_0. This allows the designer to understand the exploration result and observe exactly which modifications the exploration suggested. Overall, this can also be used in an interactive exploration process as outlined in Reference [20].

We found no source that showed a formal study on the complexity of MCTS. In our practical application, the simulation step consumes a dominant part of computation time. The runtime of the algorithm depends linearly on the number of explored NoC architectures multiplied by the time to evaluate one NoC architecture in terms of area, power, and latency. This motivates us to develop faster NoC evaluation methods, which is discussed in the following section.

5 LATENCY ESTIMATION USING RNNS

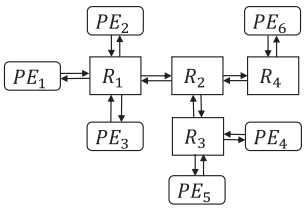
In the evaluation of candidate NoC architectures in DSE, the bottleneck is the SystemC simulation to obtain average packet latency. The main purpose of the RNN development is to speed up the evaluation of NoC candidate architectures to achieve a faster DSE. There exist other methods to speed up this process, e.g., Queuing Theory (QT). Yet, our implementation of QT (see Section 6 and Section 7.4) showed low accuracy in high-traffic scenarios during our DSE, hence, an RNN-based approach was developed as an alternative. To design an RNN for estimating the average latency of the packets of each traffic flow, several design choices need to be taken. First, the features of the input sequence and the output of the RNN must be defined. Subsequently, the RNN's architecture must be found such that it suits the application and also is able to learn and generalize from the features. As shown in Section 3.2.1, either LSTM or GRU can be chosen to build the RNN. In the following, we introduce the design of our RNN's input sequence and output as well as the training process, which includes the hyperparameter optimization to create the RNN's architecture, including the RNN type (LSTM or GRU), layer number, and so on.

5.1 RNN's Input Sequence and Output Design

The design of the input sequence and the definition of the output have a huge impact on both the training effort and the accuracy of the trained RNN. The output of the RNN is defined based on the target application. While existing NoC simulators, e.g., Booksim [31], directly report the average NoC latency, we design the RNN to estimate the latency Y_n of one specific traffic flow n. This design decision mostly yields advantages in terms of generalization. This made it possible to train the RNN once and apply it for the latency estimation for different NoC architectures.

To reduce the training effort and to avoid over-fitting, the RNN input sequence should focus on the input features that directly influence the latency of a flow n. In NoCs, the latency of a traffic flow is determined by two factors: the number of hops on its routing path and the waiting time in the router buffers at each hop. Waiting time arises due to contentions between multiple traffic flows using the same output port of a router. When flows from different input ports request to send data to the same output port, they must be arbitrated, hence, only one flow is serviced. Therefore, the waiting time is determined by the number of traffic flows competing for the output port and their bandwidth. To accurately estimate the latency of a flow, the input of the RNN should model both the number of hops on the routing path and all contentions. As a result, the input sequence $\mathbf{X}_n$ of the RNN is chosen as follows:

$$\mathbf{X}_n = [\mathbf{x}_{n,1} \quad \cdots \quad \mathbf{x}_{n,k} \quad \cdots \quad \mathbf{x}_{n,K_n-1}]. \tag{4}$$



Flow Description

Flow	Bandwidth	Routing Path
1	BW_1	$PE_1 \rightarrow R_1 \rightarrow R_2 \rightarrow R_4 \rightarrow PE_6$
2	BW_2	$PE_2 \rightarrow R_1 \rightarrow R_2 \rightarrow R_4 \rightarrow PE_6$
3	BW_3	$PE_3 \rightarrow R_1 \rightarrow R_2 \rightarrow R_4 \rightarrow PE_6$
4	BW_4	$PE_4 \rightarrow R_3 \rightarrow R_2 \rightarrow R_4 \rightarrow PE_6$
5	BW_5	$PE_5 \rightarrow R_3 \rightarrow R_2 \rightarrow R_4 \rightarrow PE_6$

Flow 1: Hop 1 ($\mathbf{x}_{1,1}$): $PE_1 \rightarrow R_1$, no sharing with others
Hop 2 ($\mathbf{x}_{1,2}$): $R_1 \rightarrow R_2$, share output with Flow 2 and 3 from different inputs of R_1
Hop 3 ($\mathbf{x}_{1,3}$): $R_2 \rightarrow R_4$, share output with Flow 4 and 5 from the same input of R_2
Hop 4 ($\mathbf{x}_{1,4}$): $R_4 \rightarrow PE_6$, no sharing with others

$\forall_{Flow\, n}\, (u_n = \frac{BW_n}{BW_{max}})$

RNN Input for Flow 1:

$$\mathbf{X}_1 = \begin{pmatrix} \mathbf{x}_{1,1} & \mathbf{x}_{1,2} & \mathbf{x}_{1,3} & \mathbf{x}_{1,4} \end{pmatrix} = \begin{bmatrix} u_1 & u_1 & u_1 & u_1 \\ 0 & u_2 & u_4 + u_5 & 0 \\ \vdots & u_3 & 0 & \vdots \\ \vdots & 0 & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Fig. 5. RNN input sequence example.

The length of the input sequence X_n is determined by the number of hops $K_n - 1$ on the routing path of the flow n. Each vector $\mathbf{x}_{n,k}$ of the sequence X_n models the possible contentions of the flow n at the kth hop. As RNNs only process input vectors of a fixed size, an upper bound for the number of elements was chosen with $|\mathbf{x}_{n,k}| = M$, where M is defined as the maximum number of input ports that each router is allowed to have. To model the contentions of a flow n at the kth hop, the elements in the vector store the bandwidths of the flows that compete with flow n. Competing flows share the same router output port as flow n but enter the router from a different input port. In this process, the flow bandwidths BW_n are normalized by the maximum allowed bandwidth BW_{max}, as shown in the following equation to obtain the average link utilization u_n of flow n on the link of hop k:

$$u_n = \frac{BW_n}{BW_{max,k}}. \tag{5}$$

Typically, $BW_{max,k}$ compromises to the maximum capacity of the link used in hop k in the NoC architecture. Thus, $BW_{max,k}$ can be calculated as the product of the link width and the corresponding clock frequency.

Figure 5 illustrates the creation of the RNN's input sequence with an example. The NoC architecture is described by a graph, where each node is either a PE or a router and each directed edge is a link connecting two NoC blocks. The bandwidth and the routing paths of the traffic flows are given in the table on the right-hand side of the figure. For simplicity, the link index k is omitted and we assume all links have the same bandwidth capacity BW_{max}. The RNN's input sequence $\mathbf{X}_1$ is then generated for flow 1. On the routing path of flow 1, four hops are needed to get from the source PE_1 to the destination PE_6 over the three routers R_1, R_2, and R_4. So the RNN's input sequence consists of four vectors: $\mathbf{X}_1 = [\mathbf{x}_{1,1} \quad \mathbf{x}_{1,2} \quad \mathbf{x}_{1,3} \quad \mathbf{x}_{1,4}]$. The first element in each vector $\mathbf{x}_{n,k}$ is always defined to store the normalized bandwidth of flow n, so $\forall_{k=1,2,3,4}(x_{1,k,1} = u_1)$. The following vector elements in $\mathbf{x}_{1,k}$ are used to describe the possible contentions of flow 1 at the kth hop. At the first hop from PE_1 to R_1, there can be no competing flows, because flow 1 uses the output port of its source PE exclusively, hence, $\forall_{m=2,\ldots,M}(x_{1,1,m} = 0)$. At the second hop from R_1 to R_2, flow 1 shares the output port of R_1 with flow 2 and flow 3. As the three flows use three different input ports of R_1, all flows are competing for the output port. Thereby, the contention probability depends on the bandwidth of those flows. During the contention, the input port requests are sent to an arbiter and granted to use the output port one after the other. By default,

Table 1. List of Symbols in Algorithm 2

Symbol	Meaning
N	Total number of traffic flows
M	Maximum number of input ports a router is allowed to have
$\mathbf{p}_n$	Routing path for flow n
$p_{n,k}$	The kth block (PE/router) on the routing path $\mathbf{p}_n$
K_n	Number of blocks on the routing path $\mathbf{p}_n$
l_d	NoC uni-directional link
$u_{n,k}$	Utilization of flow n on link in kth hop
(v_d, w_d)	v_d is the source block of l_d and w_d is destination block of l_d
$\mathbf{X}_n$	RNN's input sequence for flow n
$\mathbf{x}_{n,k}$	The kth vector in the sequence $\mathbf{X}_n$
$x_{n,k,m}$	The mth element in the vector $\mathbf{x}_{n,k}$
Y_n	RNN output (estimated latency) for flow n

the arbitration policy, e.g., round-robin, provides the same priority to each requesting input port. Thus, the three data flows have the same possibility to be forwarded, which is why the normalized bandwidths of the conflicting flows are stored within the first three elements of $\mathbf{x}_{1,2}$, as shown in Figure 5. At the third hop, flow 1 shares the output port of R_2 with flow 4 and flow 5. Here, the conflicting flows 4 and 5 arrive at the same input port of R_2. Since the arbitration policy provides fair access between input port requests and not between flows, the bandwidth of the flows 4 and 5 are summed up and stored as one element in the input vector. As shown in Figure 5, the element $x_{1,3,2}$ is equal to $u_4 + u_5$ and the following elements are zero, because there are no competing flows from other input ports. This is also the reason why the vector length of any input vector $\mathbf{x}_{n,k}$ is fixed by the maximum allowed input port number M. At the fourth hop, flow 1 shares no router output port with other flows, so the vector only stores the normalized bandwidth of flow 1 at the first element of the input vector and the remaining elements are set to zero. The algorithm to compute the RNN's input sequence is given in Algorithm 2. For convenience, all symbols used in the algorithm are listed in Table 1. The inputs of the algorithm are the routing paths of all flows in the NoC. For each flow n, its routing path is represented as a vector of ordered blocks $\mathbf{p}_n = [p_{n,1} \quad p_{n,2} \quad \cdots \quad p_{n,K_n}]$, where $p_{n,1}$ is always the source PE of flow n and p_{n,K_n} is always the destination PE of flow n. For the example illustrated in Figure 5, the routing path of flow 1 corresponds to $\mathbf{p}_1 = [PE_1 \quad R_1 \quad R_2 \quad R_4 \quad PE_6]$. Algorithm 2 returns the RNN's input sequences for all flows. They are computed one after the other as described in the following:

To compute the RNN input sequence $\mathbf{X}_n$ for a flow n, the first step is to find the number of blocks K_n on the routing path $\mathbf{p}_n$. In this process, we also obtain the hop number as $K_n - 1$. Then for each hop k, we create one vector $\mathbf{x}_{n,k}$ and initialize it as $[u_n \quad 0 \quad \cdots \quad 0]$. For any flow n, the first hop is always the packet injection from the source PE to its connected router. No router output port is used at the first hop, so there exists no contention and $\mathbf{x}_{n,1}$ remains as initialized. The contentions of the following hops are computed consecutively in a *for* loop, as illustrated in line 8 of Algorithm 2. According to the routing path of flow n, the kth hop goes from the block $p_{n,k}$ to the block $p_{n,k+1}$. Since the loop starts from the second hop, $2 \leq k \leq (K_n - 1)$, the block $p_{n,k}$ must be a router. Here, the task is to find the competing flows, which share the output port of p_{n_k} with flow n for each input port of p_{n_k}, respectively. The separation of the competing flows is necessary, because the arbitration policy of the used NoC routers grants requests fairly among input ports and not among flows. Therefore, as shown in line 9 of Algorithm 2, the function GetInputLinks($p_{n,k}$) returns the set of links that are connected to the input ports of the router $p_{n,k}$. The link used by

the flow n is then deleted from the set $\mathcal{U}$, and the remaining links are candidates that may be used by competing flows. Thus, for each link $l_d = (v_d, w_d) \in \mathcal{U}$, the algorithm checks all flows one-by-one. From the computation of $\mathcal{U}$, its element must satisfy $w_d = p_{n,k}$. For each flow i, if there exists a hop on its routing path that satisfies ($p_{i,j} = p_{n,k}$ AND $p_{i,j+1} = p_{n,k}$), then flow i is a competing flow for the flow n at the kth hop. Furthermore, the algorithm also checks whether link $l_d = (v_d, w_d)$ is used on the routing path by comparing $p_{i,j-1}$ with v_d. The normalized bandwidth of all competing flows on the link l_d will be summed up and stored in the contention vector $\mathbf{x}_{n,k}$. After computing the vector at every hop, they are combined to create the RNN's input sequence for the flow n as $\mathbf{X}_n = [\mathbf{x}_{n,1} \quad \cdots \quad \mathbf{x}_{n,K_n-1}]$. This process is repeated for all flows.

ALGORITHM 2: RNN Input Sequences Computation.

Input: Routing paths of all flows in the NoC: $\mathbf{p}_1, \ldots, \mathbf{p}_N$
Output: RNN input sequences for all flows: $\mathbf{X}_1, \ldots, \mathbf{X}_N$

for $n \leftarrow 1$ to N **do**
  $K_n \leftarrow |\mathbf{p}_n|$ // K_n gets the number of blocks on routing path
  for $k \leftarrow 1$ to $K_n - 1$ **do** // $K_n - 1$ is equal to the number of hops
    Initialize: $\mathbf{x}_{n,k} \leftarrow [\underbrace{u_n \quad 0 \quad \ldots \quad 0}_{|\mathbf{x}_{n,k}| = M}]$ // M is the max allowed input port number
  end for
  for $k \leftarrow 2$ to $K_n - 1$ **do** // loop to compute the contention at the kth hop
    $\mathcal{U} \leftarrow$ GetInputLinks($p_{n,k}$) // find links connected to input ports of router $p_{n,k}$
    $\mathcal{U} \leftarrow \mathcal{U} \setminus \{(p_{n,k-1}, p_{n,k})\}$ // delete the input link used by flow n
    $m \leftarrow 2$
    for all link $l_d = (v_d, w_d) \in \mathcal{U}$ **do**
      for $i \leftarrow 1$ to N **do**
        // check whether flow i uses link l_d and shares router output with flow n
        // competitive flows on link l_d are summed for using a same router input port
        if $\exists_j$ ($p_{i,j-1} = v_d$ AND $p_{i,j} = p_{n,k}$ AND $p_{i,j+1} = p_{n,k+1}$) **then**
          $x_{n,k,m} \leftarrow x_{n,k,m} + u_i$
        end if
      end for
      if $x_{n,k,m} \neq 0$ **then** // check if competitive flow exists on link l_d
        $m \leftarrow m + 1$
      end if
    end for
  end for
  $\mathbf{X}_n \leftarrow [\mathbf{x}_{n,1} \quad \ldots \quad \mathbf{x}_{n,K_n-1}]$
end for

return $\mathbf{X}_1, \ldots, \mathbf{X}_N$

function GetInputLinks($p_{n,k}$)
$\mathcal{U} \leftarrow \emptyset$
for all link $l_d = (v_d, w_d) \in \{l_1, l_2, \ldots, l_L\}$ **do** // loop over all links in the NoC
  if $w_d = p_{n,k}$ **then**
    $\mathcal{U} \leftarrow \mathcal{U} \cap \{l_d\}$
  end if
end for
return $\mathcal{U}$

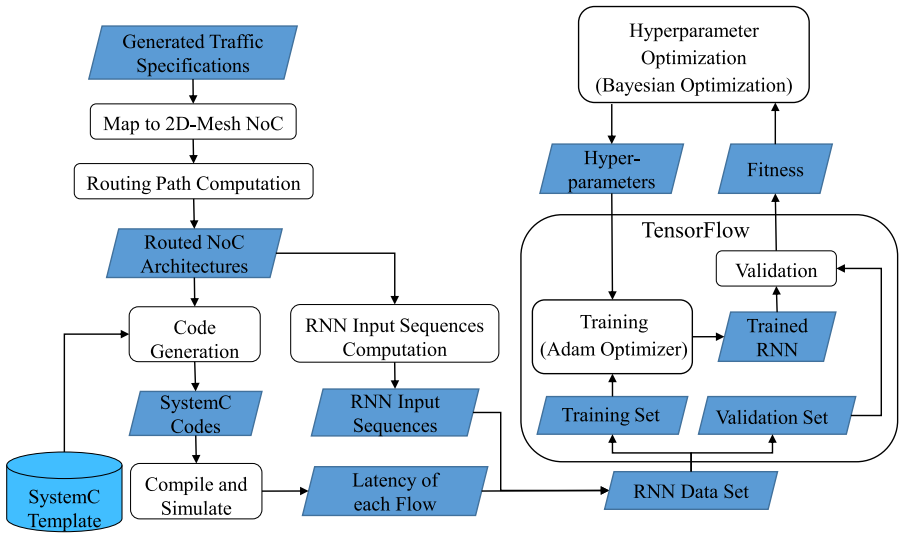


Fig. 6. Training process of the RNN.

5.2 Training Process

Before training the RNN, one necessary step is to prepare a dataset. We did not have a dataset with a sufficiently large number of traffic specifications for different MPSoCs to train the RNN. Yet, NoCs have the clear advantage that one can generate arbitrary MPSoC traffic specifications and use them to train the RNN. If these generated synthetic traffic specifications sufficiently cover the space of possible MPSoCs, the RNN will then perform well for realistic MPSoC workloads if it can generalize from the features. In this article, the dataset is created from randomly generated MPSoC traffic specifications. Each such synthetic traffic specification consists of 16 PEs. The number of flows between PEs and the bandwidth of those flows is obtained by sampling a random process following a normal distribution.

With the generated MPSoC traffic specifications, the RNN is trained following the process illustrated in Figure 6. In the first step, the generated synthetic MPSoC traffic specifications are mapped to a 2D-mesh NoC architecture. Even though the RNN is trained with 2D-mesh architectures, it still can be used to estimate the latency of an arbitrary NoC architecture, since the RNN estimates the latency for each flow individually. After mapping, routing paths of all flows are computed by solving an integer linear programming (ILP) model with the Gurobi solver [2]. It minimizes the contentions between traffic flows and also guarantees that the routing paths are deadlock-free. Afterwards, the routed NoC architectures are sent to two parallel branches. On the first branch, the Freemarker code generation engine [1] is used to generate the code for a cycle-accurate SystemC simulation for each NoC architecture, respectively. It uses a SystemC template for code generation as described in Reference [21]. The generated code can be automatically compiled and simulated to obtain the average packet latency of each traffic flow. In parallel to the code generation, the RNN's input sequences are computed according to Algorithm 2. By combining the RNN's input sequence with the simulated flow latency, we obtain the RNN dataset for training. It consists of pairs of an RNN's input sequence and its golden output, which is equal to the flow latency obtained from the cycle-accurate simulation.

In this article, the TensorFlow framework [16] is used to train the RNN using the Adam optimizer. A Bayesian optimization method provided by the Scikit-learn library [36] is used to optimize

the hyperparameters that control the network architecture and training process. The hyperparameters include the learning rate, the weight decay factor in the Adam optimizer, the RNN cell type (GRU or LSTM), the number of units per cell, and the layer number. As shown in Figure 6, the RNN dataset is split into the training and the validation set. While the training set is used to train the RNN, the validation set is used by the hyperparameter optimization to compute the fitness, measured by the negative mean squared error, of the trained RNN architecture. Based on the fitness value, the Bayesian optimization method then selects a new set of hyperparameters, which is used for the next iteration of the training process. This procedure repeats until an upper bound of iterations is reached. Once the hyperparameter optimization terminates, the RNN architecture with the highest fitness value is selected and exported. The test set is not shown in Figure 6, because the trained RNN is not tested on the NoC architectures obtained for the synthetic traffic specifications. To guarantee the generalization, the RNN is instead tested on benchmarks that are created from real MPSoC traffic specifications, as described in Section 7.

5.3 Trained RNN Structure

As mentioned before, the RNN that has the highest negative mean squared error is exported from the hyperparameter optimization. Meanwhile, we also obtain its hyperparameters and validation errors. In our case, the exported RNN is constructed with GRU cells, and each GRU cell consists of 57 units. Inside the RNN, there exists an input layer, an output layer, and four GRU layers in between. The RNN is trained with the learning rate of $1.6465e^{-5}$ and the weight decay factor of $3.122e^{-3}$. From the validation, the mean squared error is measured as 0.01567. Meanwhile, the mean absolute error is reported as 0.05654, which is 10.07% of the golden output values on average.

6 NOC DSE FRAMEWORK

Initially, the given traffic specification (PEs and their flows) are mapped to a 2D-mesh NoC architecture, which is our initial design. NoC architectures are stored in XML files following the communication exchange format (CEF) [4]. The design space is then explored with the framework shown in Figure 7, which is based on the Eclipse Modelling Framework (EMF). To evaluate performances and costs of the NoC architecture, its routing paths must be computed. As mentioned before, routing paths can be obtained by solving an ILP model using a custom routing tool. This tool uses the Gurobi ILP solver [2]. After routing, Orion 3.0 [24], which is calibrated with the LISNoC [43], is used to estimate the power and area of NoC architectures. The routed NoC architecture given as s can be sent to the Freemarker code generation engine. Following the corresponding template, the router configuration file of Orion is automatically generated for each input NoC architecture. It is then compiled together with the Orion library and executed to obtain the estimated power $g_2(s)$ and area $g_3(s)$ for this NoC architecture. Besides power and area, the latency is another important metric. Three alternative approaches are provided to estimate NoC latency. The first approach uses code generation to generate a cycle-accurate SystemC (SC) simulation model. The model is automatically compiled and simulated. From the simulation report, the average packet latency for each flow is obtained. This is the most accurate approach, but also the most computationally expensive one. With the latency of each single flow and its share of the overall communication bandwidth, we compute the overall average packet latency $g_1(s)$ as the weighted average. The second approach is to use the Queuing Theory (QT). We used the approach in Reference [33] to guide our QT implementation using routing paths and flow bandwidth information to create a QT to estimate the average latency of each single flow in the NoC. Then, we compute the NoC average latency $g_1(s)$ in the same way as in SC. It must be noted that there might exist more complex and accurate QT models that may perform better and could be explored as further alternative. Yet, an advantage of our third approach to use our trained RNN was that it was very

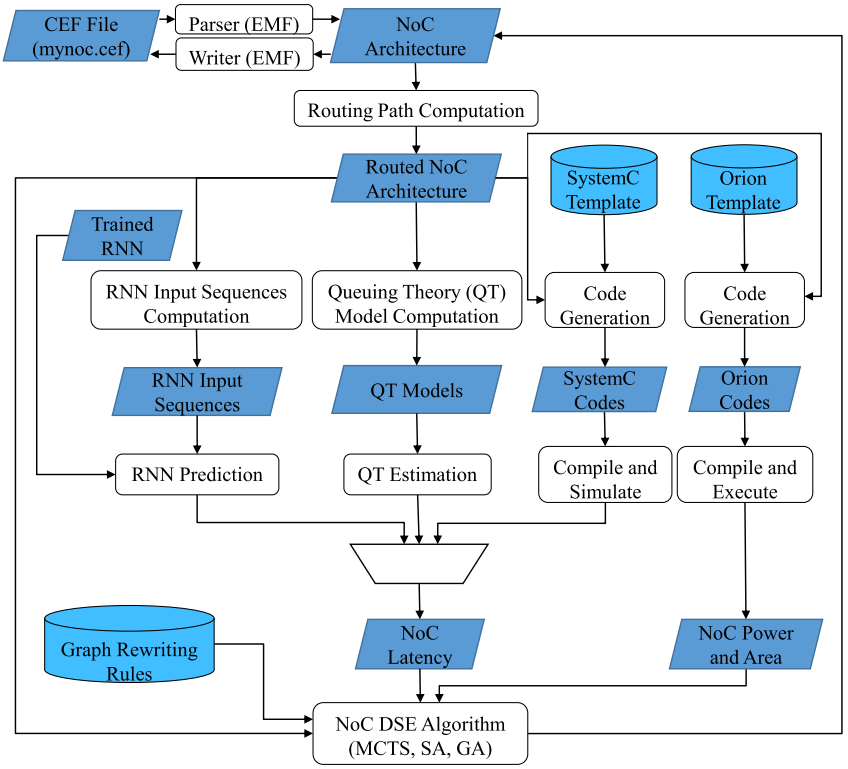


Fig. 7. NoC DSE framework.

straightforward to implement once the RNN structure and feature vector were defined. The RNN input sequences for all flows is computed from the routed NoC architecture using Algorithm 2. The TensorFlow framework loads the trained RNN and then predicts the single flow latency as output Y_n according to the input sequence. Again, weighted averaging is used to obtain overall latency $g_1(s)$ from the individual Y_n. The three metrics $g_1(s)$, $g_2(s)$, and $g_3(s)$ are used by the NoC DSE search heuristics to compute the reward function $Q_{ws}(s)$ according to Equation (3).

Three different NoC DSE search heuristics are implemented: Monte-Carlo-Tree Search (MCTS), Simulated Annealing (SA), and a Genetic Algorithm (GA). All heuristics can run until a termination condition is fulfilled, e.g., the computation budget runs out. Then the best explored NoC architecture will be returned.

7 EXPERIMENTAL RESULTS

7.1 Experimental Setup

Our experiments are performed with several benchmark applications. They consist of the multimedia application from Reference [19], the SoC benchmark used in the European project NaNoC [4], the industrial mobile phone application [3], and one synthetic MPSoC application with uniform distributed communication load. The multimedia application and MPSoC application each consist of 16 PEs, the mobile application consists of 22 PEs, and the NaNoC application consists of 25 PEs. Our target is to optimize area, power, and latency for all benchmark applications by maximizing the reward function $Q_{ws}(s)$ defined by Equation (3). This is equal to minimizing the total cost function $-Q_{ws(s)}$. We chose $\lambda_1 = \lambda_2 = \lambda_3 = 0.33$ for the weight parameters of the reward

Table 2. Comparison of Results (Averaged over 10 Runs) between Optimization Heuristics

Testbench	Method	Area (μm^2)	Power (mW)	Latency (Cycles)	Constraint Penalty	Total Cost $-Q_{ws}(s^*)$	Improv.
MPSoC	Initial	324,207	66.31	23.9	1.6325	2.6225	-
	SA	69,949	14.44	19.67	0	0.4147	84.19%
	GA	157,288	32.58	20.54	0	0.6058	76.90%
	MCTS	53,047	10.95	19.46	0	0.3771	85.62%
Mobile	Initial	563,529	115.28	25.91	0.4972	1.4872	-
	SA	203,445	42.35	20.52	0	0.5017	66.26%
	GA	349,993	72.36	24.04	0	0.7182	51.70%
	MCTS	73,184	15.13	22.64	0	0.3745	74.82%
NaNoC	Initial	571,403	116.97	28.17	0	0.99	-
	SA	213,152	44.36	22.29	0	0.5094	48.55%
	GA	414,551	86.01	24.37	0	0.7676	22.47%
	MCTS	80,982	16.73	23.30	0	0.3669	62.94%
Multimedia	Initial	324,207	66.7	20.32	0.7372	1.7272	-
	SA	66,590	13.85	15.65	0	0.3905	77.39%
	GA	139,522	29.02	18.05	0	0.5787	66.49%
	MCTS	52,645	10.95	15.97	0	0.3671	78.75%

function $Q_{ws}(s)$. The same priority is given to area, power, and latency. Furthermore, we penalize constraint violations with a weight parameter of $\lambda_4 = 0.1$. This means that a 10% reduction in all three main objectives is equal in terms of the reward gain to reducing the worst-case average upper bound latency violation by 1 cycle. Hence, the optimization prioritizes the avoidance of the penalty heavily, resulting in solutions with no or very small constraint violations.

7.2 Comparison of MCTS vs. GA and SA Heuristics

In this study, SystemC simulation was used to obtain average latency. The power and area are estimated with Orion 3.0. Our experiments compare the MCTS optimization heuristic with other popular methods: SA and a GA. For fair comparison, all three methods start from the same initial designs and are allocated the same computational budget.

Each optimization method is repeated 10 times with different random number seeds on all benchmark applications to average statistical effects. The averaged optimization result for each benchmark application is given in Table 2. Compared with initial designs, all three methods obtain significant improvements. Clearly, all benchmark systems do benefit from a custom-fit NoC architecture. According to SystemC simulation, constraint violation exists in initial architectures of MPSoC, Mobile, and Multimedia benchmarks, so they are penalized. E.g., for the one of MPSoC, the average latency of the constrained flow is 16.325 cycles slower, so it is weighted by λ_4 and brings a penalty of 1.6325. As can be seen, after optimization, violations are all removed and MCTS can always obtain more improvements than SA and GA. On average of four benchmarks, SA reduces the total cost by 69.10%, GA by 54.39%, and MCTS by 75.53%. Table 3 further measures the standard deviation of the optimized total cost $-Q_{ws}(s^*)$ for all 10 runs and shows the results of the best run and the worst run. The difference of improvements between the best and worst run is always less than 15%, and the standard deviation has small values, which indicates stable outputs for all optimization heuristics. MCTS overall shows good behavior and its worst run even outperforms the best run of SA and GA for the MPSoC and Mobile benchmark. So, as can be seen, MCTS is a very competitive heuristic. Of course, each heuristic can be further optimized to

Table 3. Comparison of Extreme Cases and Deviation between Optimization Heuristics

Testbench	Method	Opt. Result (Best run)		Opt. Result (Worst run)		Standard Deviation
		$-Q_{ws}(s^*)$	Improv	$-Q_{ws}(s^*)$	Improv	$\sigma(-Q_{ws}(s^*))$
MPSoC	SA	0.3906	85.11%	0.4604	82.45%	0.0215
	GA	0.5150	80.36%	0.7130	72.81%	0.06105
	MCTS	0.3563	86.41%	0.4178	84.07%	0.0216
Mobile	SA	0.4578	69.22%	0.5512	62.93%	0.0275
	GA	0.6822	54.13%	0.7508	49.52%	0.02459
	MCTS	0.3338	77.56%	0.4075	72.60%	0.0218
NaNoC	SA	0.4441	55.14%	0.5807	41.34%	0.0407
	GA	0.6721	32.11%	0.8247	16.70%	0.05333
	MCTS	0.3412	65.54%	0.4166	57.92%	0.0205
Multimedia	SA	0.3656	78.83%	0.4154	75.95%	0.01476
	GA	0.5004	71.03%	0.6653	61.48%	0.0513
	MCTS	0.3568	79.34%	0.3767	78.19%	0.0074

Table 4. Speed Comparison: SC vs. QT vs. RNN

Benchmark	Method	Computation Time (ms)				Speed-up
		Latency	Routing	Orion	Total	
MPSoC	SC	3,058	439	65	3,562	1.0×
	QT	21			525	6.78×
	RNN	105			609	5.85×
Mobile	SC	6,713	803	102	7,618	1.0×
	QT	23			928	8.21×
	RNN	126			1,031	7.39×
NaNoC	SC	6,762	810	108	7,680	1.0×
	QT	26			944	8.13×
	RNN	122			1,040	7.38×
Multimedia	SC	3,302	362	69	3,733	1.0×
	QT	26			457	8.17×
	RNN	120			551	6.77×

perform better on the NoC DSE problem. We believe the comparison is fair, as a problem-optimized implementation of each heuristic is used. In our eyes, MCTS performs very well and better than the other heuristics because it is designed to balance exploration vs. exploitation.

7.3 Speed Comparison of RNN vs. Simulation and QT Latency Estimation

As mentioned before, the main purpose of the RNN development is to speed up the evaluation of NoC candidate architectures. We compare the speed of the RNN estimation with the cycle-accurate SystemC simulation (SC) and the estimation using Queuing Theory (QT). For each of the four benchmarks, the computation time is measured and the speed-up is investigated. As shown in Table 4, both QT and RNN are much faster than SC. For the Mobile benchmark, the SC consumes $6,713ms$ on the latency computation. The QT takes only $23ms$ and this is 0.34% of the SC computation time. The RNN consumes $126ms$. It is not as fast as QT, but still is only 1.88% of the SC computation time. Even for the MPSoC benchmark, where the SC is faster than in other

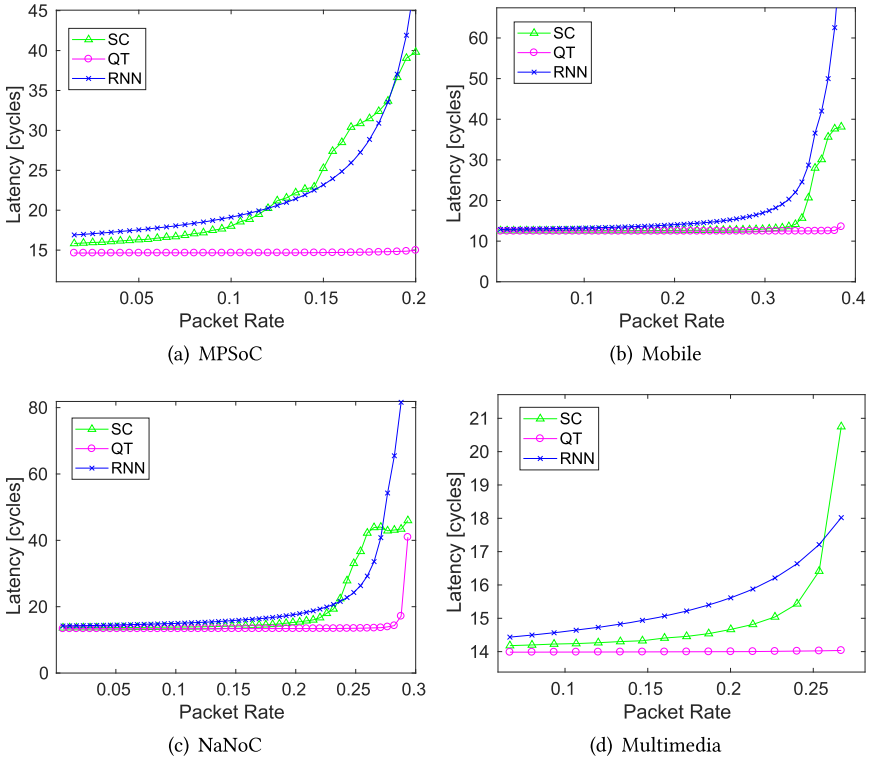


Fig. 8. Comparison of latency over packet rate for SC vs. QT vs. RNN for one NoC architecture.

benchmarks, QT takes only 21*ms* to estimate latency, which is 0.69% of the SC computation time. RNN takes 105*ms* and this is only 3.43% of the SC computation time.

To compare the speed-up obtained for the full DSE, Table 4 shows the computation time for routing and for executing Orion for power and area estimation. The total time to evaluate a single NoC architecture during the DSE is the sum of computation time for latency estimation, routing, and Orion execution. Since both QT and RNN heavily reduce the computation time for latency, the overall speed-up is limited now by the routing time. As a result, QT and RNN obtain similar speed-up: compared to SC, QT provides 6.78× to 8.21× speed-up, while RNN provides 5.85× to 7.38× speed-up.

7.4 Accuracy Comparison of RNN vs. Simulation and QT Latency Estimation

The cycle-accurate SystemC simulation (SC) consumes a lot of computation time, but provides accurate latency estimation. The QT and RNN have proven their advantages in speed, but it is also very important to guarantee good accuracy. Figure 8 compares the computed latency from SC, QT, and RNN under different packet rates. When the packet rate is low, the computed latency from QT and RNN are similar to SC, which indicates high accuracy. Besides, in the region with a low packet rate, the latency curves are almost flat. The latency is not sensitive to the increasing packet rate, because the flow contention rate is also low. Therefore, the waiting time at each hop is only a small contribution into the overall latency. The latency mainly depends on the number of hops on the routing path, which can be accurately determined by all three methods. However, when the packet rate increases above a certain threshold, the contention will have a strong influence.

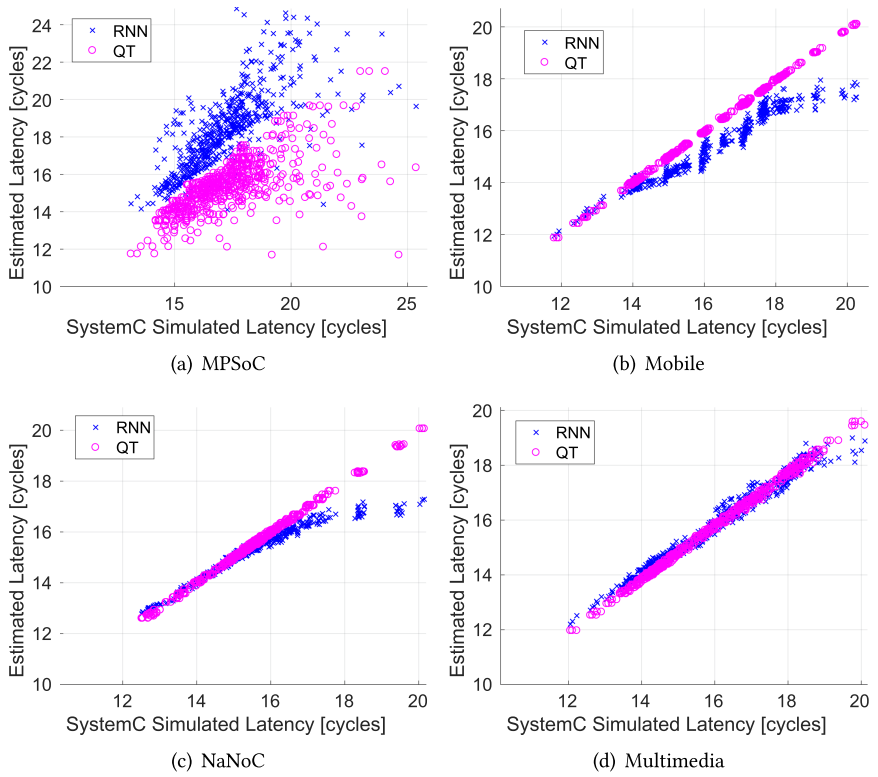


Fig. 9. Comparison of latency for SC vs. QT vs. RNN for several NoC architectures (low packet rate).

The latency will increase rapidly with the packet rate, e.g, in Figure 8(b), the latency increases exponentially with the packet rate when the packet rate is above the threshold 0.32 flit/cycle. Our QT model does not capture this dependency accurately, e.g, in Figure 8(c), the QT latency increases rapidly when packet rate is above 0.29 flit/cycle, while the actual threshold is around 0.21 flit/cycle. As shown in the other Figures 8(a), 8(b), and 8(d), QT has the same problem for the other benchmarks. In comparison to QT, the trained RNN fits the SC curve better. Both SC and RNN curves have similar thresholds, such that the error between SC curve and RNN curve is smaller for high-traffic scenarios.

For each benchmark, we also generate randomly different NoC architectures, including not only 2D-mesh architectures but also heterogeneous application-specific NoC architectures. For each architecture, we use all three approaches to compute the latency for a low-traffic scenario and a high-traffic scenario by scaling the packet injection rate (low and high), which is in general equal to adapting the router frequencies. Figure 9 shows the comparison between QT and RNN at low packet rate. Each point represents one specific NoC architecture. The X-axis is the actual latency obtained from SC and the Y-axis is the estimated latency from QT or RNN. Ideally, the estimated latency should be equal to the simulated latency and points should be located on the diagonal line. As shown in Figure 9(b) for Mobile, 9(c) for NaNoC, and 9(d) for Multimedia, QT and RNN points are located close to the diagonal, which means the estimated latency is matching with SC if the overall packet latency is low. For NoC architectures with high average latency, e.g., in NaNoC above 18 cycles, the RNN points are located slightly below the diagonal, such that latency is under-estimated. But its absolute errors are mostly below three cycles. For the MPSoC benchmark shown

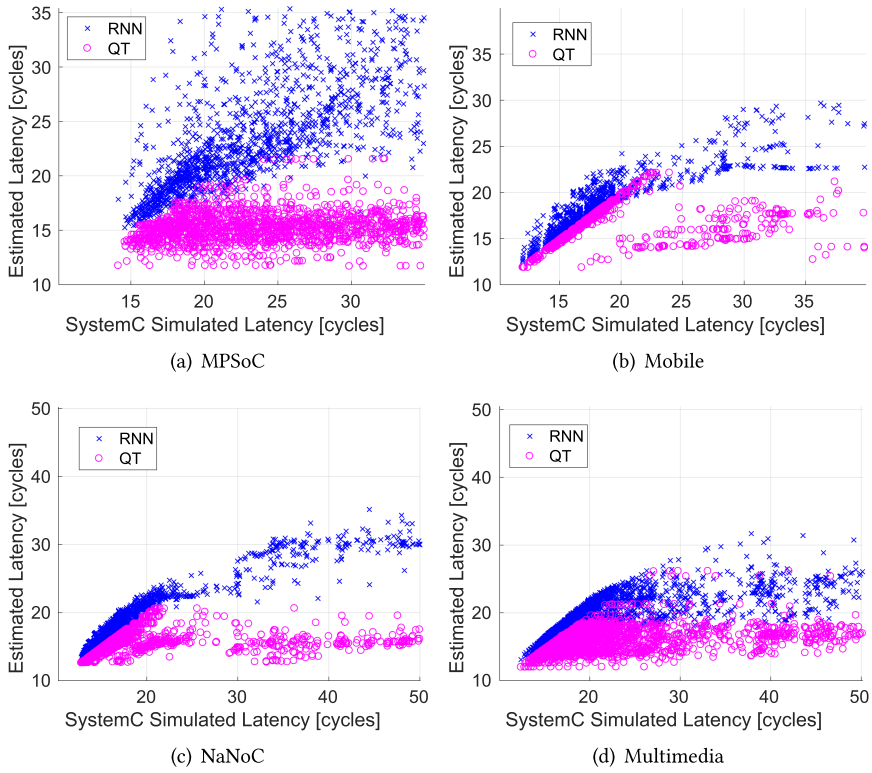


Fig. 10. Comparison of latency for SC vs. QT vs. RNN for several NoC architectures (high packet rate).

in Figure 9(a), both RNN and QT show higher errors. The RNN and QT are also compared using a high packet rate in Figure 10. Here, as can be seen, errors are much higher. Overall, estimation accuracy suffers for those scenarios with flow contention. These can be either due to the NoC architecture forcing flows to conflict or due to high packet rates. Yet, during a DSE, we need to predict these contentions to identify good NoC architectures. For this purpose high fidelity rather than high accuracy is required. Therefore, we look into the fidelity of these methods in the next section.

7.5 Fidelity Comparison of RNN vs. Simulation and QT Latency Estimation

Fidelity is an important metric when the estimation is used to guide an DSE process. Assume we have Z pairs of variables, $\{(\alpha_1, \beta_1), \ldots, (\alpha_z, \beta_z), \ldots, (\alpha_Z, \beta_Z)\}$, where α_z is the actual value and β_z is the estimated value. The fidelity is defined to measure correlations between the ordering of the estimated points and the ordering of the actual points. Hence, it shows whether a search using the estimated values would guide the DSE in the direction of better actual values. That is to say, if $\alpha_1 < \alpha_2$, then β_1 should be smaller than β_2 as well. There exist various measurements for fidelity. This article uses the popular Kendall's tau coefficient [17] and Spearman's rank correlation coefficient [38] as suggested in Reference [13].

Kendall's tau coefficient can be calculated as

$$\tau = \frac{c_1 - c_2}{\frac{Z \times (Z-1)}{2}}, \tag{6}$$

where c_1 is the number of pairs that are correctly ordered and c_2 is the number of pairs that are inversely ordered. The number of pairs is Z. The value of τ lies between -1 and 1. A value of $\tau = 1$

Table 5. Fidelity Comparison (Low Packet Rate): RNN vs. QT

Benchmark	Figure Index	τ		ρ	
		RNN	QT	RNN	QT
MPSoC	9(a)	0.6336	0.5642	0.809	0.7394
Mobile	9(b)	0.8832	0.9687	0.9836	0.9981
NaNoC	9(c)	0.8998	0.9531	0.9836	0.9961
Multimedia	9(d)	0.9228	0.9659	0.9919	0.9982
Average		0.8349	0.8629	0.9420	0.9329

indicates that all pairs are correctly ordered and, vice versa, $\tau = -1$ indicates all pairs are inversely (wrongly) ordered.

Spearman's rank correlation coefficient can be calculated as

$$\rho = 1 - \frac{2 \times \sum_{z=1}^{Z} (\alpha_z^r - \beta_z^r)}{\frac{Z \times (Z^2 - 1)}{3}}, \tag{7}$$

where α_z^r and β_z^r are the rank of α_z and β_z, respectively, and $(\alpha_z^r - \beta_z^r)$ calculates the rank distance. The rank can be understood as the location in the ordered list. E.g., if α_z has the smallest value among all α, then α_z^r is equal to one, the second smallest element has the rank of two, and the largest element has the rank of Z. Similarly to Kendall's tau coefficient, the value of ρ lies between -1 and 1. A value of $\rho = 1$ indicates all pairs are correctly ordered, and $\rho = -1$ indicates that the rank distances are maximized, i.e., all pairs are inversely ordered.

Within the NoC DSE process, the latency obtained from the cycle-accurate SystemC simulation is taken as golden reference (the actual value) and the latency obtained from the QT and RNN are the two estimated values. The fidelity is checked on the four benchmarks using again Figure 10 and Figure 9. As was seen, both RNN and QT suffer from larger absolute errors in the case of high contention in the NoC. However, the RNN still maintains good fidelity. Especially for Mobile in Figure 10(b) and NaNoC in Figure 10(c), the RNN points are distributed along a straight line: When SC latency increases, the RNN estimated latency also increases. In contrast, QT is not able to maintain good fidelity. When the latency is low, QT estimated latency fits SC latency very well. E.g., in Figure 10(b), when simulated latency is below 24 cycles, the QT points are located nicely on the diagonal line. However, when the simulated latency increases above 24 cycles, the estimated latency from QT is even becoming smaller, which violates the fidelity requirement. The same problem of QT exists in other benchmarks as well as seen from the figures. When the simulated latency is increasing, we can hardly see the increasing trend with the QT estimation. This is confirmed in the computed Kendall's tau coefficient τ and Spearman's rank correlation coefficient ρ in Table 6.

Table 5 computes Kendall's tau coefficient τ and Spearman's rank correlation coefficient ρ using the points in Figure 9. For the MPSoC benchmark, the RNN shows better fidelity, since both correlation coefficients τ and ρ have larger values compared to QT. In the other benchmarks, both QT and RNN have very good fidelity. The value of τ_{RNN} is between 0.8832 to 0.9238, while the value of τ_{QT} is above 0.95. In terms of Spearman's rank correlation coefficient ρ, both RNN and QT are very close to one. On average among four benchmarks, τ_{RNN} is 0.8349, while τ_{QT} is slightly larger and is 0.8629. When comparing the average Spearman's rank correlation coefficient, ρ_{RNN} is 0.942 and ρ_{QT} is 0.9329. Both are very close to one, which indicates that both QT and RNN estimation have very good fidelity. This also fits our observation in Figure 9.

According to Table 6, for the MPSoC benchmark, τ_{QT} is reduced to 0.1996, which indicates a weak correlation between estimated latency and actual latency. Meanwhile, ρ_{QT} is also heavily

Table 6. Fidelity Comparison (High Packet Rate): RNN vs. QT

Benchmark	Figure Index	τ		ρ	
		RNN	QT	RNN	QT
MPSoC	10(a)	0.5935	0.1996	0.7789	0.2896
Mobile	10(b)	0.8237	0.5829	0.952	0.6293
NaNoC	10(c)	0.8438	0.527	0.9613	0.6386
Multimedia	10(d)	0.7306	0.4429	0.8811	0.6099
Average		0.7479	0.4381	0.8933	0.5418

Table 7. DSE Results: MCTS with SystemC Simulation (SC) vs. with Queuing Theory (QT) vs. with RNN

Benchmark	Latency Esti.	Area (μm^2)	Power (mW)	Latency (Cycles)	Constraint Penalty	Total Cost ($-Q_{ws}(s^*)$)	Cost Incr.
MPSoC	SC	53,047	10.95	19.46	0	0.3771	-
	QT	38,361	7.96	21.50	0.2357	0.6115	60%
	RNN	47,785	9.91	20.53	0.0964	0.4780	25%
Mobile	SC	73,184	15.13	22.64	0	0.3745	-
	QT	70,555	14.61	25.22	0.7989	1.2029	221%
	RNN	73,224	15.12	23.92	0.2201	0.6114	63%
NaNoC	SC	80,982	16.73	23.30	0	0.3669	-
	QT	79,924	16.53	29.79	0.1094	0.5512	50%
	RNN	82,281	16.99	25.58	0.0227	0.4178	14%
Multimedia	SC	52,645	10.95	15.97	0	0.3671	-
	QT	57,384	11.99	16.10	0.0186	0.3985	8.38%
	RNN	60,007	12.49	15.84	0	0.3808	3.57%

reduced to 0.2896. In comparison, τ_{RNN} still is 0.5935 and ρ_{RNN} is 0.7789. These values are almost three times the value of τ_{QT} and ρ_{QT}. On average, τ_{RNN} is 0.7479 while τ_{QT} is only 0.4381. The average ρ_{RNN} reaches 0.8933, which is a very high value and indicates that most estimated latencies are ordered the same as the simulated latencies. In contrast, the average ρ_{QT} is only 0.5418. This also confirms our observation in Figure 10 that the RNN still maintains good fidelity and its estimated latency will increase with an increasing simulated latency. However, this trend is not obvious in QT estimated latency.

7.6 DSE with RNN vs. Simulation and QT Latency Estimation

We test SC, QT, and RNN for guiding the DSE process for our four benchmarks. For fair comparison, they use the same DSE algorithm (MCTS) with the same budget of NoC evaluations (fixed number of investigated NoC architectures). The limit is set to 3K evaluations. QT- and RNN-based DSE are repeated 10 times with different random number seeds to average statistical effects. The cost factors are set to $\lambda_1 = \lambda_2 = \lambda_3 = 0.33$, while the penalty term for $\lambda_4 = 0.1$. This means that a worst-case latency constraint violation by 1 cycle adds 0.1 to the cost, which equals a 10% reduction in all objectives. This pushes the DSE to solutions where critical flow latency constraints are kept.

QT and RNN provide around 5× to 8× speed-up, as reported in Section 7.3, and finish significantly faster. Yet, their provided estimations are not accurate, hence, we expect some loss in quality in comparison to the slower SC-based DSE. The corresponding DSE results are shown in Table 7. The cost increase of the QT-based DSE compared to SC (our baseline) is between 3.57%

Table 8. Estimation Accuracy of DSE Solutions: SC vs. QT vs. RNN

Benchmark	Method	Latency (*Cycles*)		Constraint Penalty		Total Cost ($-Q_{ws}(s^*)$)		
		Est	Actual	Est	Actual	Est	Actual	Error
MPSoC	SC	-	19.46	-	0	-	0.3771	-
	QT	14.69	21.50	0	0.2357	0.2817	0.6115	−53%
	RNN	17.80	20.53	0	0.0964	0.3440	0.4780	−28%
Mobile	SC	-	22.64	-	0	-	0.3745	-
	QT	14.20	25.22	0	0.7989	0.2638	1.2029	−78%
	RNN	22.12	23.92	0	0.2201	0.36756	0.6114	−40%
NaNoC	SC	-	23.30	-	0	-	0.3669	-
	QT	15.23	29.79	0	0.1094	0.2714	0.5512	−50%
	RNN	20.18	25.58	0	0.0227	0.3318	0.4178	−20%
Multimedia	SC	-	15.97	-	0	-	0.3671	-
	QT	14.12	16.10	0	0.0186	0.3477	0.3985	−12%
	RNN	15.79	15.84	0	0	0.3799	0.3808	−0.22%

for the Multimedia benchmark and up to 221% for the Mobile benchmark. In comparison, the cost increase for the RNN speed-up method is far less, between 8.38% and 60%. Hence, the machine learning–based RNN-based method is superior to the used QT method.

To analyze the results further, Table 8 shows a comparison between the estimated costs of QT and RNN and the actual cost of the final solution. Both methods underestimate the costs, as they mostly predict latency too low, which also corresponds to the findings on the accuracy in Section 7.4. This has two effects: first, the latency component in the cost function is underestimated, but secondly and more importantly, both methods predict that they found a solution without any latency constraint violation for the critical flows (estimated constraint penalty is zero). Yet, actual simulation shows that there are partly severe constraint violations on the critical flows (a value of 0.79 corresponds to an average latency constraint violation of about 8 cycles on some critical flow). Overall, QT underestimates total cost by up to 78%, RNN only by 40%. This mis-prediction guides the QT and RNN to sub-optimal solutions. Yet, as the RNN prediction is more accurate and fidelity is higher, it identifies better solutions than QT while providing approximately the same speed-up.

8 CONCLUSION

This article uses graph rewriting to modify NoC architectures and models the NoC DSE as an MDP. Then, MCTS from reinforcement learning is applied as the search heuristic. Compared with other heuristics, SA and GA, MCTS shows a better efficiency for the four evaluated benchmarks. Besides, an RNN is developed to replace the SystemC simulation (SC) for estimating the flow latency, which is the bottleneck of the DSE computational time. Compared to the popular analytical model for latency estimation based on Queuing Theory (QT), the RNN obtains almost the same speed-up in the DSE. However, the outputs from QT-based DSE are not as good in terms of quality as SC-based DSE, while the RNN-based DSE has similar speed-up with much lower sacrifice in quality of result.

REFERENCES

[1] FreeMarker Code Generation Engine. 2019. Retrieved from http://freemarker.org/index.html.
[2] Gurobi Optimizer. 2019. Retrieved from http://www.gurobi.com/.
[3] Mobile Benchmark. 2019. Retrieved from https://www.eda.ei.tum.de/forschung/electronic-system-level/.
[4] NaNoC Project. 2019. Retrieved from https://sites.google.com/site/nanocproject/communication-exchange-format-cef.

[5] M. Bedford Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. 2003. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*. 341–353. DOI: https://doi.org/10.1109/HPCA.2003.1183551

[6] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intell. AI Games* 4, 1 (Mar. 2012), 1–43. DOI: https://doi.org/10.1109/TCIAIG.2012.2186810

[7] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti. 2015. Noxim: An open, extensible and cycle-accurate network on chip simulator. In *Proceedings of the IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP'15)*. 162–163. DOI: https://doi.org/10.1109/ASAP.2015.7245728

[8] J. W. Chen, L. Tang, H. S. Xi, and J. Zhu. 2011. A stochastic network calculus based approach for on-chip networks. In *Proceedings of the 30th Chinese Control Conference*. 4545–4549.

[9] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR* abs/1406.1078 (2014).

[10] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR* abs/1412.3555 (2014).

[11] J. W. d. Mesquita, M. O. d. Cruz, M. M. Pereira, and M. E. Kreutz. 2016. Design space exploration using UTNoCs and genetic algorithm. In *Proceedings of the Brazilian Symposium on Computing Systems Engineering (SBESC'16)*. 198–202. DOI: https://doi.org/10.1109/SBESC.2016.038

[12] S. Das, J. R. Doppa, D. H. Kim, P. P. Pande, and K. Chakrabarty. 2015. Optimizing 3D NoC design for energy efficiency: A machine learning approach. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'15)*. 705–712. DOI: https://doi.org/10.1109/ICCAD.2015.7372639

[13] R. J. Douma, S. Altmeyer, and A. D. Pimentel. 2015. Fast and precise cache performance estimation for out-of-order execution. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'15)*. 1132–1137. DOI: https://doi.org/10.7873/DATE.2015.0066

[14] D. Silver et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR* abs/1712.01815 (2017).

[15] D. Silver et al. 2017. Mastering the game of Go without human knowledge. *Nature* 550 (2017), 354–359.

[16] M. Abadi et al. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *CoRR* abs/1603.04467 (2016).

[17] Z. Govindarajulu. 1992. Rank correlation methods (5th ed.). *Technometrics* 34 (1 1992), 108–108. DOI: https://doi.org/10.1080/00401706.1992.10485252

[18] S. Hochreiter and J. Schmidhuber. 1997. Long short-term memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. DOI: https://doi.org/10.1162/neco.1997.9.8.1735

[19] J. Hu and R. Marculescu. 2005. Energy- and performance-aware mapping for regular NoC architectures. *IEEE Trans. Comput.-Aided Des. Integ. Circ. Syst.* 24, 4 (Apr. 2005), 551–562. DOI: https://doi.org/10.1109/TCAD.2005.844106

[20] Y. Hu, D. Mueller-Gritschneder, and U. Schlichtmann. 2018. Wavefront-MCTS: Multi-objective design space exploration of NoC architectures based on Monte Carlo tree search. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18)*. ACM, New York, NY. DOI: https://doi.org/10.1145/3240765.3240863

[21] Y. Hu, D. Müller-Gritschneder, and U. Schlichtmann. 2017. Model-based framework for networks-on-chip design space exploration. In *Proceedings of the 2nd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems (AISTECS'17)*. ACM, New York, NY, 32–35. DOI: https://doi.org/10.1145/3073763.3073769

[22] Y. Hu, Y. Zhu, H. Chen, R. Graham, and C.-K. Cheng. 2006. Communication latency aware low power NoC synthesis. In *Proceedings of the 43rd ACM/IEEE Design Automation Conference*. 574–579. DOI: https://doi.org/10.1109/DAC.2006.229293

[23] M. Jassi, Y. Hu, D. Mueller-Gritschneder, and U. Schlichtmann. 2018. Graph-grammar-based IP-integration (GRIP)—An EDA tool for software-defined SoCs. *ACM Trans. Des. Autom. Electron. Syst.* 23, 3 (Apr. 2018), 26 pages. DOI: https://doi.org/10.1145/3139381

[24] A. B. Kahng, B. Lin, and S. Nath. 2012. Explicit modeling of control and data for improved NoC router estimation. In *Proceedings of the 49th Design Automation Conference (DAC'12). ACM/EDAC/IEEE*, 392–397. DOI: https://doi.org/10.1145/2228360.2228430

[25] A. B. Kahng, B. Lin, and S. Nath. 2015. ORION3.0: A comprehensive NoC router estimation tool. *IEEE Embed. Syst. Lett.* 7, 2 (June 2015), 41–45. DOI: https://doi.org/10.1109/LES.2015.2402197

[26] G. N. Khan and A. Tino. 2012. Synthesis of NoC interconnects for custom MPSoC architectures. In *Proceedings of the IEEE/ACM 6th International Symposium on Networks-on-Chip*. 75–82. DOI: https://doi.org/10.1109/NOCS.2012.16

[27] A. E. Kiasari, Z. Lu, and A. Jantsch. 2013. An analytical latency model for networks-on-chip. *IEEE Trans. Very Large Scale Integ. (VLSI) Syst.* 21, 1 (Jan. 2013), 113–123. DOI: https://doi.org/10.1109/TVLSI.2011.2178620

[28] A. Kumar and B. Talawar. 2018. Machine learning based framework to predict performance evaluation of on-chip networks. In *Proceedings of the 11th International Conference on Contemporary Computing (IC3'18)*. 1–6. DOI: https://doi.org/10.1109/IC3.2018.8530505

[29] B. Li, J. Zhao, J. Wang, and W. Dou. 2012. A max-plus algebra approach for network-on-chip end-to-end delay estimation. In *Proceedings of the 8th International Conference on Semantics, Knowledge and Grids*. 217–220. DOI: https://doi.org/10.1109/SKG.2012.6

[30] S. J. Lu, R. Tessier, and W. Burleson. 2015. Reinforcement learning for thermal-aware many-core task allocation. In *Proceedings of the 25th Great Lakes Symposium on VLSI (GLSVLSI'15)*. ACM, New York, NY, 379–384. DOI: https://doi.org/10.1145/2742060.2742078

[31] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally. 2013. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*. 86–96. DOI: https://doi.org/10.1109/ISPASS.2013.6557149

[32] N. Nikitin, J. de San Pedro, and J. Cortadella. 2013. Architectural exploration of large-scale hierarchical chip multiprocessors. *IEEE Trans. Comput.-Aided Des. Integ. Circ. Syst.* 32, 10 (Oct. 2013), 1569–1582. DOI: https://doi.org/10.1109/TCAD.2013.2272539

[33] U. Y. Ogras, P. Bogdan, and R. Marculescu. 2010. An analytical approach for network-on-chip performance analysis. *IEEE Trans. Comput.-Aided Des. Integ. Circ. Syst.* 29, 12 (Dec. 2010), 2001–2013. DOI: https://doi.org/10.1109/TCAD.2010.2061613

[34] C. Olah. 2017. Understanding LSTM Networks. Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[35] R. Pascanu, T. Mikolov, and Y. Bengio. 2012. Understanding the exploding gradient problem. *CoRR* abs/1211.5063 (2012).

[36] 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. 2011. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Rese.* 12 (2011), 2825–2830.

[37] Y. Qian, Z. Lu, and W. Dou. 2009. Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip*. 44–53. DOI: https://doi.org/10.1109/NOCS.2009.5071444

[38] C. Spearman. 1987. The proof and measurement of association between two things. *Amer. J. Psychol.* 100, 3/4 (1987), 441–471. Retrieved from http://www.jstor.org/stable/1422689.

[39] M. Tagel, P. Ellervee, and G. Jervan. 2010. Design space exploration and optimisation for NoC-based timing sensitive systems. In *Proceedings of the 12th Biennial Baltic Electronics Conference*. 177–180. DOI: https://doi.org/10.1109/BEC.2010.5631145

[40] V. Todorov, D. Mueller-Gritschneder, H. Reinig, and U. Schlichtmann. 2014. Deterministic synthesis of hybrid application-specific network-on-chip topologies. *IEEE Trans. Comput.-Aided Des. Integ. Circ. Syst.* 33, 10 (Oct. 2014), 1503–1516. DOI: https://doi.org/10.1109/TCAD.2014.2331556

[41] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. 2007. An 80-Tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. 98–589. DOI: https://doi.org/10.1109/ISSCC.2007.373606

[42] F. Vardi, A. Khadem-Zadeh, and M. Reshadi. 2017. A heuristic clustering approach to use case-aware application-specific network-on-chip synthesis. *J. Supercomput.* 73, 5 (01 May 2017), 2098–2129. DOI: https://doi.org/10.1007/s11227-016-1905-6

[43] S. Wallentowitz, A. Lankes, A. Zaib, T. Wild, and A. Herkersdorf. 2012. A framework for open tiled manycore system-on-chip. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. 535–538. DOI: https://doi.org/10.1109/FPL.2012.6339273

Received September 2019; revised March 2020; accepted May 2020