Breaking Cyclic Dependencies for Network Calculus using Service Partitioning

Boyang Zhou
boz319@lehigh.edu
Lehigh University

Isaac Howenstine
isaac.howenstine@gmail.com
EAB

Liang Cheng
liang.cheng@utoledo.edu
University of Toledo

Steffen Bondorf
steffen.bondorf@rub.de
Ruhr University Bochum

ABSTRACT

Network Calculus (NC) is a method for providing certification evidence in networked systems, ensuring proper functioning of time-critical traffic. Traditional NC analyses focus on feedforward networks that are networks without cyclic dependencies. However, some methods, like the fix-point method and turn prohibition, apply NC to non-feedforward networks but exhibit limitations in stability, adaptability, and flexibility. We propose an alternative method, service partitioning, supported by theorems and lemmas, to address these limitations. Service partitioning breaks cyclic dependencies in non-feedforward networks using a breaking method that leverages Quality of Service (QoS) mechanisms (Weighted Round-Robin, Static Priority, Time-Aware Shaper), by assigning flows that form cycles to distinct buffers with dedicated service allocations. This method offers enhanced flexibility by allocating different network resources to buffers based on multi-class scheduling during the breaking process. In contrast to existing methods, service partitioning does not require rerouting or additional hardware and avoids deriving invalid solutions. The paper investigates the performance of service partitioning in terms of flexibility, result tightness, adaptability, and stability to show that service partitioning is superior to existing methods. One limitation of service partitioning is that it cannot fully break cyclic dependencies in some scenarios, requiring the assist from solving methods, which can be used to apply network calculus to networks with cyclic dependencies. However, when combined with solving methods, service partitioning can still improve solution quality, reducing potentially invalid results in simulated ring networks by over 30% compared with results derived by solving methods alone.

1. INTRODUCTION

Network Calculus (NC) is a verification method that has been utilized to assess the delay performance of flows [1]. There are various toolboxes that implement different NC techniques for different types of networks [2, 3, 4]. NC can offer deterministic delay bounds for systems, such as avionics systems [5] and industrial Ethernet [6]. However, traditional NC methods (Separated Flow Analysis (SFA), Total Flow Analysis (TFA), and Pay Multiplexing Only Once (PMOO)) are typically applied to feedforward networks [7], where there is no cyclic dependency. Many systems that require delay and backlog guarantees have non-feedforward networks, which makes direct usage of traditional methods difficult [8, 9, 10].

Various methods have been developed for applying NC to non-feedforward networks, such as PMOC [11], the fix-point method [1], turn prohibition, and FP-TFA [10]. These methods are categorized into two classes, which are breaking methods and solving methods. Breaking methods break the cyclic dependencies in the network and then apply network calculus for analyses while solving methods analyze the network with cyclic dependencies using mathematical ways. These approaches, discussed in Section 3, have limitations. Some solving methods may not provide delay bounds under certain Quality of Service (QoS) mechanisms or produce valid results, while breaking methods can restrict network resources or require extra hardware.

To overcome the limitations of existing methods, we propose a novel breaking method called service partitioning. This method leverages various QoS mechanisms to break cyclic dependencies in non-feedforward networks. Modern switches typically support QoS mechanisms, such as Strict Priority (SP) and Weighted Round-Robin (WRR), enabling the implementation of service partitioning without incurring additional costs. Service partitioning splits nodes into buffer groups with dedicated resources. Besides QoS mechanisms, redundancies in MIMO communication can also create independent buffer groups [12]. In the literature, queues and buffers are used interchangeably. By assigning flows that form a cycle to separate buffer groups, service partitioning effectively eliminates their mutual dependency.

Due to a limited number of buffers, service partitioning may not always completely break cyclic dependencies. In cases with residual cycles, solving methods can be employed. Combining solving methods and service partitioning reduces invalid results, as discussed in Section 5.4. The service partitioning workflow is depicted in Figure 1. Since service partitioning splits nodes to break cyclic dependencies, the first step is to find a set of nodes to split so that all cyclic dependencies can be broken or partially broken. This step is accomplished by using Johnson's algorithm [13] to find all cyclic dependencies, followed by the use of either integer programming or a greedy algorithm to identify candidate nodes to split. The second step is to split each node identified in the first step into several buffer groups and assign

flows that form cyclic dependencies to different buffer groups in order to eliminate their mutual dependencies. The third step is to assign dedicated service curves to different buffer groups in the split nodes. The last step is to choose different methods to analyze the network performance. The method selection is based on whether there are remaining cycles in the network. Further explanation of the diagram is provided in Section 4. The main challenges of service partitioning are proving the validity of service partitioning and identifying a suitable set of nodes to split.

Contributions: Our main contributions are:

1. We propose service partitioning, which utilizes Quality of Service (QoS) mechanisms to break cyclic dependencies in non-feedforward networks. Its validity is substantiated by Theorem 4.1, which proves that cyclic dependencies can be broken by splitting a set of nodes in the cycle.
2. We propose to use either an integer programming approach or a greedy algorithm to identify a set of nodes to split for breaking cyclic dependencies using service partitioning based on Theorem 4.1.
3. We perform comprehensive numerical analyses to examine the flexibility and scalability of service partitioning. Furthermore, by comparing the results of service partitioning and existing methods, we demonstrate its superiority in terms of scalability, stability, and adaptability.
4. We demonstrate that even when service partitioning cannot completely break all cyclic dependencies, it enhances the performance of solving methods, decreasing the time complexity of solving methods and reducing invalid results by over 30%.

Section 2 presents the NC background. Section 3 shows existing methods for analyzing non-feedforward networks using NC. Section 4 introduces service partitioning and its implementation details. Section 5 evaluates the performance of service partitioning. Section 6 concludes the article.

2. BACKGROUND

In this section, the basic knowledge of NC is introduced. NC provides a set of theorems that derive deterministic delay bounds of networks based on a mathematical model using min-plus algebra. Min-plus algebra provides convolution and deconvolution operations of two functions f_1 and f_2, which are defined as:

$$\textit{convolution} \quad (f_1 \otimes f_2)(d) = \inf_{0 \le s \le d} \{f_1(d-s) + f_2(s)\}, \quad (1)$$

$$\textit{deconvolution} \quad (f_1 \oslash f_2)(d) = \sup_{u \ge 0} \{f_1(d+u) - f_2(u)\}. \quad (2)$$

Since NC is used to derive the worst-case delay bound, flows are modeled by their worst-case arrival processes (Arrival curves), and nodes (i.e., switches and routers) are assumed to provide minimum service (Service Curves).

DEFINITION 2.1. *An arrival curve $\alpha(t)$ is a bounding envelope that characterizes all possible arrival processes. Given an arrival process $F(t)$ describing the cumulative amount of data arriving at a node in the network up to time t, a non-negative, non-decreasing function $\alpha(t)$ defined for any $t \ge 0$ is an arrival curve of $F(t)$ if and only if*

$$\forall t \ge s \ge 0 : F(t) - F(s) \le \alpha(t-s). \quad (3)$$

This paper uses leaky bucket arrival curves, shown in Eq. 4.

$$\alpha_{\rho,b}(t) = \begin{cases} \rho t + b & t > 0 \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

DEFINITION 2.2. *Given a flow passing a node with an incoming arrival process $F^{in}(t)$ and an outgoing process $F^{out}(t)$, where $F^{in}(t)$ and $F^{out}(t)$ depict the cumulative amount of data up to time t coming to and leaving the node, a non-negative, non-decreasing function $\beta(t)$ is a service curve for that node if and only if*

$$F^{in}(t) \ge F^{out}(t) \ge F^{in}(t) \otimes \beta(t), \textit{where } \beta(t) \ge 0. \quad (5)$$

Rate-latency service curves are used in this paper, which is shown in Eq. 6, where $[a]^+$ equals to $\max\{a, 0\}$.

$$\beta_{R,T}(t) = [R(t-T)]^+ \quad (6)$$

It is ubiquitous that several flows might pass the same node in real-world applications. In this case, leftover service curves need to be calculated. The definition of leftover service curves is as follows:

DEFINITION 2.3. *Suppose two flows f_1 and f_2, which have arrival curves α_1 and α_2, pass a lossless node with FIFO multiplexing. The service curve of the node is $\beta(t)$. The leftover service curve β_θ^1 of f_1 is calculated as follows (Proposition 6.2.1 in [1]):*

$$\beta_\theta^1 = [\beta(t) - \alpha_2(t-\theta)]^+ 1_{\{t>\theta\}}. \quad (7)$$

θ depends on the latency of the service curve and rates and bursts of the arrival curves. There are different derivations for θ [14]. Note that the default calculation for θ from NetworkCalculus.org DNC [7] is used in this paper.

The arrival curve of a flow will change once it passes a node. If a flow has a path length larger than one node, it is necessary to calculate the outgoing arrival curve of the flow in each node, which is also the incoming arrival curve of the next node on its path.

DEFINITION 2.4. *Given an incoming arrival curve α^{in} of a flow and a service curve β of a node, the arrival curve of the outgoing flow can be calculated using Eq. 8.*

$$\alpha^{out}(t) = (\alpha^{in} \dot{\oslash} \beta)(t) = \begin{cases} 0 & t = 0 \\ (\alpha^{in} \oslash \beta)(t) & \textit{otherwise} \end{cases} \quad (8)$$

After introducing how to calculate the arrival curve and the service curve. We can then derive the worst-case delay bound as follows:

$$\textit{delay} : \forall t \ge 0 : D(t) \le \inf\{d \ge 0 | (\alpha \oslash \beta)(-d) \le 0\}. \quad (9)$$

3. RELATED WORK

Finzi et al. [9] identify two types of methods for addressing the non-feedforward limitation. One is to break the cyclic dependencies, which is called the breaking method. The other is to compute the bounds while taking cyclic dependencies into account, defined as the solving method. Service partitioning is a breaking method. Table 1 compares existing methods with service partitioning. Service partitioning can overcome most of the limitations encountered by existing methods.

Table 1: Comparison of service partitioning and existing breaking and solving methods

Property \ Method	Turn prohibition	Regulator insertion	Service partitioning	solving methods
Rerouting	Requires rerouting	Not required	Not required	Not required
Resource restriction	Restrict the usage of some resources	No restriction	No restriction	No restriction
Extra hardware	Not required	Required (regulators)	Not required	Not required
Invalid results	No	No	No	May produce invalid results
Applicability	Not applicable to some scenarios [9]	High applicability	High applicability	Applicability to WRR or TAS QoS mechanisms is an open research topic
Flexibility	No flexibility	No flexibility	High flexibility by allocating service curves arbitrarily	No flexibility

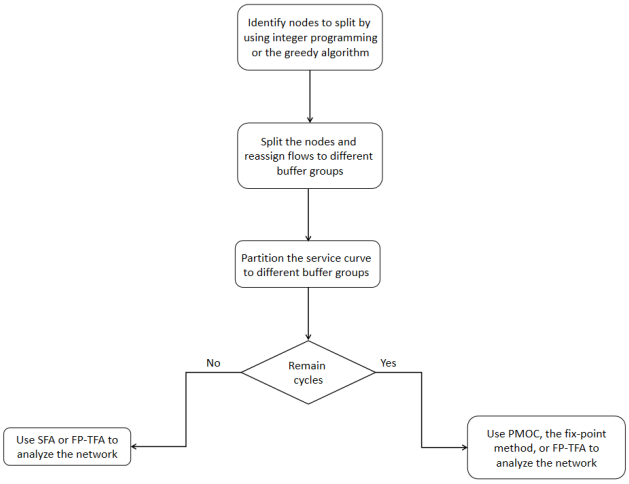

Figure 1: The workflow for applying service partitioning to non-feedforward networks

3.1 Turn Prohibition

Turn prohibition [8] relies on graph-theoretic methods to strategically underutilize certain network resources to prevent cyclic dependencies. It restricts the usage of network resources and requires rerouting to break the cyclic dependencies. However, this method does not always work, as in some cases, the traffic load might be higher than the available capacity after the restriction [9], preventing the usage of turn prohibition. Additionally, these methods are topology-based and do not consider the actual flow paths, resulting in suboptimal solutions. Better results can be achieved by taking the flows into consideration.

3.2 Regulator Insertion

Thomas et al. [10] propose a method to break the cyclic dependencies by inserting regulators into the non-feedforward network. Due to the "shaping-for-free" principle [1, 10], adding regulators does not induce extra delays. However, "shaping-for-free" is not always a fair claim because it ignores the processing delay of the shaper (See Lemma 1.5.2 in [1]). Since regulators need to parse the header and calculate the sending time of each packet, the influence of the processing delay still needs to be evaluated. Furthermore, this method requires additional hardware.

3.3 Solving Methods

The three solving methods are the fix-point method [15, 1], PMOC [11], and FP-TFA [10]. The fix-point method is a standard method for dealing with non-feedforward networks. PMOC is another method for non-feedforward networks. The difference between the fix-point method and PMOC is that PMOC considers how flows are serialized by paying the bursts only at convergence points, which are the nodes where two flows join and begin traveling down a common subpath [11]. Both methods are applied to networks with arbitrary multiplexing. FP-TFA is a method to derive the worst-case delay bound in both feedforward and non-feedforward networks with FIFO multiplexing based on TFA++ [16]. It uses a similar technique to the fix-point method. Applied to a network without cyclic dependencies, it derives the same result as TFA++ mentioned above [17].

These methods may provide unrealistically large results and produce negative results due to the necessary matrix inverse calculation (Section 6.3.2 in [1]). Moreover, these methods cannot bound the delay of flows in switches with QoS mechanisms that assign proportional service curves, such as WRR and TAS, as discussed in Section 5.2.

4. SERVICE PARTITIONING

Service partitioning essentially involves subdividing nodes into independent buffers or groups of buffers. Depending on the QoS mechanisms implemented by the switches in a network, a single output port can be transformed into multiple parallel subnodes or groups of buffers. These groups operate concurrently, each with its own dedicated leftover service curve for the flows that pass through them. Although a buffer group shares network resources with other groups, these shared resources are isolated from one another via QoS mechanisms. Contention for network resources among individual flows only occurs within the same buffer group. Service partitioning can be effectively implemented using the existing QoS mechanisms, including Strict Priority (SP), Weighted Round Robin (WRR), and Time-Aware Shaper (TAS). To illustrate this concept, consider Figure 2 (a), which represents a network with three flows, f_1, f_2, and f_3, forming a cyclic dependency. In NC, the dependency occurs when multiple flows share the resources of the same node. Thus, dependencies between f_1 and f_2, between f_2 and f_3, and between f_3 and f_1 occur at ports 5, 10, and 20, respectively. Consequently, the cyclic dependency in this network becomes $10 \rightarrow 20 \rightarrow 5 \rightarrow 40$, where $\rightarrow$ denotes the order and direction since directed graphs are used. Disrupting the dependency of flows among ports 10, 20, or 5 effectively breaks the cyclic dependency. Figure 2 (b) provides an example of service partitioning, where port 5 from Figure 2 (a) is subdivided into lower-level buffer groups, thereby eliminating the cyclic dependency in the network. Port 5 is split into two buffer groups with no contention for network resources between them. A reassignment of violating traffic flows to different buffer groups can break the cyclic dependency. In

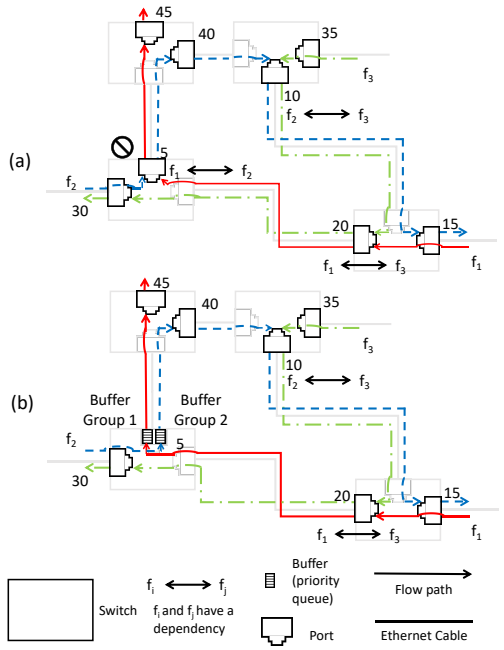

Figure 2: This figure illustrates the working mechanism of service partitioning. Part (a) shows the scenario where there is a cyclic dependency formed by three flows in the network. The transition from part (a) to (b) shows that converting certain output ports to buffer groups can break the cycles. This is done by eliminating the dependency/contention between f_1 and f_2 at port 5 .

this scenario, flow f_1 is assigned to Buffer Group 1, and flow f_2 is assigned to Buffer Group 2.

Without loss of generality, we assume that flows in non-feedforward networks are not assigned specific priorities in advance, meaning that all flows can be assigned to any priority. In case there is a pre-defined priority assignment, we may create new contention in another buffer group. Then, service partitioning needs to be run repeatedly. This scheme terminates when either the scheme converges (i.e., no new contention among individual flows is created in a buffer group) or a solving method is applied to compute the delay bound despite the presence of cyclic dependencies.

Service partitioning involves three key steps: identifying nodes to split, reassigning flows to different buffer groups, and partitioning the service curve. The choice of NC method applied depends on whether there are remaining cyclic dependencies after the breaking process. Figure 1 depicts the workflow for applying service partitioning. Identifying nodes is a critical component of the process, as the other two steps are relatively straightforward in comparison.

In order to proceed with the breaking process of service partitioning, it is necessary to clarify the definitions of three terms: sides, shared nodes, and penalty nodes. Figure 2 (a) is used to clarify these definitions.

Definition 4.1. *A side refers to the set of nodes in a cycle that intersects with the nodes along a specific flow path. Let's consider a cycle C with nodes $1 \to 2 \to ... \to n$ and a flow f with a path of $fp_1 \to fp_2 \to ... \to fp_m$. Then, the side s with nodes $s_1 \to s_2 \to ... \to s_k$ of flow f and cycle C must meet the following conditions:*

$$k \geq 2, \tag{10}$$

$$s_i \in C \wedge s_i \in f, \forall 1 \leq i \leq k, \tag{11}$$

$$s_{(i+1)\%n} - s_i = \begin{cases} 1 & 1 \leq i < n \\ 1-n & i{=}n \end{cases}. \tag{12}$$

10 ensures that each side must have at least two nodes. 11 indicates that all nodes in a side must be in both the corresponding cycle and the corresponding flow. 12 guarantees that the consecutive nodes in a side must appear consecutively in the corresponding cycle without any skipped nodes in between. For example, f_2 in Figure 2 (a) has a side of $5 \to 40 \to 10$ with the cycle. One flow can have multiple sides with a cycle.

Sides can be classified into two types: sub-sides and super-sides. In a cycle, a sub-side is a side that is fully contained within another side in the same cycle. Any side that is not a sub-side can be considered a super-side.

Definitions of shared nodes and penalty nodes are defined below:

Definition 4.2. *Shared nodes within a cycle are defined as the nodes that are traversed by more than one of the flows that form the cycle, which are flows having validated sides with the cycle.*

In Figure 2 (a), ports 5, 10, and 20 are shared nodes since each of them is traversed by two flows, while port 40 is not a shared node.

Definition 4.3. *Penalty nodes are shared nodes that are crossed by more than eight flows forming the same cycle, under the assumption that there are at most eight queues in each node when using WRR, SP, or TAS.*

There is no penalty node in Figure 2 (a) because ports 5, 10, and 20 are only traversed by two flows forming the cycle, while at least nine flows are required for penalty nodes.

Splitting a non-penalty shared node can partition the shared node into a set of non-shared subnodes. This can be achieved by simply assigning flows forming the cycle to different queues. However, splitting a penalty node cannot fully eliminate the competition for network resources among the flows forming the cycle, because at least two flows are assigned to the same queue in a penalty node, which means that the competition still exists after splitting a penalty node. Therefore, a penalty is applied when a penalty node is split. Knowing the definition of sides, shared nodes, and penalty nodes, we leverage a collection of theorems and lemmas to demonstrate the process of breaking cyclic dependencies through service partitioning. This is accomplished by manipulating and managing the sides, shared nodes, and penalty nodes involved in the cycle.

Theorem 4.1. *Splitting any $k-1$ non-penalty shared nodes of any super-side will break the cycle, where k is the number of shared nodes on the super-side. If there are fewer than $k-1$ non-penalty shared nodes in any super-side, the cyclic dependency cannot be fully broken.*

The correctness of the theorem is proved by the following three lemmas.

LEMMA 4.1.1. *A side must have at least two shared nodes.*

PROOF. A cyclic dependency is caused by the contention for network resources. A side must have at least two shared nodes because flows have to touch each other at two nodes to form a cycle. □

LEMMA 4.1.2. *Splitting all shared nodes of a side except one will break the side if all split shared nodes are not penalty nodes.*

PROOF. Based on Lemma 4.1.1, there should be at least two shared nodes for each side that forms the cycle. If a non-penalty shared node is split, it can be partitioned into several non-shared subnodes. Thus, splitting all shared nodes except one can break the side because there is only one shared node left if all split nodes are non-penalty nodes. □

LEMMA 4.1.3. *Breaking any super-side of a cycle will break the cyclic dependency.*

PROOF. When breaking a super-side s_1 of a cycle, only one shared node remains within s_1. If the cycle still exists, it implies the existence of another side s_2 that encompasses all nodes from s_1 with two or more shared nodes. However, this contradicts the definition of a super-side, which states that a super-side cannot be fully contained within another side in the same cycle (as per Lemma 4.1.2). □

Service partitioning cannot fully break the cyclic dependency *if and only if all super-sides of a cycle have more than one penalty node*. In this case, the penalty nodes can be split to partially break the cycles. A key challenge in service partitioning is determining the set of nodes to split in order to break or partially break all cyclic dependencies in the network. Thomas et al. [10] propose a minimum feedback arc set (MFAS) problem for finding the fewest edges to break a set of cycles using regulator insertion. However, since service partitioning breaks super-sides instead of edges, MFAS cannot be applied to service partitioning. Thus, we propose two methods, integer programming and a greedy algorithm, to identify nodes to split.

4.1 Identifying the Subset of Nodes to Split

In this section, we propose integer programming and a greedy algorithm for identifying nodes to split to break cyclic dependencies. Theorem 4.1 provides a guideline for splitting nodes to break cyclic dependencies. However, there are different sets of nodes that can be split to break cyclic dependencies based on the theorem. The greedy algorithm greedily selects one set of nodes without a global view, while integer programming tries to minimize the overall splitting cost, which is the weighted sum of the number of nodes to split and the penaly for splitting penalty nodes (see Eq. 13). Table 2 denotes all notations we use.

4.1.1 Integer Programming

An integer programming problem is formulated to identify the nodes that need to be split while minimizing the cost.

$$\min \sum_{i=1}^{m} c_i + W \cdot \sum_{i=1}^{m} \epsilon_i \cdot c_i \tag{13}$$

Objective Function: Function 13 is the objective function, aiming to minimize the cost function. $\sum_{i=1}^{m} c_i$ is the number of shared nodes to split, and $\sum_{i=1}^{m} \epsilon_i \cdot c_i$ is the sum of the penalty for splitting penalty nodes. A shared node i might be a penalty node in one cycle, while it might be a non-penalty node in another cycle. In order to simplify the problem, a positive ϵ_i is assigned if node i is a penalty node in any cycle. Different penalties are given to different penalty nodes. ϵ_i equals $\max_j\{nf_{ij}\} - 8$ when node i is a penalty node and 0 otherwise. W is a weight to balance the importance of the number of split nodes and the penalty. When W is large, splitting nodes with less penalty is dominant. Otherwise, splitting fewer nodes is preferred.

Table 2: Table of Notations

m	Number of shared nodes in all cycles
n	Number of cycles
N_i	Number of super-sides in cycle i
S_{ij}^k	Binary: $S_{ij}^k = 1$ indicates that the shared node k is on the jth super-side of cycle i. Otherwise, $S_{ij}^k = 0$
nf_{ij}	The number of flows forming cycle j at shared node i
ϵ_i	Penalty parameter for splitting shared node i, where $\epsilon_i = \max\{\max_j\{nf_{ij}\} - 8, 0\}$
t_{ij}	Number of shared nodes of the jth super-side in cycle i, where $t_{ij} = \sum_{k=1}^{m} S_{ij}^k$
W	Weight parameter for the objective function
	Decision Variables
c_i	Binary Variable: $c_i = 1$ indicates that the ith shared node is split. Otherwise, $c_i = 0$
B_{ij}	Binary Variable: $B_{ij} = 1$ indicates that the jth super-side of cycle i is broken or partially broken. Otherwise, $B_{ij} = 0$

Constraints: 14-15 are the constraints for the optimization problem. Constraint 14 guarantees that each cyclic dependency should have at least one super-side broken or partially broken. Constraint 15 ensures that when the jth super-side of cycle i is broken or partially broken (i.e., $B_{ij} = 1$), there should be at least $t_{ij} - 1$ shared nodes split on the side. In this way, a super-side can be successfully broken or partially broken according to Lemma 4.1.2.

$$\text{s.t.} \sum_{j=1}^{N_i} B_{ij} \geq 1, \text{for all } 1 \leq i \leq n \tag{14}$$

$$\sum_{k=1}^{m} S_{ij}^k \cdot c_k \geq B_{ij} \cdot (t_{ij} - 1), \text{for all } 1 \leq i \leq n, \quad 1 \leq j \leq N_i \tag{15}$$

4.1.2 Greedy Algorithm

Since integer programming is np-hard, we also propose an alternative greedy algorithm to identify nodes to split. The greedy algorithm splits one super-side of each cycle greedily. For each cycle, one of two cases is encountered. In the first case, the cycle can be fully broken. There must be a set of super-sides containing less than two penalty nodes in the cycle. In this case, the greedy algorithm will select the

shortest super-side in the set to break. In the second case, all super-sides have no fewer than two penalty nodes. Then, the greedy algorithm will break the shortest super-side among all super-sides in the cycle.

4.2 Traffic Class Reassignment

Once the set of nodes to split has been identified, the next step is to reassign traffic classes for the flows passing over the split nodes. To prevent service contention, flows forming a cyclic dependency must be allocated to separate buffers, allowing for dedicated service curves to be assigned to each flow. If the network uses TAS, WRR, or SP, eight queues corresponding to eight different priorities are assumed. Therefore, if no more than eight flows causing a cyclic dependency cross the same node, the flows can be easily assigned to different buffers.

If a penalty node is split, the cyclic dependency cannot be fully broken, and solving methods can be stacked on top of service partitioning. In this case, the network after the breaking process is a new non-feedforward network with fewer cyclic dependencies compared with the original one. Solving methods can be directly applied to the network. However, service partitioning can still benefit the solving methods in three ways: i) the flows forming the cycles can be allocated to the same FIFO queues, so all three solving methods can be used to bound the delay performance of the remaining cycles; ii) it reduces flows formulating cyclic dependencies in the network, which can reduce the matrix size in solving methods, shortening the time consumption of solving methods; iii) it can reduce the invalid solutions produced by solving methods (See Section 5.4). Since service partitioning reduces the time consumption of solving methods and invalid solutions produced by solving methods, it increases the scalalability of solving methods.

4.3 QoS Mechanisms for Service Allocation

Once the nodes are split and traffic flows are reassigned, the next step is to determine the service allocation. There is no fixed scheme for service allocation, as it can vary based on the specific requirements of the network. Service partitioning enables flexible control of delay performance by utilizing various QoS mechanisms. In this paper, we introduce three QoS mechanisms for service allocation: Strict Priority (SP), Time-Aware Shaper (TAS), and Weighted Round Robin (WRR). The calculations for service allocation in these mechanisms are outlined as follows.

4.3.1 Strict Priority

SP always forwards the packets in the highest priority queue among non-empty queues. Suppose the service curve of the node is β, and α_i is the sum of arrival curves of flows having priority i. Priority j has a higher priority than priority i if $j < i$. The service curve β_i assigned to the queue with priority i can be calculated using Eq. 16. Note that $[a]^+ = \max\{0, a\}$. Each priority queue is allocated its own service curve if the arrival curves and priorities of all flows are known.

$$\beta_i = [\beta - \sum_{j<i} \alpha_j]^+ \tag{16}$$

4.3.2 TAS Node Splitting

TAS can reserve service curves for different groups by assigning different time slots to different groups of buffers. Service curve calculations for different buffer groups are proposed in [18] [19]. TAS allocates proportional service curves to different groups, meaning that the rates of service curves assigned to buffer groups are the proportions of the rate of the output port. The sum of rates assigned to all groups should be no greater than the rate of the output port.

4.3.3 Weighted Round-Robin

WRR is a QoS mechanism that can allocate network resources to different groups of buffers by assigning different weights. A larger weight assigned to the group means that this group of buffers is allotted more network resources. Suppose there are m groups of buffers in WRR. w_i is the weight assigned to group i. Suppose C is the bandwidth of the node, and the node has 0 s latency. The leftover service curve β_i for group i can be calculated using Eq. 17 [20]. WRR allocates proportional service curves to different buffers. IWRR, or Interleaved WRR, is a variation of the WRR QoS mechanism that serves groups of buffers in a round-robin way. In IWRR, the leftover service curve β_i for group i can be calculated using Eq. 17 [20], where l_i^{min} and l_i^{max} represent the smallest and largest packet size of group i, respectively. WRR, including IWRR, allocates proportional service curves to different buffers.

$$\beta_i = [\frac{w_i l_i^{min}}{w_i l_i^{min} + \sum_{j\neq i} w_j l_j^{max}} \cdot C(t - \frac{\sum_{j\neq i} w_j l_j^{max}}{C})]^+ \tag{17}$$

In this section, we have introduced service partitioning and verified its validity using Theorem 4.1. Since service partitioning is built upon graph theory to break cyclic dependencies, it can be applied to any network topology and traffic pattern if switches support WRR, SP, TAS, or other QoS mechanisms that provide dedicated service to queues.

5. EVALUATION

Section 5.1 evaluates the scalability of service partitioning by investigating the performance of integer programming and the greedy algorithm for identifying split nodes. Note that in the other three subsections, we use balanced ring networks for evaluation. The greedy algorithm and integer programming will produce the same set of nodes to split since there is only one cyclic dependency and all nodes in the network have the same traffic load in each scenario. Thus, using integer programming or the greedy algorithm does not influence the final analysis result. Section 5.2 contains the evaluation of the flexibility of service partitioning using IWRR and an explanation of why the fix-point method and PMOC cannot bound delays in networks using QoS mechanisms with proportional service curve allocation. Section 5.3 compares service partitioning against existing methods. Section 5.4 illustrates how solving methods benefit from service partitioning in terms of stability and adaptability.

5.1 Scalability of Service Partitioning

In this section, we aim to evaluate the scalability of service partitioning and the tradeoff between the splitting cost and time consumption for this technique. The main factor that could impact the scalability of service partitioning is its time consumption, which comprises two components: (i) the time consumption of network calculus calculation and (ii) the time consumption for identifying nodes to split. Since the time complexity of network calculus calculation with leaky bucket arrival curves and rate-latency service curves

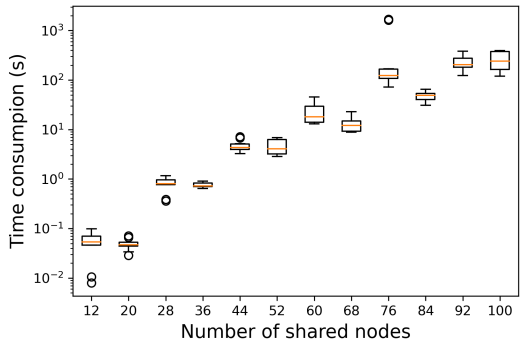

Figure 3: The impact of shared nodes on the time consumption using integer programming. Note that the median time consumption of the greedy algorithm is less than 10 ms in all scenarios

is relatively low (i.e., $O(M\bar{N})$, where M is the number of flows and $\bar{N}$ is the average path length of all flows), the dominant factor of the time consumption of service partitioning is the time consumption of identifying nodes to split.

To evaluate the scalability, we generate different traffic patterns and network topologies. In the simulated networks, each cycle contains 12 shared nodes and four flows. Every increment of eight shared nodes will create a new cycle because the new cycle will share the 4 shared nodes with one of the previous cycles. This setup guarantees that more shared nodes generate more cycles. We use Johnson's algorithm to find cycles and OR-Tools [21] to solve the integer programming problems. We conduct 10 experiments for each network and create box plots of the time consumption. Evaluations ran on a machine with an i7-10870H CPU.

We evaluate the time consumption of both integer programming and the greedy algorithm running on the simulated network. The time consumption of the greedy algorithm is negligible compared to that of the integer programming algorithm. Even when there are 100 shared nodes in the network, the median time consumption is still less than 10 ms by applying Johnson's algorithm and our greedy algorithm. Thus, it is not shown in Figure 3. Figure 3 depicts the impact of the number of shared nodes on the time consumption of identifying nodes to split using integer programming. The median time consumption generally increases as the number of shared nodes grows, though there are instances where a network with fewer shared nodes has a longer time consumption. Additionally, the time consumption of solving the same optimization problem has a wide range, as seen in the network with 76 shared nodes, where the solving time ranges from 71 s to 1691 s. This variation is attributed to the heuristic solving algorithm, which is influenced by the number of iterations, which in turn depends on the number of branch and bound nodes and can vary across solving processes for the same problem. As a result, there is no guarantee for the time consumption of solving the optimization problem.

Generally, the greedy algorithm is much faster than integer programming algorithms in service partitioning, often being hundreds to thousands of times faster. However, there is a tradeoff between splitting cost and time consumption in service partitioning. The splitting cost is defined in Function 13. Figure 4 shows the comparison of splitting cost between integer programming and the greedy algorithm. Since the greedy algorithm is stochastic when there is a tie, we consider both the worst-case and expected splitting costs of the greedy algorithm for comparison. Integer programming achieves optimal results (see Figure 4) while the greedy algorithm has a much higher splitting cost. The gap between the costs of integer programming and the greedy algorithm scales with the number of nodes.

Service partitioning demonstrates excellent scalability when using the greedy algorithm, enabling its application to large systems. However, the greedy algorithm lacks a global view, and its results are usually suboptimal. On the other hand, integer programming can achieve an optimal solution at the cost of high time consumption. Users should select the appropriate algorithm based on their specific needs.

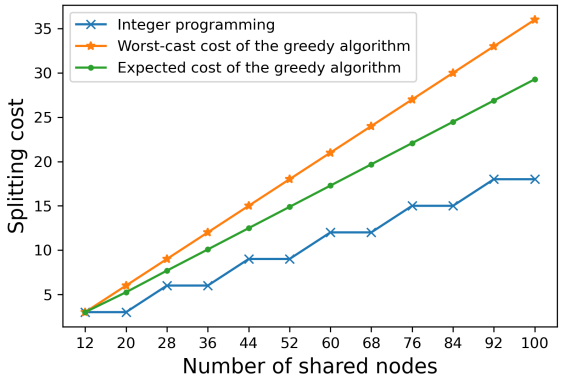

Figure 4: Comparison of integer programming and the greedy algorithm in terms of splitting cost

5.2 Flexibility of Service Partitioning

In the existing solving methods, FP-TFA only works for FIFO multiplexing. PMOC and the fix-point method fail to provide a bounded delay performance for QoS mechanisms that allocate proportional service curves, such as WRR and TAS, since the methods do not consider the proportionality of the service curves. PMOC and the fix-point method will derive the same invalid delay bounds for all weight assignments in IWRR. However, service partitioning can be

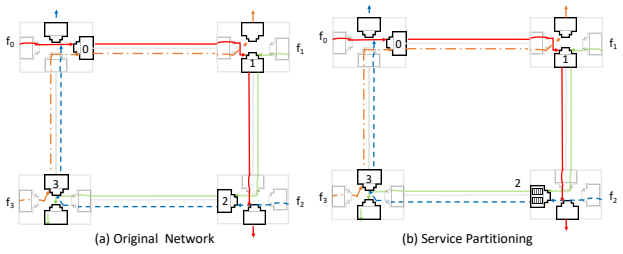

Figure 5: An example of a four-node ring network. Part (a) shows the network topology and the traffic pattern. Part (b) shows how to break the cyclic dependencies using the service partitioning by splitting Node 3. See Figure 2 for symbol definitions

used in such cases, providing flexible control of the delay performance in the network by adjusting the service curve allocations at the split nodes.

A ring network was utilized to demonstrate the flexibility of service partitioning in the study. Figure 5 (a) depicts a ring network comprising four nodes (i.e., output ports) [9], labeled from 0 to 3, which form a cyclic dependency. Flows f_0, f_1, f_2, and f_3 are involved in the cycle, with each flow having a side of length 2 with the cycle. The study assumed that all flows had the same leaky bucket arrival curve with a 10 Mbps rate and a 1 Mb burst. Each node in this ring network had a bandwidth of 100 Mbps and a latency of 0 s. Setting the latency to be 0 s was to avoid the dominant influence of the invariant service latency of nodes on the total delay bounds and better reflect the impact of weight assignments on the delay bounds. To break the cyclic dependency, Node 2 is split, as shown in Figure 5 (b). To examine how service partitioning affects the delay performance of the network, IWRR was employed to partition the service curve of Node 2 into two groups. Weights w_1 and $1 - w_1$ were assigned to f_1 and f_2, respectively.

Figure 6 shows the influence of weight assignment on the delay performance of the network. The delay bound of f_1 is approximately 100 ms when w_1 is 0.15, and decreases to less than 40 ms as w_1 increases to 0.85. Delay bounds of f_0 and f_3 change slightly with w_1 because the outgoing arrival curve of f_2 at Node 3 is influenced by w_1, which has an impact on f_3 at Node 3 and then f_0. Figure 6 shows that service partitioning enables the control of delay performance by different service curve allocations.

5.3 Comparison between Service Partitioning and existing methods

This section evaluates the performance of service partitioning by comparing its results with those from turn-prohibition, the fix-point method, FP-TFA, and PMOC. Regulator insertion is not evaluated because its influence on the processing delay is unknown. We employ SP as the QoS mechanism for all methods except FP-TFA, since it only supports FIFO. The performance of all methods is assessed in a series of ring networks, considering the tightness and validity of the results, as well as the limitations of each method. It is important to note that only the tightness of delay bounds derived by PMOC, the fix-point method, and service partitioning can be compared, as turn prohibition utilizes rerouting and FP-TFA only supports FIFO.

The ring topology is used in the evaluation. Each ring network is characterized by two parameters, N and L. N is the number of nodes/output ports in the network, and L is the number of nodes passed by each flow (i.e., side length). Each node in the ring topology is an entry point of a flow. Thus, there is a total of N flows in a ring network, and each node is passed by L flows. The topology shown in Figure 5 (a) is a ring topology with $N = 4$ and $L = 2$. This type of ring network has been used in the evaluation of breaking and solving methods in [11, 9].

To assess the performance of all five methods, evaluations are conducted using topologies with varying N and L parameters. Each node in the network is equipped with a rate-latency service curve, featuring a bandwidth of 100 Mbps and a latency of 0.1s. The chosen 0.1s latency value is intended to highlight the impact of flow length on delay performance, as rerouting in turn prohibition can alter the

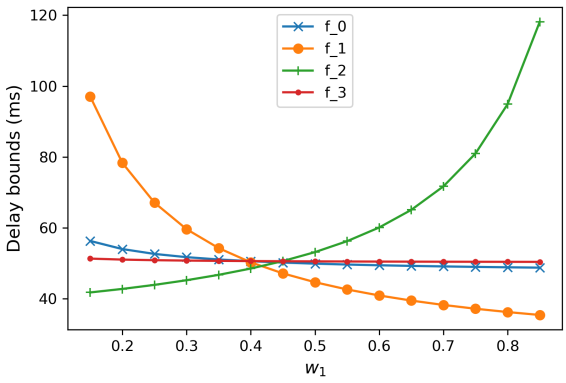

Figure 6: The influence of weight assignment on the delay performance of the network in Figure 5

flow length. All flows possess identical burst sizes of 0.1 Mb, while their rates range from 1 Mbps to 10 Mbps. For service partitioning, the service curve assigned to each queue is computed using Eq. 16. In the evaluation, the maximum value of the worst-case delay bounds across all flows, referred to as the maximum delay bound, is utilized to effectively constrain the delay experienced by all flows.

Figure 7 shows the influence of L on the largest worst-case delay bounds of the five methods when $N = 8$ and $N = 16$. L ranges from 2 to $N - 1$. The x-axis shows L, and the y-axis shows the largest worst-case delay bound of all flows. All flows have a rate of 1 Mbps. Figure 7 (a) presents the largest worst-case delay bounds of the five methods when $N = 8$. In these instances, all cyclic dependencies can be resolved using service partitioning since $L \leq 8$ and L is the number of flow passing each node. Consequently, the minimum result from FP-TFA and SFA is considered as the outcome of service partitioning after the breaking process. Service partitioning yields similar results to PMOC and the fix-point method. Additionally, FP-TFA can also produce feasible outcomes in such cases, although its results cannot be compared with others.

The maximum delay bound derived by turn prohibition initially decreases and then increases as L increases. This behavior can be attributed to rerouting. Rerouted flows have a flow length of $N-L$. when $L < 4$, rerouted flows have the largest flow length, leading to large delay bounds. This shows the limitation of turn prohibition in some scenarios, where the network topology is a sparse graph with limited path selections for rerouting.

Figure 7 (b) illustrates the largest worst-case delay bounds of the network when $N = 16$. It is worth noting that when $L > 8$, all shared nodes become penalty nodes. Consequently, the minimum feasible delay bound derived by FP-TFA, PMOC, and the fix-point method is utilized to model the remaining cyclic dependencies in service partitioning.

The largest worst-case delay bound of the turn-prohibition method exhibits a similar trend to that shown in Figure 7 (a). However, a noteworthy observation is found at $L = 2$. At this point, the largest delay bound is smaller than that when $L = 3$. This occurrence can be attributed to an increased number of flow reroutings contending for network resources when $L = 3$ compared to $L = 2$.

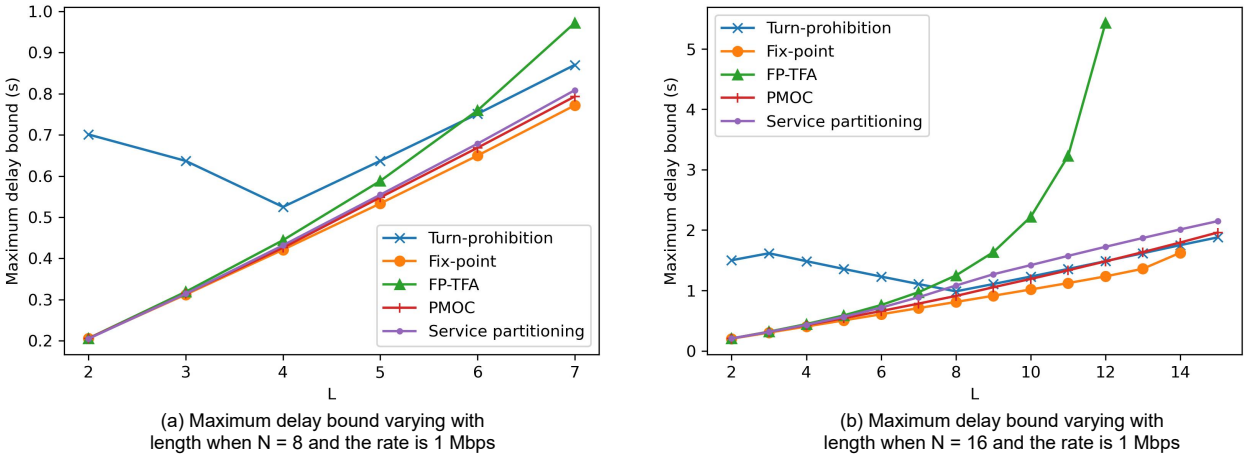

Figure 7: The largest delay bounds of five methods with different flow lengths when $N = 8$ and $N = 16$

In terms of the tightness of delay bounds, the fix-point method outperforms PMOC and service partitioning when $L \leq 13$. The curves of the fix-point method and FP-TFA terminate at $L = 13$ and $L = 12$, respectively, as both methods yield unrealistically large or negative results beyond these points. Service partitioning provides results slightly larger than those from PMOC.

Figure 8 illustrates the impact of flow rates on delay performance in two distinct scenarios. The rate of all flows varies from 1 Mbps to 10 Mbps. The x-axis represents the rate of each flow, while the y-axis denotes the largest worst-case delay bound. Due to producing invalid solutions in most cases, the results of the fix-point method and FP-TFA are not displayed in this figure.

Figure 8 (a) shows the scenario with $N = 8$ and $L = 4$. As the flow rate increases, the largest worst-case delay bounds derived by all three methods also increase. Notably, service partitioning yields tighter delay bounds than PMOC when the rate exceeds 5 Mbps. However, when the rate is no greater than 5 Mbps, both methods exhibit similar performance. Figure 8 (b) portrays the scenario when $N = 16$ and $L = 8$. In this case, PMOC achieves tighter delay bounds than service partitioning when the rate is below 4 Mbps. However, when the rate reaches or exceeds 9 Mbps, PMOC yields invalid results. It is worth mentioning that turn prohibition performs well in both scenarios, as the rerouting does not increase the flow length when $L = N/2$.

In this part of the evaluation, we have shown that service partitioning is superior to existing methods. Compared with turn prohibition, service partitioning does not lead to a significant increase in the delay bound caused by rerouting. Compared with solving methods, service partitioning does not produce invalid solutions and provides competitive delay bounds.

5.4 Adaptability & Stability by Combination of Service Partitioning & Solving Methods

Based on the previous evaluation, it was observed that FP-TFA, the fix-point method, and PMOC may produce invalid results in certain scenarios. Therefore, we further investigate the adaptability and stability of these methods. A method is considered to have good adaptability if it avoids generating negative results, while good stability implies that the method does not result in a significant increase in the delay bound when there is minimal network variation, leading to unrealistically large outcomes. Both unrealistically large results and negative results are categorized as invalid outcomes.

In service partitioning, when there are remaining cyclic dependencies, it becomes necessary to combine the solving methods. The analysis in this section demonstrates that the combination of solving methods and service partitioning exhibits superior stability and adaptability compared to using solving methods alone. For the evaluation, a total of 880 ring scenarios were generated. The parameters used include $N = 16$, L ranging from 2 to 12, flow rates ranging from 1 Mbps to 8 Mbps, and burst sizes ranging from 0.1 Mb to 1 Mb. All nodes in the network have a bandwidth of 100 Mbps and a latency of 0.1 s. In order to evaluate the stability, we need to define unstable results. A delay bound is considered unstable if it increases by a factor of 1.5 given a minimal change in the network parameters. The minimal changes in this evaluation are: an increase in flow length by one hop, an increase in burst size by 0.1 Mb, and an increase in flow rate by 1 Mbps. Unstable results are more likely to become unrealistically large results (observe the delay bound derived by the PMOC when the rate is 8 Mbps in Figure 8 (b)). When the number of unstable results for a method is large, it means that the stability of the method is poor. When the number of negative results is large, the adaptability is limited. Both negative and unstable results are treated as potentially invalid solutions.

As shown in Table 3, PMOC outperforms the other two solving methods even though it provides potentially invalid solutions in 37.2% of the experiments. FP-TFA performs the worst, which derives potentially invalid solutions in 80.3% of the experiments. Service partitioning combined with PMOC only produces 5.7% potentially invalid results, significantly less than applying solving methods directly.

The greater presence of valid results when using service partitioning with solving methods has two causes: i) there is no unstable or negative solution when all cyclic dependen-

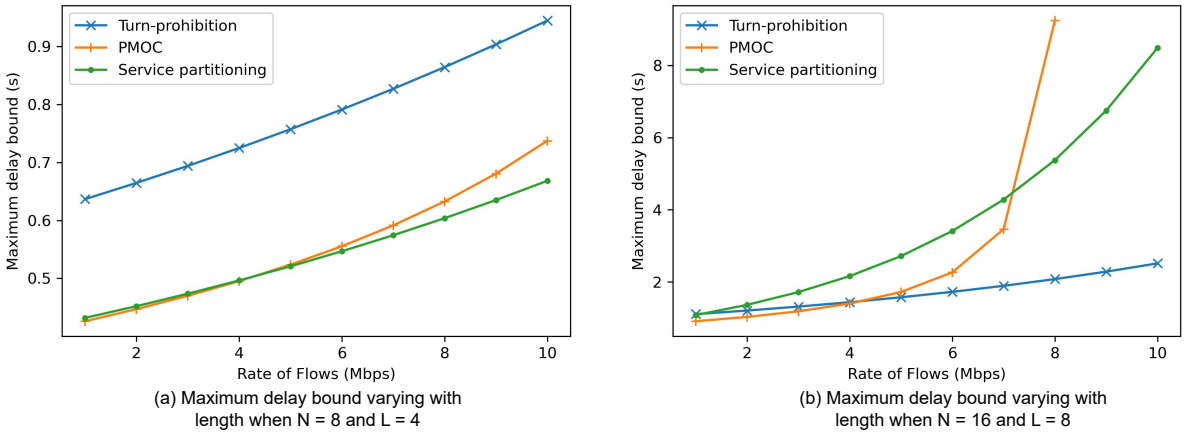

Figure 8: The largest worst-case delay bounds of three methods with different flow rates

Table 3: Evaluation of stability and adaptability of the four methods in 880 ring scenarios

Method	FP-TFA	PMOC	Fix-point method	Service partitioning
Unstable results	267	257	107	50
Negative results	440	70	370	0
Valid results	173	553	403	830

cies can be broken; ii) service partitioning can reduce the cumulative arrival rate of flows forming the cycle by removing seven flows from the cycle. This is achieved by assigning seven flows to seven lower priority queues and assigning the remaining flows which are still in the cycle to the highest priority queue in SP. The service curve assigned to these remaining flows is unchanged because they receive the highest priority service (see Equation 16), while the sum of rate of the flows forming the cycle decreases because seven flows are removed from the cycle. Figure 9 shows the impact of the sum of arrival curve rates for flows forming the cycle on the number of potentially invalid solutions produced by the three solving methods. This experiment keeps N as 16 and node bandwidth as 100 Mbps. The y-axis represents the number of potentially invalid solutions from 100 experiment instances. All three methods generate more potentially invalid solutions when the sum of rates increases. Service partitioning can reduce the sum of rates of flows forming the cycle, so it can help reduce the number of potentially invalid solutions.

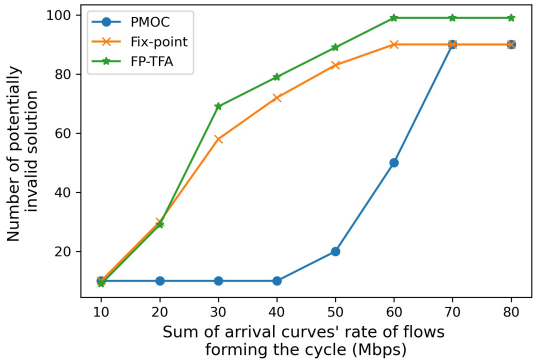

Figure 9: The influence of the sum of arrival curves' rate on the number of invalid solutions

6. CONCLUSIONS

In this paper, we propose service partitioning to enable the application of NC to non-feedforward networks. It is a breaking method, which can be combined with existing solving methods. Service partitioning has a limitation. Due to the limited number of buffers in each output port, service partitioning may not be able to fully break cycles in some networks. This will be addressed in our future work.

However, service partitioning does have a lot of advantages. Compared with PMOC and the fix-point method, it can provide competitive delay bounds. Moreover, it can bound the networks with proportional service curve allocation, which could not be solved using all solving methods. Compared with turn prohibition, service partitioning does not require rerouting or restricting the usage of links. Compared with FP-TFA, service partitioning can be applied to networks with arbitrary multiplexing. Service partitioning will not produce invalid solutions when it can break all cyclic dependencies, meaning that it has better adaptability and stability than solving methods. Although service partitioning may not be able to break all cyclic dependencies in some scenarios, stacking solving methods on top of service partitioning has better adaptability and stability than solving methods alone, reducing the occurrence of potentially invalid solutions. Moreover, service partitioning has better flexibility than existing methods by allocating network resources in the breaking process.

Acknowledgment

This work is supported by NSF Award No. 2146968. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of the sponsors of the research.

7. REFERENCES

[1] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[2] Boyang Zhou, Isaac Howenstine, Siraphob Limprapaipong, and Liang Cheng. A survey on network calculus tools for network infrastructure in real-time systems. *IEEE Access*, 8:223588–223605, 2020.

[3] Alexander Scheffler and Steffen Bondorf. Network calculus for bounding delays in feedforward networks of FIFO queueing systems. In *Proc. of QEST*, 2021.

[4] Raffaele Zippo and Giovanni Stea. Nancy: an efficient parallel network calculus library. *SoftwareX*, 19, 2022.

[5] Boyang Zhou and Liang Cheng. A reality-conforming approach for QoS performance analysis of AFDX in cyber-physical avionics systems. In *Proc. of IEEE/ACM IWQOS*, 2021.

[6] Sven Kerschbaum, Kai-Steffen Hielscher, Ulrich Klehmet, and Reinhard German. Network calculus: Application to an industrial automation network. 2012.

[7] Steffen Bondorf and Jens Schmitt. The DiscoDNC v2: A comprehensive tool for deterministic network calculus. In *Proc. of EAI ValueTools*, 2014.

[8] David Starobinski, Mark Karpovsky, and Lev A. Zakrevski. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Netw.*, 11(3):411–421, June 2003.

[9] Anaïs Finzi and Silviu S. Craciunas. Breaking vs. solving: Analysis and routing of real-time networks with cyclic dependencies using network calculus. In *Proc. of RTNS*, 2019.

[10] Ludovic Thomas, Jean-Yves Le Boudec, and Ahlem Mifdaoui. On cyclic dependencies and regulators in time-sensitive networks. In *Proc. of IEEE RTSS*, 2019.

[11] Ahmed Amari and Ahlem Mifdaoui. Worst-case timing analysis of ring networks with cyclic dependencies using network calculus. In *Proc. of IEEE RTCSA*, 2017.

[12] Kashif Mahmood, Amr Rizk, and Yuming Jiang. On the flow-level delay of a spatial multiplexing MIMO wireless channel. In *Proc. of IEEE ICC*, 2011.

[13] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[14] Anne Bouillard. Trade-off between accuracy and tractability of network calculus in FIFO networks. *Elsevier Performance Evaluation*, 153, 2022.

[15] Rene L Cruz. A calculus for network delay. ii. network analysis. *IEEE Trans. Inf. Theory*, 37(1):132–141, 1991.

[16] Ahlem Mifdaoui and Thierry Leydier. Beyond the accuracy-complexity tradeoffs of compositional analyses using network calculus for complex networks. In *Proc. of the CRTS workshop*, 2017.

[17] Stéphan Plassart and Jean-Yves Le Boudec. Equivalent versions of total flow analysis. *arXiv preprint arXiv:2111.01827*, 2021.

[18] Xiaoyu Liu, Chi Xu, and Haibin Yu. Network calculus-based modeling of time sensitive networking shapers for industrial automation networks. In *Proc. of IEEE WCSP*, 2019.

[19] Luxi Zhao, Paul Pop, and Silviu S Craciunas. Worst-case latency analysis for IEEE 802.1 Qbv time sensitive networks using network calculus. *IEEE Access*, 6:41803–41815, 2018.

[20] Seyed Mohammadhossein Tabatabaee, Jean-Yves Le Boudec, and Marc Boyer. Interleaved weighted round-robin: A network calculus analysis. *IEICE Transactions on Communications*, 2021.

[21] Google. OR-tools official website, July 2022. `https://developers.google.com/optimization`.