AN ABSTRACT OF THE DISSERTATION OF

Rafid Almahdi for the degree of Doctor of Philosophy in Computer Science presented on March 16, 2021.

Title: Meeting End-to-End Deadlines in Real-Time Networks Using Software Defined Networking (SDN)

Abstract approved: _____

Rakesh Bobba

Abstract

Network flows in Real-Time (RT) systems need to meet stringent end-to-end deadlines in order for such systems to operate safely and reliably. Today, such systems use custom or domain specific network system designs to meet end-to-end deadlines and other constraints of real-time flows. In this work we explore the design of real-time networks using common-off-the-shelf (COTS) components by leveraging Software-Defined Networking (SDN) paradigm. In particular, we explore the effectiveness of using i) spatially varying but locally static flow priorities and ii) the impact of using Least-Slack Prioritization on the performance of network path layout and provisioning algorithms. Specifically, we propose different heuristics for spatial variation of static flow priorities in a real-time network and empirically show that spatial variation of priorities can accommodate more real-time flows than sim-

ple static priorities. Further, we show that least-slack based flow priority assignment performs better than deadline monotonic priority assignment for multiple path layout algorithms considered in this work.

Meeting End-to-End Deadlines in Real-Time Networks Using Software Defined Networking (SDN)

by

Rafid Almahdi

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented March 16, 2021
Commencement June 2021

Doctor of Philosophy dissertation of Rafid Almahdi presented on March 16, 2021.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

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

Rafid Almahdi, Author

ACKNOWLEDGEMENTS

Thanks very much to Prof. Rakesh Bobba for his feedback, assistance with creating and editing this work, and guidance during this project. Special thanks to Jed Irvine for the mentoring and support. Thanks also to Prof. Sibin Mohan, Rakesh Kumar, Monowar Hasan, and Konstantin Evchenko from the SDN research group at the University of Illinois for discussing and providing feedback on the main ideas of this work. Lastly, thanks to Brandon Ellis for programming and writing assistance.

I express my appreciation to the members of my committee Prof. Bella Bose, Prof. Bechir Hamdaoui and Prof. Amir Nayyeri for offering advice and reviews for my research works. I thank my Graduate Council Representative (GCR) Prof. Kyle Niemeyer.

I express my appreciation to the Ministry of Higher Education and Scientific Research (MOHESR), University of Thi-Qar, IRAQ, for the financial support.

Last but most important, I thank my family, my wife (Wasan Fayyadh), my children (Shahad, Noor, and Ali) for their love and encouragement.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1: Introduction

Network flows in Real-Time (RT) systems need to meet stringent end-to-end deadlines for safe and reliable operation of such systems [11,60]. If not carefully designed and provisioned, contention between flows in real-time (RT) networks can result in delays that cause network flows to not meet deadline constraints. If critical messages experience delays or delivered incorrectly, [19] then the system could fail to operate. For example, if the front bumper of a car has a sensor that serves the purpose of detecting an impact, the information obtained is relayed to the electronic control unit (ECU) within 20 ms followed by an initiation of airbag deployment within a few milliseconds [31]. Any delays that can occur in the processes could result in serious injuries to the driver and/or the passengers. In addition to timing constraints, some (critical) flows have distinct priorities and isolation requirements [18, 25, 45].

There are current designs that address this, though they have two main problems. Solutions specific to their environment, such as the avionics full-duplex switch Ethernet(AFDX) [5,15,35] and the controller area network (CAN) [26,41–43,58,62], can be too conservative or end up over-engineered. Additionally, due to the limitations of physical implementations, there is a large overhead incurred in order to manage the solution. In situations similar to this, a dynamic configuration is often necessary in routing packets based on flow delays and switch workloads that are successful in meeting all the high priority (QoS) requirements

We believe that if the system had 'global' visibility built into the network and could perform combined resource allocation for the RT and non-RT flows, then the design of RT networks could be significantly improved [10]. Software-defined networks (SDNs) have the potential to fulfill these requirements [38].

Kumar et al. [34] created one of the initial "real-time aware" SDNs, designed for use with applications that have hard-timing constraints. In their approach, they dedicate a whole queue to a real-time flow, which results in the network resources being under-utilized. Other recent works on this topic includes the papers of Qian et al. [48] and Lee et al. [36]. There are, however, two key issues with these implementations. First, they suffer from being overly conservative, which results in under-utilization of the available resources in a network [34, 36]. Second, they are unable to be implemented using commodity off-the-shelf (COTS) components [36, 48], as a result of needing capabilities that are not available in COTS switches. An example of this is Lee et al. [36], that adds tolerance for link-failures to the Kumar et al. [34] approach. This approach is done by altering the data plane of a given switch, but will still encounter problems when dealing with link failures, possibly leading to dropped packets or the delay increasing. Our proposed work attempts further improvements to the work of Kumar et al. and Kashinath et al. [31, 34]

1.1 Proposed Research

1.1.1 Spatially-Varying Static Priority Scheduling

Kashinath et al. presented an algorithm that statically allocates paths, pre-computes

backup paths,and multiplexes within the same queue. It maintains compatibility with COTS switch schemes and has the goal of guaranteeing end-to-end deadlines within a real-time system. [31] We investigated whether adding a followup heuristic that surgically adjusts priorities for particular flows at particular switches could improve end-to-end deadline guarantees without losing COTS compatibility.

1.1.2 Least-Slack Prioritization

Within static priority schemes, only deadline monotonic approaches have been studied. We propose a novel approach using Least-Slack Prioritization where priority assignment factors in path hop counts to more effectively utilize network resources.

Chapter 2: Background

2.1 Quality of Service

Quality of Service is extremely important for certain applications, and different kinds of applications behave differently. For example, a video application might be tolerant of slight delay and require a large amount of bandwidth, while a voice application may be tolerant of loss but extremely intolerant of delay due to the real-time nature of the experience.

The Quality of Service for a flow in a network may be analyzed with respect to four performance parameters: reliability, delay, jitter, and bandwidth. Different applications have a wide variety of Quality of Service (QoS) requirements. Networks that attempt to provide guarantees can be roughly categorized into three groups [61].

The first group, Best Effort networks, do their best to deliver high quality of service but can provide no guarantees or assurances. This is a very basic approach that is often insufficient and doesn't provide the granularity required by modern applications that have complex (QoS) requirements which this group fails to meet.

The second group which we consider are Integrated Service networks [9]. These networks provide QoS guarantees as requested by individual flows and applications on a per-flow basis. Unfortunately, this is often complicated. For example, the network must maintain a significant amount of information about the state of each flow in order to continually monitor their performance. This makes such a network

inherently difficult to scale; however, they can be very effective on a small scale.

Lastly, Differentiated Service networks [9] provide QoS 'guarantees' on a per-class basis. Each class is associated with different QoS parameters and the network attempts to ensure that as many of these are met as possible. This means that there are no firm guarantees for each flow as to the QoS it will receive (a benefit bestowed by an Integrated Service network). However, this decrease in fine-tuned control allows for much better scalability with respect to Integrated Service networks.

In this work our goal is to provide stringent latency guarantees for flows in a closed network while also enabling best effort flows to share the same resources. The approach we consider falls somewhere between these last two categories,and closer to integrated service.

2.2 Software Defined Networking (SDN)

An expand SDN contains a control plane and a data plane, which serve different purposes. In SDN, a software program known as the controller provides guidance for an entire network through one of several APIs. The control plane is centralized and implemented on the controller, which drives network devices. Each individual device follows flow rules installed by the controller, thus allowing the control plane to configure the data plane with ease [14]. See Figure 2.1 that depicts this separation of concerns. The controller maintains full knowledge of the data plane and enables a control application at a high level to make determinations about flow routing. We propose using SDN to achieve the network wide view we require. Strict priority

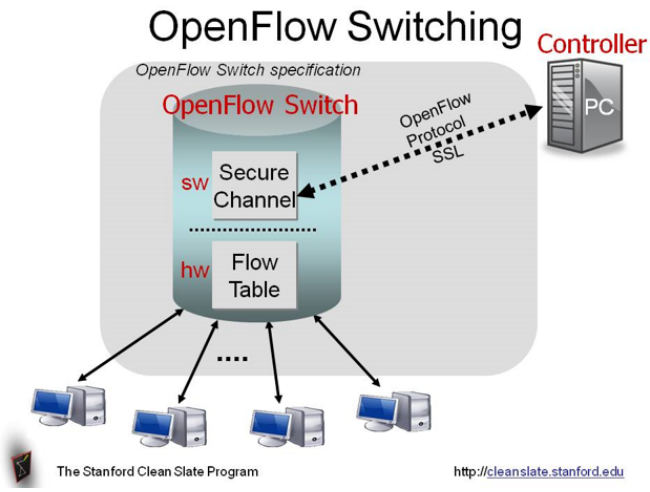

Figure 2.1: SDN explained through the OpenFlow Switch Model.

scheduling that is used in this work is widely available on COTS switches. SDNs also provide higher compatibility with a wider range of hardware manufacturers [43]. For these reasons, SDNs have received serious attention from the networking field. Today, SDNs have been adopted in enterprise networks [37], data-center networks [28], military networks [54], and even in industrial control networks [8, 46].

2.3 Scheduling Algorithms

Scheduling algorithms are needed in networks to ensure that packets are able to be delivered on time. If a packet is unable to be delivered on time, the signal to deploy the airbag in the event of a collision for example, the results could be disastrous. In many real life cases, ensuring that traffic in a network can be scheduled is vital to ensuring safety to the users. Scheduling algorithms come in many flavors and a wide variety of factors influence which choice may be optimal in any given

system. The two most important distinctions in scheduling algorithms are whether they are preemptive or non-preemptive, and whether they are static or dynamic. In preemptive systems, a new task (or packet) arriving at a processor (switch) may interrupt the execution transmission of a previous task, if appropriate, and insert itself immediately into the execution transmission order. In non-preemptive systems, this is not allowed [13]. Instead, new tasks or packets must wait at least until the current execution transmission is finished before they may be considered for execution transmission [51].

In dynamic systems, the priority of a given task may change depending on current conditions. For example, a packet that has a great deal of remaining deadline budget might be prioritized lower than a packet which must be delivered quickly, even if they are both part of hard real-time flows. In static systems, all factors that are included in consideration for prioritization are preordained. For example, in a static priority network system each flow of packets will have a preset priority that does not change based on the actual conditions in the system when those packets arrive at a switch [51].

This work explores static non-preemptive scheduling to provide end-to-end deadline guarantees.

Chapter 3: Related Work

Much work has been done to solve the problem of meeting end-to-end deadlines in real-time network. Some relevant approaches as applied to SDN are discussed here.

3.1 End-to-End Network Delay Guarantees for Real-Time Systems using SDNs

Kumar et al. [34] proposed a framework with a heuristic algorithm that is used for two purposes. First, to solve path layout optimization problem with deadline and bandwidth constraints and second, to isolates flows in the system into discrete queues in order to better guarantee flows will meet their end-to-end deadlines. Fundamental to their approach was the setting of packet servicing priorities on a flow-by-flow (static) basis. This more delay-aware SDN approach did make improvements in the meeting of end-to-end deadlines. While their implementation treats the priority as a fixed value, our work attempts to change the priorities on a switch-by-switch basis and also explores different priority assignment schemes.

3.2 Safety Critical Networks using Commodity SDNs.

Kashinath et al. [31] presented RealFlow, a framework intended to ensure a practical application of commodity SDNs in current safety-critical environments. To achieve

this, RealFlow uses a novel static path allocation algorithm as well as pre-computed backup paths for flows of a critical nature. Additionally, flows are multiplexed within the same queue. RealFlow implements all of this as a path allocation algorithm, while still maintaining compatibility with current COTS switch schemes and still guaranteeing end-to-end deadlines within a real-time system.

This differs to the work by Kumar et al. by utilizing flow multiplexing. That is, flows with common priorities share the same queue. In our work, we build and extend it in two directions. (i) We study flow priorities that are different from deadline-monotonic. (ii) We study flow priority that we vary from switch to switch.

3.3 A linux real-time packet scheduler for reliable static SDN routing.

Qian et al. [48] introduced an algorithm that aids in routing and determining the forwarding paths for real-time message moves in distributed calculating domains while also creating a scheduler that helps in enforcing a message forwarding policy on network devices. In addition to that, this routing algorithm takes into consideration network resource demands of real-time messages and deadlines. Due to this, no results of real-time messages are allowed to be dropped due to the network contention that performs when being controlled by the message scheduler while also no real-time messages unable to meet their deadlines. However, this scheme requires modifications of switch scheduling mechanisms and hence cannot be realized with COTS hardware. Comparatively, our proposal does not require a hardware modification and can work with COTS SDN switches.

3.4 A priority based packet scheduler with deadline considerations.

Tamer [18] provided a dynamic priority scheduler which used cost functions to help provide better QoS . In this scheme, when a packet is about to be served its 'cost' is calculated. Then, if the cost is above a threshold, that packet is served. Otherwise, the packet is pushed back in the queue and the next set of packets are sorted, then examined based on the same principle. This was shown to be fairly effective at minimizing delay, but was also extremely expensive and resulted in higher loss rates due to deadline violations. The expensive run-time resorting calculations made are something our proposal avoids, done so by pre-computing priorities for switches.

3.5 Hybrid EDF Packet Scheduling for Real-Time Distributed Systems.

Qian and Mueller [47] showed that an 'Earliest Deadline First' approach could be used to address problems with communication delay variances . They used an EDF scheduler to forward packets in an experiment, along with periodic message transmission, to minimize the affect of application-level interrupt being handled on system predictability. In their example, the EDF packet scheduler can also minimize their traffic control mechanism (periodic transmission) decreased variability and the deadline miss rate [49]. However, their implementation has the chance to insufficiently compute the risk of a packet missing its deadline. In our proposal, a different cost analysis calculation is made, resulting in fewer packets missing their deadline.

3.6 Coarse-grained Scheduling with SDN Switches

Rifai et al. [50] proposed two schedulers designed to minimize time to flow completion in an SDN, for the majority of flows. Based on the premise that the majority of flows are very brief with comparison to a few, larger flows, they offered schedulers which gave priority for transmission to flows which were shorter. The first (dubbed the 'stateful scheduler') was implemented by giving newer flows a higher priority than older flows; thus, the longer a flow existed the lower priority it obtained. Priorities were recalculated at a set interval, in order to reduce overhead. Their second scheduler, the 'scalable scheduler', operated on the same premise with the caveat that flows were only monitored if they exceeded 10% of the switch's bandwidth (thus meriting additional examination). Their recalculation of priorities on a set interval produces an overhead. An overhead that our implementation is able to avoid, by only calculating priorities when a scheduling issue occurs.

3.7 Can QoS be dynamically manipulated using end-device initialization?

Sardis et al. offered an interesting alternative to the usual flow initialization procedure, wherein the end host initialized new flows with a source host and specified QoS requirements [52]. They found that delay in establishing communication depended upon client processing power (some end hosts are very small embedded systems) and the amount of metadata required to establish a connection. Their solution of allowing a host to initialize a flow and modify its needs, results in more control over

the network environment at the cost of overhead. Our approach does not attempt to achieve this level of overhead and network control, instead choosing to use the already allocated resources to find a solution.

3.8 Universal Packet Scheduling

Radhika et al. [40] begun by using more of a theoretical perspective and started to analyze the existence of a single universal packet that would be considered as a scheduling algorithm that is able to successfully replay all viable schedules. They prove that even an algorithm such as that can not exist, the classical Least Slack Time First (LSTF) is similar to being one (in terms of the number of congestion points it is able to control). They also displayed the capability of LSTF to approximately replay a broad range of scheduling algorithms while undergoing varying network settings. While it is a theoretical interest to replay a given schedule, it requires the knowledge of viable output times that is not available in practice. In this proposal, it is shown that a universal packing algorithm does not exist, but an LSTF is the closest implementable algorithm. LSTF is used in our work for this reason.

3.9 Meeting End-to-End Deadlines through Distributed Local Deadline Assignments

Hong et al. introduces a distributed approach that works on combining local-deadline assignment with feasibility analysis making the resulting deadline assignment guar-

anteed to be schedulable [29, 30]. Their approach works on formulating the local-deadline assignment problem making it a mathematical programming problem that includes greatly increasing the minimum slack time among all the existing jobs that are executed on each processor [7]. To account for job interferences on each processor, it is important that they introduce a novel transformation of the sufficient and necessary condition. Their proposed mathematical programming formulation addresses the shortcomings of existing work in two ways. First, by using a shared processor to consider resource competition between all jobs. Second, by using different stages along the job's execution path to work on organizing the local deadline distribution. However, processors that do not have an execution order do not work for this approach because it can not perform in those situations [24]. While their proposal focuses on altering the local deadline for processors, ours aims to change the priorities of network flows.

3.10 Cost-based scheduling and dropping algorithms to support integrated services

Peha and Tobagi at Carnegie Mellon proposed a number of cost-based scheduling algorithms. These allowed for an arbitrary cost function to be used in order to determine the priority of each packet [45]. Unfortunately, this cost calculation is expensive as it requires O(n) operations (where n is the number of packets in the queue) each time a packet is sent. This is due to each remaining packet's priority must be recalculated at the new time step. Furthermore, in a traditional network,

while it is possible to design cost algorithms which are sensitive to the time remaining until a packet's deadline or the time that a packet has thus far spent in queue, it is not feasible to calculate cost based on the remaining *distance* that a packet has to travel. This is because traditional switches do not have access to that information. The implementations described by Peha and Tobagi do not use SDN and have the requirement of resorting packets. Comparatively, our proposal provides control over the packet priority without recomputing the priority.

3.11 Achieving end-to-end real-time quality of service with SDN.

Guck and Kellerer propose deterministic end-to-end real-time QoS model for the realization of a centrally planned real-time communication service [21]. This service implements a joint routing and access control system, which is based on SDN leveraging the centralized view that its controller has on the network. Their model allows an admission control algorithm to be implemented on a SDN controller. It is the objective of this work to build a QoS concept for real-time applications by exploiting the benefits provided by SDN. The centralized view of SDN allows for deterministic end-to-end QoS planning. The main part of this work introduces a network calculus based end-to-end traffic model, which supports a centralized QoS resource allocation planning. Their concept can be used for access control for real-time applications [21].

3.12 Dynamic priority-adjustment for real-time flows in SDN.

An et al. [6] attempts to guarantee the different delay deadlines of real-time flows in SDN-based networks, they proposed two priority-adjustment algorithms. The first adjusts the per-flow priority and the second adjusts per-router priority. Priorities are given to each flow in the per-flow priority-adjustment algorithm that also works on changing the priority of the flow containing the highest normalized delay to the highest priority. Like a bubble sort, it then updates the priorities of all the other flows from the highest to the lowest priority level. In addition to that, each switch in the per-router priority-adjustment algorithm is given different priorities for a flow. But this is a greedy algorithm for the priority adjustment [6]. While their proposed algorithm takes a greedy approach to the highest normalized delay, one which does not consider the relationship between a flow's delay and deadline, our solution appropriately addresses the correlation between the delay cost and deadline requirement.

3.13 Quality of service guaranteed dynamic resource management in SDN.

Xu et al. [59] design a novel QoS-enabled management framework to create an end-to-end communication service over software defined networking (SDN). This framework classifies flow into different level and allocates network resource dynamically to ensure the request of services. The high priority flow will be rerouted by route optimization algorithm when the original path can't provide available bandwidth. Once there is no

feasible path for high priority flow, this framework will enable queue mechanism to guarantee the transmission of QoS flow [59]. The QoS requirements of this proposal allow for a high priority flow to simply have more resources given to it as a solution to ensure scheduling. Comparatively, our proposal attempts to find a scheduling solution without creating and allocating additional resources.

Chapter 4: System Model

We now describe our system model that closely follows previous literature [31, 34].

4.1 Network and Flow Specifications

Consider a network with a specific set of real-time flows $\mathcal{F}$ (that are generated from real-time applications), OpenFlow switches, and a controller that have pre-specified end-to-end delays and bandwidth guarantee requirements. Each flow $F_i \in \mathcal{F}$ is characterized by the tuple $F_i = (src_i, dst_i, T_i, D_i^{max}, pkt_i, pri_i)$ in which src_i and dst_i are considered the source and destination host, D_i^{max} is the end-to-end delay bound (deadline), $T_i \in \mathbb{Z}^+$ is the inter-arrival (period), pkt_i is the size of the packet in bits ($e.g.$, 1kB packets = 8kbits) and pri_i accounted as the priority resulting in $B_i = \frac{1}{T_i} \times packet\_size_i$ being the bandwidth requirement of F_i. The network is modeled as an undirected graph $N(\mathcal{V}, \mathcal{E})$. $\mathcal{E}$ is a set of edges, with each edge representing a possible path from one switch (π_s) to another ($\pi_{s'}$) and $\mathcal{V}$ is a set of nodes, with each node representing a switch π_s.

Queues and Priorities: All switches $\pi_i \in \mathcal{V}$ have a set of $L+1$ priority queues. It is assumed that the priorities for a give flow are chosen from $\mathcal{L} = \{0, 1, \cdots, L-1\}$, a set of predefined distinct priority-levels in which $pri_i \in \mathcal{L}$ and level-0 are considered the highest priorities. This is where flows with a priority-level of $l \in \mathcal{L}$ are given to $l + 1$ -th queue for each switch $\pi_s \in \mathcal{V}$. The priority levels are derived from the

Table 4.1: Key Mathematical Notation

Notation	Interpretation
Topology	
$\mathcal{F}$	Set of RT flows in the network
$\mathcal{F}'$	Set of critical RT flows for which we assign backup paths ($\mathcal{F}' \subseteq \mathcal{F}$)
$\tilde{F}'_s$	Set of RT flows with priority-level l routed through the switch π_s $\left(\tilde{\mathcal{F}}'_s \subseteq \mathcal{F}\right)$
π_i	OpenFlow switch in the network ($\pi_i \in \mathcal{V}$)
QoS	
$D^{tp}_{s',s}$	Transmission and propagation delay incurred on the edge $(\pi_{s'}, \pi_s)$
q^s_i	Queuing and processing delay (FIFO) of flow F_i at switch π_s
I^s_i	Queuing and processing delay (Interference) of flow F_i at switch π_s
β^s_i	Queuing and processing delay (Blocking) of flow F_i at switch π_s
Q^s_i	Queuing and processing delay (Total) of flow F_i at switch π_s
$\mathcal{D}_i\,(\mathcal{P}_i)$	The total delay for F_i over a path $\mathcal{P}_i$
$\mathcal{B}_i\,(\mathcal{P}_i)$	The total bandwidth utilization for F_i over a path ($\mathcal{P}_i$)

applications that would naturally generate or use this data. Which levels are used is decided by the designers of the application or system in question. Finally, in our work, we assumed that the application, and their correspondingly generated network flows, have statically assigned priorities. This assumption is based on statically assigned priorities being the most common type of real-time system in use currently.

Note: It's important to consider that more than one flow at a given time can share the same priority-level (e.g., $\text{pri}\,i_i = pri_j, \exists F_i \neq F_j$). In addition to that, flows with identical priority-level l can be multiplexed to the $l+1$ -th queue successfully.

The deadline and bandwidth-aware path selection is expressed as a multi-constrained path (MCP). The key mathematical notations used are listed above in Table 4.1.

4.2 Delay and Bandwidth Calculations

Now we compute the delay and bandwidth values for each real-time flow $F_i \in \mathcal{F}$, for its path $\mathcal{P}_i$ from switch sra_i to switch dst_i $\{\pi_s\}$.

4.2.1 Delay Calculations

In each flow's route, it will experience the following delays along the network path:

1. Transmission and Propagation Delay: A delay based on the data rate of a link and signal's transmission speed. Modeled as a constant upper-bound, based on the given link capacity, material, and length.

2. Queuing and Processing Delays: Occurs at each switch in a path, consisting of:

   (a) FIFO Queuing Delay: Caused by flows at the same priority-level being routed through the same switch in a path $\mathcal{P}_k$.

   (b) Interference Delay: Caused by higher-priority flows sharing the same switch in a path $\mathcal{P}_k$.

   (c) Blocking Delay: A delay caused by the non-preemptives of packet transmission.

4.2.2 Computed Delays

- Transmission and Propagation Delay = Propagation delay + Transmission delay
- Queuing and Processing Delay = Interference delay + Blocking delay + FIFO delay

4.2.3 Transmission and Propagation Delay

All flows, $F_i \in \mathcal{F}$, are affected by transmission delay, computed as $D^t_{s',s} = \frac{pkt_1}{\text{data rate}}$. Additionally, every flow is also affected by signal propagation delay as a result of the medium. This is computed as $D^p_{s',s} = \frac{\text{Length of the link}}{\text{velocity of signal in medium}}$. The total packet transmission and propagation delay that is incurred on edge $(\pi_{s'}, \pi_s) \in \mathcal{E}$ (i.e., the link between $\pi_{s'}$ and π_s), is denoted as $D^{tp}_{s',s}$. This can be computed as:

$D^{tp}_{s',s} = D^t_{s',s} + D^p_{s',s'}$

4.2.4 Processing and Queuing Delay from Flows at the Same Priority

Flows sharing a priority level l, are put in the same $l+1$ -th queue (assuming bandwidth requirements are met) for each switch π_s and processed in a first in, first out (FIFO) manner. Multiplexing these same priority flows into one queue handles the case of there being more flows than queues. This is compared to using a one-flow-per-queue scheme which only accepts a fixed number of flows. The queuing delay is computed for each packet in flow F_i because of the interference from other flows with

the same priority level routing through the same switch ($i.e., \forall F_j \neq F_i \mid pri_j = pri_i$). Denoting $\tilde{\mathcal{F}}_s^l \subset \mathcal{F}$ as a set of flows with a priority level l with a path through switch π_s(i.e., each flow $F_i \in \tilde{\mathcal{F}}_s^l$ shares the same queue ψ_l in π_s).

For give flow $F_i \in \tilde{\mathcal{F}}_s^l$, the worst case queuing delay can be computed as : $q_i^s = \sum_{F_k \in \mathcal{F}_0!} \left\lceil \frac{T_1}{T_k} \right\rceil \hat{d}_k$ where $\hat{d}_k$ is the per-packet processing delay. This worst case delay occurs when $F_k \neq F_i$ are scheduled before F_i, assuming that switch π_s arbitrarily scheduled one of these flows when packets of multiple flows with conflicting priority levels all arrived together). For example, consider the two flows F_1 ($T_1 = 10$) and F_2 ($T_2 = 5$). If F_2 arrives at a quicker rate, then in the worst-case F_1 will experience a FIFO queuing delay from $\left\lceil \frac{T_1}{T_2} \right\rceil = 2$ packets of F_2 within one period of F_1.

4.2.5 Processing and Queuing Delay from Blocking and Interference Delay

Switches process packets in order. The order is based on arrival time and priority, where lower priority and late arriving packets will be processed later. Therefore, for a flow F_i, any interference from other flows routed through a common switch needs to be addressed.

Flows with a higher priority will interfere with lower priorities flows, as the switch scheduler will dequeue the higher priority flow first. Let $hp\left(\overrightarrow{\mathcal{F}}_s^l\right)$ refer to a set of flows with a higher priority than l, that are routed together through switch π_s. The interference delay, caused by higher priority flows, can have its upper bound estimated using a response-time analysis. This calculation is shown below:

$I_i^s = \sum_{F_h \in h_p(\bar{F}!)} \left\lceil \frac{T_i}{T_n} \right\rceil \hat{d}_h$ where I_i^s is the interference experienced by each flow $F_i \in \tilde{\mathcal{F}}_s^l$

A switch processes flows in a non-preemptive manner, meaning in the worst case, a given flow may experience at most one packet delay from lower-priority flows. This blocking delay can be computed as follows:

$\beta_i^s = \max_{F_k \in \mathcal{F}_b, k \neq i} \hat{d}_k$

Therefore, the total queuing and processing delay (presented in for F_i at the switch π_s, Q_i^s) can be expressed as follows:

$$Q_i^s = q_i^s + I_i^s + \beta_i^s = \text{ FIFO delay } + \text{ Interference delay } + \text{ Blocking delay} \quad (4.1)$$

The highest-priority flows do not suffer interference delays (*i.e.*, $I_i^s = 0$ if $pri_i = l = 0$). However, they can experience FIFO delays from other flows, if those flows are multiplexed into same queue (the highest-priority one). Flows that are naturally sensitive to latency (*e.g.*, engine control messages in avionics systems, airbag and ABS deployment messages in cars, etc.) should be given the highest priority (see Eqs. 4.1).

Total Delay: The total delay for flow F_i over path $\mathcal{P}_i$ is calculated by taking into account transmission and propagation delay at every edge and both the queuing and processing delays at every switch in $\mathcal{P}_i$, as show in Eqs. 4.2:

$$\mathfrak{D}_i\left(\mathcal{P}_i\right) = \sum_{(\pi_{s'}, \pi_s) \in \mathcal{P}_i} D_{s',s}^{tp} + \sum_{\pi_s \in \mathcal{P}_i} Q_i^s \quad (4.2)$$

4.2.6 Bandwidth Calculations

All flows consume resources quantified by its bandwidth use. This can be defined as the ratio of the bandwidth requirement to the bandwidth which is still available on the link. Therefore, the bandwidth utilization for a link $(\pi_{s'}, \pi_s)$ by a flow F_i is defined as:

$\mathfrak{B}_i(\pi_{s'}, \pi_s) = \frac{B_1}{B^R(\pi_{s'}, \pi_s)}$ where B_i is the bandwidth requirement of F_i and $B^R(\pi_{s'}, \pi_s)$ is residual (viz., available) bandwidth of an edge $(\pi_{s'}, \pi_s) \in \mathcal{E}$.

It's important to consider that a flow can be assigned to a given path only if the flow's bandwidth utilization is less than one on every edge in the flow's path.

Total Bandwidth: The total bandwidth utilization for F_i through a path $(\mathcal{P}_i)$ is computed by considering the bandwidth utilization for every edge in $\mathcal{P}_i$, as shown in Eqs. 4.3:

$$\mathfrak{B}_i(\mathcal{P}_i) = \sum_{(\pi_{s'}, \pi_s) \in \mathcal{P}_i} \mathfrak{B}_i(\pi_{s'}, \pi_s) \tag{4.3}$$

4.3 QoS-Aware Path Selection [31].

First, we introduce the QoS constraints and metrics that must be adhered to. Second, we describe the path selection scheme propsed in [31] to compute a route $\mathcal{P}_i$ that goes from a given source host s_i to a destination host t_i for all flows RT flow $F_i \in \mathcal{F}$, with respect to the QoS constraints. The expressions for end-to-end delay and bandwidth constraints of a flow F_i are derived as follows.

4.3.1 Delay Constraint

$D_i^{\max}$ is derived as the deadline constraint. Using the end-to-end delay for flow F_i over a path P_i, defined as $\mathfrak{D}_i(\mathcal{P}_i)$, the deadline constraint is computed using Eq. 4.2. Thus, in order for flow F_i's delay requirements to be met, the end-to-end delay constraint is derived as show in Eq. 4.4:

$$\mathfrak{D}_i(\mathcal{P}_i) \leq D_i^{\max} \tag{4.4}$$

4.3.2 Bandwidth Constraint

The bandwidth calculation used is the same one utilized in Kumar et al. [34]. The bandwidth bound of a flow F_i over a path $\mathcal{P}_i$ is computed as: $\mathrm{B}_i(\mathcal{P}_i) \leq \max_{(\pi_{s'},\pi_s)\in\mathcal{E}} \mathfrak{B}_i(\pi_{s'},\pi_s)|\mathcal{V}|$ where $|\mathcal{V}|$ represents the cardinality for a switch set in the topology N.

Therefore, in order to satisfy the bandwidth requirement of B_i for the flow F_i,:

$$\mathfrak{B}_i(\mathcal{P}_i) \leq \widehat{B}_i, \quad \widehat{B}_i = \max_{(\pi_{s'},\pi_s)\in\mathcal{E}} \mathfrak{B}_i(\pi_{s'},\pi_s)|\mathcal{V}| \tag{4.5}$$

A path $\mathcal{P}_i$ is viable if the QoS guarantees for flow F_i are able to be kept, i.e. it satisfies (see Eqs. 4.4 and 4.5). The formal definition for the path layout problem is as follows:

Given the SDN topology $N(\mathcal{V},\mathcal{E})$ and the set of flows $\mathcal{F}$ with paths over N, the goal is to find a set of $|\mathcal{F}|$ feasible paths $\mathcal{P} = \left\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_{|\mathcal{F}|}\right\}$ where $\forall F_i : \mathcal{P}_i =$

$\pi_1\pi_2\cdots\pi_k$ with $\pi_1 = s_i, \pi_k = t_i$ and $(\pi_j, \pi_{j+1}) \in \mathcal{E}$ for all $1 \leq j \leq k-1$.

The problem of finding a valid path layout, with respect to the deadline and bandwidth-constraints, is an the NP-Complete, multi-constrained path (MCP) problem [16, 27]. Kashinath et al. [31] use a low-complexity heuristic solution in order to compute the path of a flow [17, 22].

4.4 Interference Index [31].

For a set of paths $\hat{\rho}$, there is a path $\mathcal{P}_i \in \widehat{\rho}$ for flow F_i. Have $\widehat{\mathcal{F}}_s^l$ be a set of flows of priority-level l where, for all paths in the set $\hat{\rho}$, their flow passes through the switch π_s.

Formally, $\widehat{\mathcal{F}}_s^l = \bigcup \{F_j \mid pri_j = l : \forall \mathcal{P}_j \in \widehat{\rho} \wedge \pi_s \in \mathcal{P}_j\}$ and let $\hat{\mathcal{F}}_s = \bigcup_{l \in \mathcal{L}} \left\{\widehat{\mathcal{F}}_s^l\right\}$.

To phrase another way, $\widehat{\mathcal{F}}_s$ represents a set of flows from $\mathcal{P}_j$ that all intersect at the given switch π_s, such that $\mathcal{P}_j \in \widehat{\rho} \wedge \pi_s \in \mathcal{P}_j$. See Algorithm 1.

Algorithm 1 Delay and Bandwidth-Aware Path Selection(DBAPS) [31].

Input: Network topology $N(\mathcal{V}, \mathcal{E})$ and a flow set $\mathcal{F}$ with QoS (delay and bandwidth) requirements
Output: The path assignment $\mathcal{P}^* = \bigcup_{F_i \in \mathcal{F}} \{\mathcal{P}_i^*\}$, if there exists a feasible path for all flows, Unschedulable otherwise.

Step 1 (Candidate Path Generation):
For each flow $F_i \in \mathcal{F}$, generate all candidate paths,$\rho_i = \bigcup \{\mathcal{P}_i\}$
$\hat{\rho} \leftarrow \bigcup_{F_i \in \mathcal{F}} \{\rho_i\}$ /* Set of all possible candidate paths for all flows */
For each path $\mathcal{P}_k \in \widehat{\rho}$: calculate $\mathrm{II}(\mathcal{P}_k)$
Step 2 (Low Interference Path Selection):
$\mathrm{v}(\mathcal{P}_k) \leftarrow$ false, $\forall \mathcal{P}_k \in \widehat{\rho}$/*$A$ boolean flag* /
/ * repeat the iteration if some flow has more that one candidate path * /
while $\exists F_i, |\rho_i| > 1$ do
  /* Find the path with highest II (e.g., maximum interference) */
  $\hat{k} = \underset{\mathcal{P}_k \in \widehat{\rho} \wedge v(\mathcal{P}_k) = \text{ false}}{\operatorname{argmax}} \mathrm{II}(\mathcal{P}_k)$
  $\mathcal{P}_{\max} \leftarrow \mathcal{P}_k$
  $F_\alpha \leftarrow$ the flow contains the path $\mathcal{P}_{\max}$
if $|\rho_\alpha| > 1$ then
/* Remove the path with maximum interference */
$\hat{\rho} \leftarrow \hat{\rho} \backslash \mathcal{P}_{\max}$
Update $\mathrm{II}(\mathcal{P}_j)$ for all paths $\mathcal{P}_j$ that intersect with $\mathcal{P}_{\max}$
else
/* This is the only available path for F_α */
$\mathcal{P}_a^* \leftarrow \mathcal{P}_{\max}$/*Assign the path to F_α */
$v(\mathcal{P}_k) \leftarrow$ true /* Update the flag (since the path is selected) */
end if
end while
/ " Unable to find a path that respects QoS constraints * /
if $\exists F_i \in \mathcal{F}$ such that $\mathfrak{D}_i(\mathcal{P}_i^*) > D_i^{\max}$ or $\mathfrak{B}_i(\mathcal{P}_i) > \widehat{B}_i$ then
return Unschedulable
end if
return $\mathcal{P}^* = \bigcup_{F_i \in \mathcal{F}} \{\mathcal{P}_i^*\}$ /* Return the path */

Given a set of paths $\hat{\rho}$ and flow F_i, the worst-case queuing interference, $\widehat{Q}_i^s$, can be estimated. This is done utilizing Eqs. 4.1 with modified values, specifically, changing $\tilde{\mathcal{F}}_s$ to $\widehat{\mathcal{F}}_s$ and $\tilde{\mathcal{F}}_s^l$ to $\hat{\mathcal{F}}_s^l$.

The worst-case residual bandwidth can be formally calculated as, $B^R(\tau_{s'}, \tau_s) = B(\tau_{s'}, \tau_s) - \sum_{\forall F_j \in \widehat{\mathcal{F}}_{(\pi_{s'}, \pi_s)}} B_j$. Where, $(\pi_{s'}, \pi_s)$ represents the link, $B(\tau_{s'}, \tau_s)$ is the link capacity, and $\hat{\mathcal{F}}_{(\pi_{s'}, \pi_s)} = \cup \{F_j : (\pi_{s'}, \pi_s) \in \mathcal{P}_j \wedge \mathcal{P}_j \in \widehat{\rho}\}$ is the set of all unique

flows that pass through the given link $(\pi_{s'}, \pi_s)$.

Note: If the routes of two flows travel over the same switch(es) at the same time, the link bandwidth available will be reduced and the flows will interfere with each other.

In order to calculate the effect of this cross interference, the interference index (II), defined as follows is used.

$$\mathrm{II}\left(\mathcal{P}_i\right) \stackrel{\text{def}}{=} \left(D^{tp}\left(\mathcal{P}_i\right) + \widehat{Q}\left(\mathcal{P}_i\right) - D_i^{\max}\right) + \left[U\left(\mathcal{P}_i\right)\right]^+,$$

for a given set of paths $\widehat{\rho}$(4.6)

Where:

- $\widehat{Q}\left(\mathcal{P}_i\right) = \sum_{\pi_s \in \mathcal{P}_i} \widehat{Q}_i^s$ represents the worst-case queuing delay.
- $D^{tp}\left(\mathcal{P}_i\right) = \sum_{(\pi_{s'}, \pi_s) \in \mathcal{P}_i} D_{s',s}^{tr}$ is the transmission and propagation delay for the path $\mathcal{P}_i$.
- $D^{tp}\left(\mathcal{P}_i\right) + \widehat{Q}\left(\mathcal{P}_i\right) - D_i^{\max}$ is the difference of the potential worst case for end-to-end delay and the relative deadline $D_i^{\max}$.
- $U\left(\mathcal{P}_i\right) = \sum_{(\pi_{s'}, \pi_s) \in \mathcal{P}_i} \frac{B_t}{B^R(\pi_{s'}, \pi_s)}$ represents the worst-case link utilization. Where, $U\left(\mathcal{P}_i\right)$ is a large, non-negative, number (*i.e.*, $\exists\left(\pi_{s'}, \pi_s\right) : B^R\left(\pi_{s'}, \pi_s\right) = 0$). This is done to avoid both divide by zero errors and high amounts of interference on a link.
- The operator $[x]^+ = x$ if $x \geq 0$, else $\bar{x}^+$. Where, $\bar{x}^+$ is an arbitrary large, non-negative, number.

If the resulting value of II is positive, it means that the path encounters interference. With a larger positive value indicating more interference is present. If the value of II is negative, it means the path has communication laxity, or slack. Again, a "larger" negative number represents more slack being present in this path.

If given a set of paths which are potential swap candidates, metric II defines how much interference from other flows a given path $\mathcal{P}_i$ encounters.

The work done by [34] showed lower end-to-end delays when flows encounter less interference from queuing. Thus, the path selection algorithm aims to, for each flow F_i, determine a path $\mathcal{P}_i$ that with the least interference, or, the smallest II ($\mathcal{P}_i$).

4.5 Path Layout Algorithm

In Kashinath et al. [31], they develop an iterative, pruning-based scheme (Algorithm 1) in order to calculate a flow's path. The algorithm has two steps. First, for all flows, generate a set of candidate paths and second, assign generated paths to flows by discarding paths with higher interference. We used this path layout scheme (shown in Algorithm 1) as a base case in our implementation.

First, beginning with a set of candidate paths $\widehat{\rho} = \bigcup_{F_i \in \mathcal{F}} \{\rho_i\}$. Let ρ_i represents a set of candidate paths for flow F_i (Line 2) and then compute the interference index of all paths $\mathcal{P}_i \in \widehat{\rho}$ (Line3). All simple paths , from s_i to t_i can be calculated by utilizing previously proposed approaches [39, 53, 57]. This calculation requires the given network topology and the source s_i and destination t_i for a flow F_i. Then, from the output of this calculation, we only select a set of candidate paths ρ_i that

are able to ensure the below condition is met:

$\forall \mathcal{P}_k \in \rho_i : \sum_{(\pi_{s'}, \pi_s) \in \mathcal{P}_k} D^{tp}_{s',s} \leq D^{\max}_i$

Any paths that fail to meet this condition are considered trivially unschedulable. Meaning, these paths fail to meet the delay constraint defined in Eq. 4.4.

Second, during each iteration of the algorithm, the path $\mathcal{P}\max$ (the path with the highest II or the maximum interference) is discarded. After this discard, the interference index is then recalculated for the remaining candidate paths $\widehat{\rho} \backslash \mathcal{P}\max$ (Line 14). During any of these iterations, if the only candidate path is the the path with the highest II (*e.g.*, $|\rho_i| = 1$), then this path is assigned to the flow (Line 17). The algorithm will continue as long as there is at least one flow, with more than one candidate paths ((*i.e.*, $\exists F_i : |\rho_i| > 1$). If an assigned path is found to violate the QoS constraint for a given flow (Eq.4.4 and Eq. 4.5), then a feasible path can not be found and the flow-set is marked as unschedulable (Line 23).

4.5.1 Time Complexity

The step of reducing candidates (Lines 4-20), has a finite number of iterations due to the finite set of initial candidate paths. This results in Algorithm 1 having a time complexity of $O(|\widehat{\rho}|)$.

Step 1 of Algorithm 1 is the generation of $\widehat{\rho}$, which is all possible paths for each flow. Given only a single path, the complexity to find a solution would be $O(\mathcal{V} + \mathcal{E})$. However, the number of simple paths that a topology could have can be quite a bit larger than one. Resulting in a complexity of closer to $O(n!)$ for a

complete graph of order n. Finally though, the majority of practical networks are not fully-connected [33], resulting in the complexity of candidate path generation being polynomial, or $O(|\widehat{\rho}|(\mathcal{V}+\mathcal{E}))$. However, since these paths are computed offline the polynomial overhead for the path layout does not impact real-time performance.

Chapter 5: Spatially-Varying Locally Static Priorities

5.1 Research Question

The goal of a Real-Time Network (RTN) is to ensure that all packets arrive at their appropriate destinations while meeting their basic Quality of Service requirements (reliability, delay, jitter, and bandwidth) [55]. Accomplishing this goal poses a difficult problem; determining a schedule to satisfy end-to-end deadlines of all packets in any given system. Finding a feasible schedule in non-preemptive context is known to be NP-complete [16, 44].

Furthermore, scheduling packets in a network becomes more difficult as the network load and demands of users or machines within the network increase. Networks are made up of multiple hubs. Each individual hub may experience unique conditions depending on the amount and kind of traffic it must process.We focused on networks extremely sensitive to perturbation and require guarantees on delivery times for packets. Examples include power substations, avionics, industrial control systems, manufacturing plants, automobiles and more.

Performance constraints of networks can be captured by the notion of flow priorities and deadlines. Flow priority governs packet processing at each switch, and resulting delays can dictate whether deadlines for that flow are met or not. Not all configurations of priorities and deadlines, local or otherwise, will guarantee that all flows throughout the network will meet their requirements. [56]. For the case of an

unschedulable network topology (flow does not meet their deadline requirements). So far, related work using static priorities maintain the same priorities throughout the network. We will investigate whether spatially-varying locally static priorities can help improve schedulability.

5.1.1 Motivating Example

The work by Kumar et al. [34] took the approach of setting packet servicing priorities on a flow-by-flow (static) basis in a deadline-monotonic fashion, and this achieves a certain performance [34]. Deadline-monotonic priority settings honor the urgency of traffic as the key factor. Instead of keeping the priorities same through out the network we explore using different priorities for the flow in different parts of the network. This suggests a more surgical refinement of flow priorities that takes into account one flow's characteristics vs another at a given switch. Our approach will attempt to utilize the slack available in a higher priority flow to allow a previously non-schedulable flow to meet its deadline. This is done by switching priorities for the two flows at the switch [24]. We will determine the degree of improvement achieved by the surgical adjustments of priorities through various heuristic approaches.

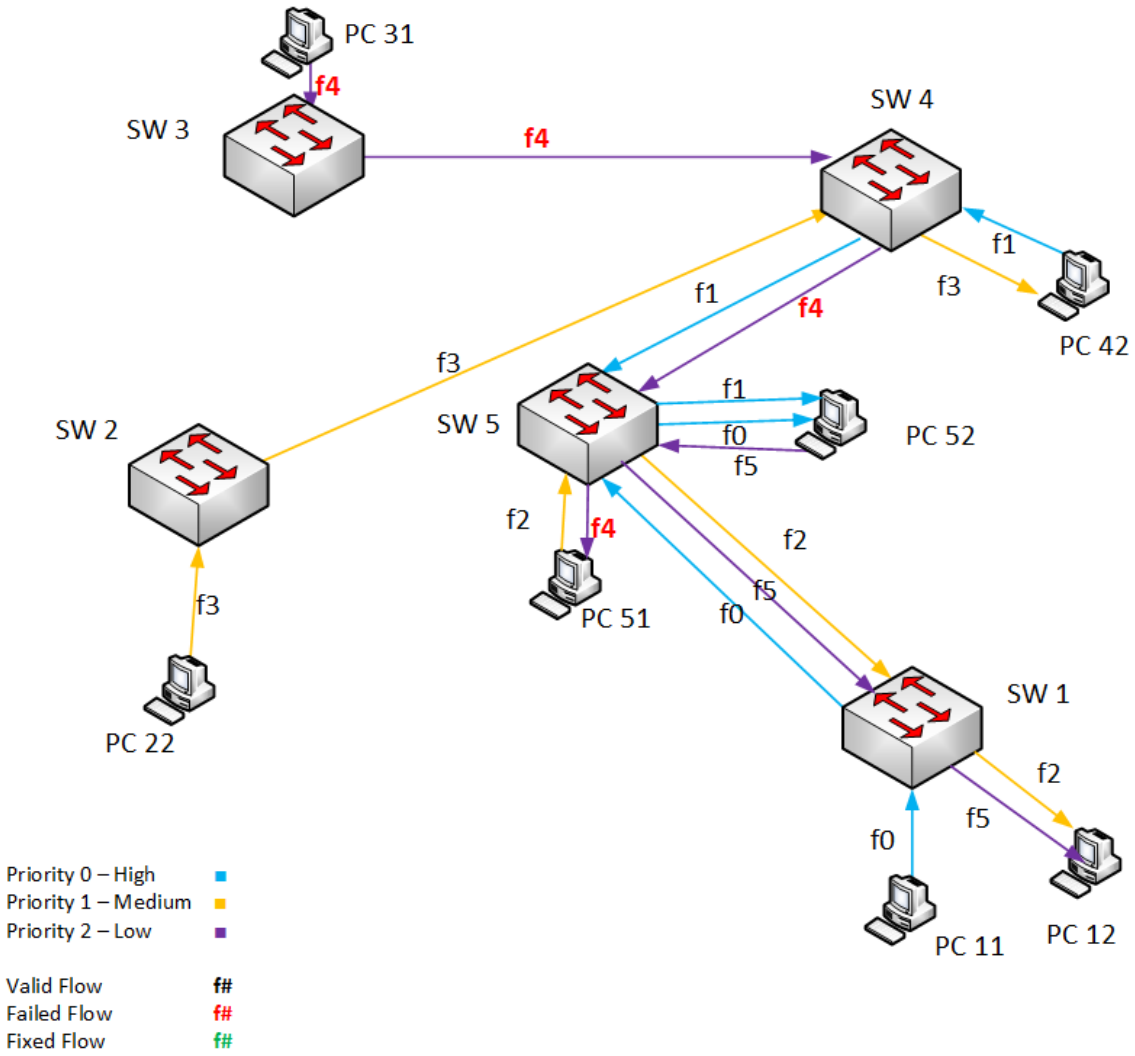

Figure 5.1: Original Topology: The 5 switch topology used for this experiment. Each switch would also connect to the controller via a management port (not shown). Within this topology, Flow 4 is unable to reach its destination.

As an example, we use the topology shown in Figure 5.1. This example contains a network topology with six flows. Of these six flows, all are schedulable within their required deadline, with the exception of flow $f4$. Flow $f4$ has a required deadline of

2.24ms. However, the total delay for flow $f4$ is 2.4126ms, resulting in it being not schedulable. In order to improve flow $f4$'s performance, we can leverage a higher priority flow that has more slack, such as $f1$. Since $f1$ only has two switches in it's path, it has enough slack such that we can swap its priority relative to $f4$ at $sw4$. The extra slack means that $f1$ doesn't need to have a higher priority than $f4$ at every switch their paths share, even though $f1$ has a shorter deadline. See Figure 5.2.

The System Model utilized is described in Section 4, System Model. For details about the implementation of the Network and Flow Specifications, Delay and Bandwidth Constraints, QoS-Aware Path Selection, Interference Index, or our Path Layout Algorithm, please refer to the aforementioned section.

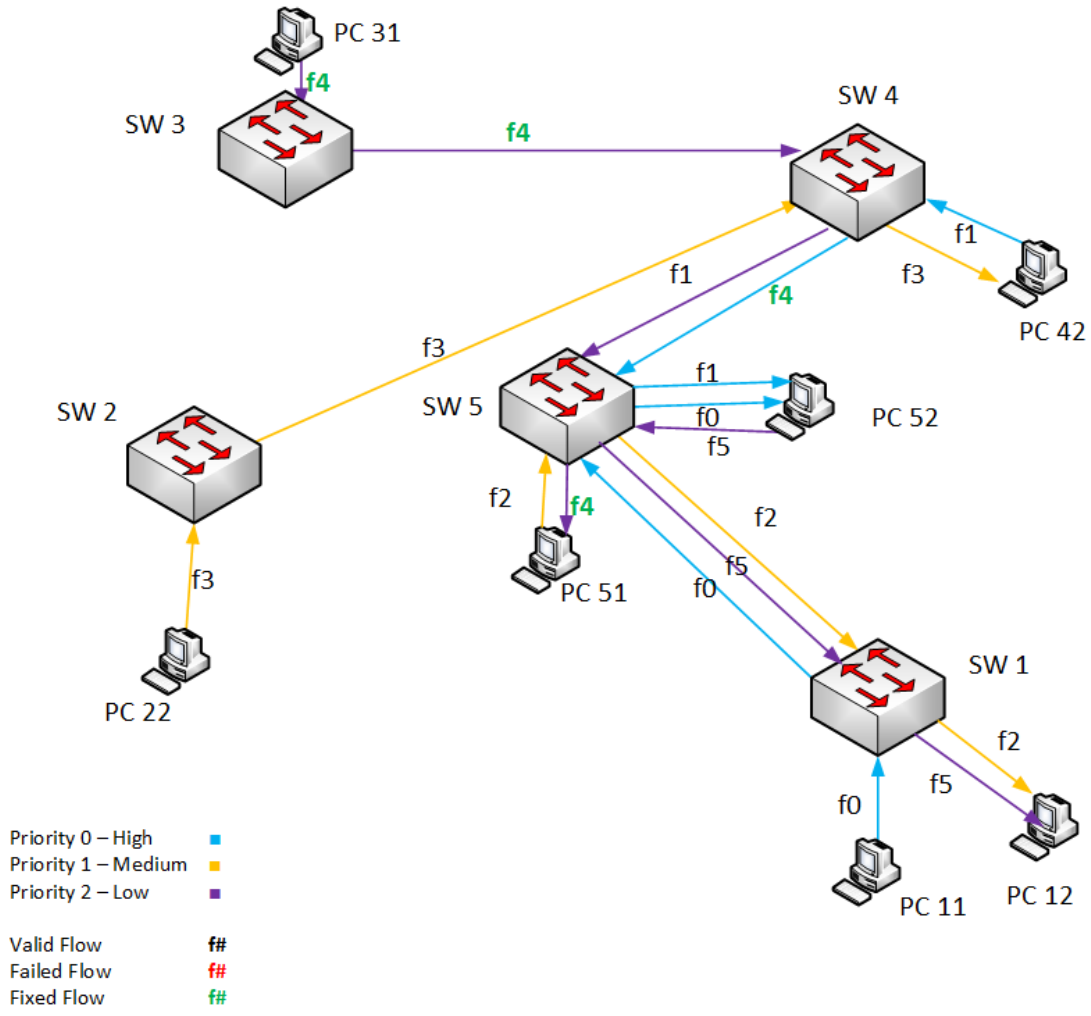

Figure 5.2: Improved Topology: The 5 switch topology used for this experiment. This version shows the resulting corrections made by our algorithm, allowing Flow 4 to be scheduled.

5.2 Proposed Solution

In order to solve the problem of network scheduling and deadline guarantees, algorithms have been developed for other networking contexts that utilize local priority

assignment to better address competition for resources in a network [20]. The essential idea is to use the schedulability conditions proposed by Park [44] along with an SDN in order to leverage the high-level view of the network an SDN provides to better guarantee end-to-end deadlines.

The main benefit of SDN comes when additional flows are added to a system. The SDN can determine the schedulability of the system when new flows are added [44] by leveraging its awareness of network characteristics. Kumar et al. proposes a framework that solves a multi-constraint optimization problem using SDN in a way that makes SDNs more aware of delays in a system. It uses a heuristic algorithm based on static priority for each flow to solve an optimization problem with multiple constraints. Additionally, it also isolates flows in the system into discrete queues to better guarantee that jobs meet their end-to-end deadlines [34].

We also use heuristic algorithms to improve schedulability for their framework that leverage the ability to refine the priorities switch-by-switch within each flow. Details of our heuristic algorithm are described below. This includes the network specifications, calculations for both delay and bandwidth, path selection, Interference Index, the algorithm determining path layout, and our heuristics for improving priorities.

5.2.1 Spatially-Varying Static Priority Scheduling Heuristics

As mentioned in the section (5.1.1). We pursue a surgical approach where we refine flow priorities, taking into account one flow's characteristics vs. another at a given

switch. We swap priorities for two flows at the switch in an attempt to utilize the slack available in the higher priority flow to allow the non-schedulable flow to meet its deadline. This approach serves as the basis of our heuristics, described below.

5.2.1.1 Heuristic 1 (Single Swap at Switch).

Given a topology that fails (non-schedulable), we look at the flow(s) that exceed their deadline with current paths assigned by DBAPS [31]. For a given flow's path, we start at the last switch S where the flow is still under the deadline. We then create a list of other flows that share the same forward link. We iterate through candidate list in decreasing slack order. We swap flow priority of the failed flow with the candidate flow at the switch S. We test if this change results in the failed flow not exceeding its deadline. If the flow does not exceed its deadline, we test if the topology is now valid (schedulable). If the topology is now valid that is, schedulable, we return success. If the topology is not yet schedulable, we attempt to fix the next failed flow. If that flow then still exceeds the deadline, we test the next candidate flow with higher slack. If no candidates result in a success after swapping flow priority, we attempt a second fix on that failed flow, by trying the priority swap at the switch before the current switch S. Note that in this process, for a given improvement attempt, if that attempt does not make the flow schedulable, then that change is tossed aside and the next attempt will be made against the original topology and performance data for that flow. This approach is captured in Algorithm 2. Applying this heuristic to the network depicted in Figure 5.2 serves as an illustration.

A key step in our approach is to choose a swap target, based on whether or not the target in question and our failed flow have an overlapping path. A flow is also a valid swap target if it and a failed flow share a forward link in their paths. In Figure 5.1, $f4$ starts at $PC31$, travels through $s3$ to $s4$ and then to $s5$, ending at PC 51. $f1$ starts at PC 42, travels through $s4$ to $s5$, and then ends at PC 52. The link shared is the link between $s4$ and $s5$. $f1$ has a required deadline of 1.76ms and completes its path in 0.9681ms, meaning $f1$ is schedulable. Because $f1$ is under its deadline and shares the link $s4$ to $s5$, $f1$ is a valid swap target.

In the experimental framework, total path delay is computed by adding up the queuing delays at each switch, plus the propagation delays between switches. The delay at each switch is obtained by adding together two different delay values. The first, FIFO delay, is the time packets must wait due to other flows with the same priority. The second, Interference Delay, is the time packets must wait due to flows with higher priority. Prior to performing our priority swap, $f4$ has a cumulative delay of 0.6116ms and $f1$ has a cumulative delay of 0.0676ms.

After performing our priority swap for $f4$ (originally priority 2) and $f1$ (originally priority 0) at switch 4, $f4$ now has a priority level 0 and $f1$ now has a priority level 2. The cumulative queuing delay of $f1$ at switch 4 is increased from 0.0676ms to 0.0987ms, which results in the total path delay of $f1$ increasing from 0.9681ms to 0.9992ms. With this delay increase, $f1$ still meets its deadline of 1.76ms, and so continues to be schedulable. The cumulative queuing delay of $f4$ at switch 4 decreases from 0.6116ms to 0.4313ms. This reduces $f4$'s total path delay, bringing it down to 2.2323ms, which is under the deadline of 2.24ms.

The heuristic has succeeded in making the failed flow schedulable without making the swap target flow unschedulable. This means the network as a whole is now schedulable.

Algorithm 2 Single Swap at Switch

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
1:  function Heuristic1(N, F, G, S, s, t)
2:      for each failed flow g_i ∈ G do
3:          for each switches s_j under deadline do
4:              Candidates ← []
5:              for each flow f_k ∈ F do
6:                  if f_k shares a forward link with s then
7:                      Candidates.append(f_k)
8:                  end if
9:              end for
10:             Sort Candidates in decreasing order of slack
11:             for each f_k ∈ Candidates do
12:                 /* Swap flow priority of failed flow with candidate flow at a single switch. */
13:                 Swap(g_i, f_k)
14:                 /* Test if this change resulted in the failed flow not exceeding its deadline. */
15:                 if !ExceedsDeadline(g_i) then
16:                     /* Test if the topology is now valid */
17:                     if TopologyValid(G) then
18:                         Return Success
19:                     else
20:                         /* The flow has been fixed, so we proceed to trying to improve the next
    failed flow */
21:                         Break
22:                     end if
23:                 end if
24:             end for
25:         end for
26:     end for
27: end function
```

5.2.1.2 Heuristic 2: Single Boost at Switch

Is identical to Heuristic 1, except that rather than swapping priorities at a particular switch, the failed flow is given the priority of the candidate flow at that switch. The candidate flow's priority is not changed. In other words, we attempt to make a singular improvement (at one switch) that is sufficient to make the failed flow schedulable, but it only involved changing priority of the failed flow. If that works, we try the same on other failed flows. Algorithm 3, depicts this heuristic.

5.2.1.3 Heuristic 3: Single Swap at Switch and Boost Path

Is identical to Heuristic 1, except we apply the new priority to all switches in the failed flow's path. Here we broaden the scope of each change, adjusting the priority at every switch of the failed flow. Compared to Heuristic 1, this heuristic makes larger changes to the failed flow in an attempt to see if it can improve more flows than Heuristic 1. If that works, we try the same on other failed flows in our attempt to make the topology schedulable. This heuristic is captured in Algorithm 4.

Algorithm 3 Single Boost at Switch

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
1: function Algorithm2(N, F, G, S, s, t)
2:     for each failed flow g_i ∈ G do
3:         for each switches s_j under deadline do
4:             Candidates ← []
5:             for each flow f_k ∈ F do
6:                 if f_k shares a forward link with s then
7:                     Candidates.append(f_k)
8:                 end if
9:             end for
10:            Sort Candidates in decreasing order of slack
11:            for each f_k ∈ Candidates do
12:                /* Boost flow priority of failed flow with candidate flow at a single switch. */
13:                Boost(g_i, f_k)
14:                /* Test if this change resulted in the failed flow not exceeding its deadline. */
15:                if !ExceedsDealine(g_i) then
16:                    /* Test if the topology is now valid */
17:                    if TopologyValid(G) then
18:                        Return Success
19:                    else
20:                        /* The flow has been fixed, so we proceed to trying to improve the next
       failed flow */
21:                        Break
22:                    end if
23:                end if
24:            end for
25:        end for
26:    end for
27: end function
```

Algorithm 4 Single Swap at Switch and Boost Path

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
1: function Algorithm3(N, F, G, S, s, t)
2:     for each failed flow g_i ∈ G do
3:         for each switches s_j under deadline do
4:             Candidates ← []
5:             for each flow f_k ∈ F do
6:                 if f_k shares a forward link with s then
7:                     Candidates.append(f_k)
8:                 end if
9:             end for
10:            Sort Candidates in decreasing order of slack
11:            for each f_k ∈ Candidates do
12:                /* Swap flow priority of failed flow with candidate flow at a single switch. */
13:                Swap(g_i, f_k)
14:                /* Boost the flow priority of all failed flow switches to the priority of the candidate
    flow. */
15:                Boost(g_i, f_k)
16:                /* Test if this change resulted in the failed flow not exceeding its deadline. */
17:                if !ExceedsDealine(g_i) then
18:                    /* Test if the topology is now valid */
19:                    if TopologyValid(G) then
20:                        Return Success
21:                    else
22:                        /* The flow has been fixed, so we proceed to trying to improve the next
    failed flow */
23:                        Break
24:                    end if
25:                end if
26:            end for
27:        end for
28:    end for
29: end function
```

Algorithm 5 Single Boost at Switch and Boost Path

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
1: function Algorithm4(N, F, G, S, s, t)
2:     for each failed flow g_i ∈ G do
3:         for each switches s_j under deadline do
4:             Candidates ← []
5:             for each flow f_k ∈ F do
6:                 if f_k shares a forward link with s then
7:                     Candidates.append(f_k)
8:                 end if
9:             end for
10:            Sort Candidates in decreasing order of slack
11:            for each f_k ∈ Candidates do
12:                /* Boost the flow priority of all failed flow switches to the priority of the candidate flow. */
13:                Boost(g_i, f_k)
14:                /* Test if this change resulted in the failed flow not exceeding its deadline. */
15:                if !ExceedsDealine(g_i) then
16:                    /* Test if the topology is now valid */
17:                    if TopologyValid(G) then
18:                        Return Success
19:                    else
20:                        /* The flow has been fixed, so we proceed to trying to improve the next failed flow */
21:                        Break
22:                    end if
23:                end if
24:            end for
25:        end for
26:    end for
27: end function
```

5.2.1.4 Heuristic 4: Single Boost at Switch and Boost Path("Boost Path")

Is identical to Heuristic 2, where adjustment is made only to the failed flow priority at the given switch, except that we also now apply the new priority to all switches in the failed flow's path. We again have the broadened scope of change as in Heuristic 3, where priority is adjusted at all switches in the failed flow, and adjustments that do not achieve failed flow schedulability are forgotten, even if they made incremental improvement. If this fixes the failed flow, we try the same on other failed flows. This heuristic is depicted in Algorithm 5.

5.2.1.5 Heuristic 5: Multiple Switch Swap

Is similar to Heuristic 1, but with a key difference that changes are kept if they make incremental improvement to the the failed flow's schedulability, even if they don't solve it outright. As an illustration, if we perform a priority ***swap*** at a given switch in the failed flow, and the resulting total path delay is still over the deadline, but has been reduced, we retain that adjustment as part of the input for the next step. In this way, incremental reductions in total path delay can add up as we back up through the switches of the flow and make swaps. (See Figure 5.4). This heuristic is depicted in Algorithm 6.

5.2.1.6 Heuristic 6: Multiple Switch Boost

Is similar to Heuristic 2, but with a key difference that changes are kept if they make incremental improvement to the the failed flow's schedulability, even if they don't solve it outright. As an illustration, if we perform a priority ***adjustment*** at a given switch in the failed flow (priority is set as per what would have been the swap candidate flow's priority), and the resulting total path delay is still over the deadline, but has been reduced, we retain that adjustment as part of the input for the next step. In this way, incremental reductions in total path delay can add up as we back up through the switches of the flow and make swaps. This heuristic is depicted in Algorithm 7.

Algorithm 6 Multiple Switch Swap

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
 1: function Algorithm5(N, F, G, S, s, t)
 2:     FailedFlowSequences ← generateFailedFlowSequences(G)
 3:     for each FailedFlowSequence ffs_h ∈ FailedFlowSequences do
 4:         for each failed flow g_i ∈ FailedFlowSequence do
 5:             for each switches s_j under deadline do
 6:                 Candidates ← []
 7:                 for each flow f_k ∈ F do
 8:                     if f_k shares a forward link with s then
 9:                         Candidates.append(f_k)
10:                     end if
11:                 end for
12:                 Sort Candidates in decreasing order of slack
13:                 for each f_k ∈ Candidates do
14:                     /* Swap flow priority of failed flow with candidate flow at a single switch. */
15:                     Swap(g_i, f_k)
16:                     /* Test if this change improved the flow's performance. */
17:                     if PerformanceImproved(g_i) then
18:                         /* Test if this change resulted in the failed flow not exceeding its deadline. */
19:                         if !ExceedsDealine(g_i) then
20:                             /* Test if the topology is now valid */
21:                             if TopologyValid(G) then
22:                                 Return Success
23:                             else
24:                                 /* The flow has been fixed, so we proceed to trying to improve the next failed flow */
25:                                 Go To Line 2
26:                             end if
27:                         else
28:                             /* Keep improvement and attempt to fix at the switch before the current one. */
29:                             Go To Line 3
30:                         end if
31:                     end if
32:                 end for
33:             end for
34:         end for
35:         Return error
36:     end for
37: end function
```

Algorithm 7 Multiple Switch Boost

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
1: function Algorithm6(N, F, G, S, s, t)
2:     FailedFlowSequences ← generateFailedFlowSequences(G)
3:     for each FailedFlowSequence ffs_h ∈ FailedFlowSequences do
4:         for each failed flow g_i ∈ FailedFlowSequence do
5:             for each switches s_j under deadline do
6:                 Candidates ← []
7:                 for each flow f_k ∈ F do
8:                     if f_k shares a forward link with s then
9:                         Candidates.append(f_k)
10:                    end if
11:                end for
12:                Sort Candidates in decreasing order of slack
13:                for each f_k ∈ Candidates do
14:                    /* Boost flow priority of failed flow with candidate flow at a single switch. */
15:                    Boost(g_i, f_k)
16:                    /* Test if this change improved the flow's performance. */
17:                    if PerformanceImproved(g_i) then
18:                        /* Test if this change resulted in the failed flow not exceeding its deadline. */
19:                        if !ExceedsDealine(g_i) then
20:                            /* Test if the topology is now valid */
21:                            if TopologyValid(G) then
22:                                Return Success
23:                            else
24:                                /* The flow has been fixed, so we proceed to trying to improve the next failed flow */
25:                                Go To Line 2
26:                            end if
27:                        else
28:                            /* Keep improvement and attempt to fix at the switch before the current one. */
29:                            Go To Line 3
30:                        end if
31:                    end if
32:                end for
33:            end for
34:        end for
35:        Return error
36:    end for
37: end function
```

5.2.1.7 Heuristic 7: Multiple Switch Swaps and Boost Path

Is similar to Heuristic 3, except that incremental changes are kept if they improve performance, so this Heuristic is more carefully exploring all the opportunities for achieving schedulability. In Heuristic 3, if a change improves the failed flow's performance but does not get it under the deadline, that change is tossed aside rather than being incorporated for the next improvement step. This heuristic is depicted in Algorithm 8.

5.2.1.8 Heuristic 8: Multiple Switch Boosts and Boost Path

Is similar to Heuristic 4, except that incremental changes are kept if they improve performance, so this Heuristic is also more carefully exploring all the opportunities for achieving schedulability. In Heuristics 4, if a change improves the failed flow's performance but does not get it under the deadline, that change is tossed aside rather than being incorporated for the next improvement step. This heuristic is depicted in Algorithm 9.

Algorithm 8 Multiple Switch Swaps and Boost Path

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
1: function Algorithm7(N, F, G, S, s, t)
2:     FailedFlowSequences ← generateFailedFlowSequences(G)
3:     for each FailedFlowSequence ffs_h ∈ FailedFlowSequences do
4:         for each failed flow g_i ∈ FailedFlowSequence do
5:             for each switches s_j under deadline do
6:                 Candidates ← []
7:                 for each flow f_k ∈ F do
8:                     if f_k shares a forward link with s then
9:                         Candidates.append(f_k)
10:                    end if
11:                end for
12:                Sort Candidates in decreasing order of slack
13:                for each f_k ∈ Candidates do
14:                    /* Swap flow priority of failed flow with candidate flow at a single switch. */
15:                    Swap(g_i, f_k)
16:                    /* Boost the flow priority of all failed flow switches to the priority of the
       candidate flow. */
17:                    Boost(g_i, f_k)
18:                    /* Test if this change improved the flow's performance. */
19:                    if PerformanceImproved(g_i) then
20:                        /* Test if this change resulted in the failed flow not exceeding its deadline.
       */
21:                        if !ExceedsDealine(g_i) then
22:                            /* Test if the topology is now valid */
23:                            if TopologyValid(G) then
24:                                Return Success
25:                            else
26:                                /* The flow has been fixed, so we proceed to trying to improve the
       next failed flow */
27:                                Go To Line 2
28:                            end if
29:                        else
30:                            /* Keep improvement and attempt to fix at the switch before the current
       one. */
31:                            Go To Line 3
32:                        end if
33:                    end if
34:                end for
35:            end for
36:        end for
37:        Return error
38:    end for
39: end function
```

Algorithm 9 Multiple Switch Boosts and Boost Path

Input: The network $N = (V, E)$, flow(s) $F = \{f_1, \ldots, f_n\}$, failed flow(s) $H = \{g_1, \ldots g_m\}$, switches $\pi = \{s_1, \ldots, s_l\}$, source s, destination t

Output: Schedulable flow if possible. Otherwise, note non-schedulability.

```
1: function Algorithm8(N, F, G, S, s, t)
2:     FailedFlowSequences ← generateFailedFlowSequences(G)
3:     for each FailedFlowSequence ffs_h ∈ FailedFlowSequences do
4:         for each failed flow g_i ∈ FailedFlowSequence do
5:             for each switches s_j under deadline do
6:                 Candidates ← []
7:                 for each flow f_k ∈ F do
8:                     if f_k shares a forward link with s then
9:                         Candidates.append(f_k)
10:                    end if
11:                end for
12:                Sort Candidates in decreasing order of slack
13:                for each f_k ∈ Candidates do
14:                    /* Boost flow priority of failed flow with candidate flow at a single switch. */
15:                    Boost(g_i, f_k)
16:                    /* Boost the flow priority of all failed flow switches to the priority of the
       candidate flow. */
17:                    Boost(g_i, f_k)
18:                    /* Test if this change improved the flow's performance. */
19:                    if PerformanceImproved(g_i) then
20:                        /* Test if this change resulted in the failed flow not exceeding its deadline.
       */
21:                        if !ExceedsDealine(g_i) then
22:                            /* Test if the topology is now valid */
23:                            if TopologyValid(G) then
24:                                Return Success
25:                            else
26:                                /* The flow has been fixed, so we proceed to trying to improve the
       next failed flow */
27:                                Go To Line 2
28:                            end if
29:                        else
30:                            /* Keep improvement and attempt to fix at the switch before the current
       one. */
31:                            Go To Line 3
32:                        end if
33:                    end if
34:                end for
35:            end for
36:        end for
37:        Return error
38:    end for
39: end function
```

5.3 Evaluation

Our strategy for determining effectiveness is to use the Kashinath et al. [31] simulator to generate topologies that are unschedulable with DBAPS that we then can try to repair with our heuristics. This involves modifications to their simulator to extract information, and implementing and testing a new standalone system which will execute and evaluate our heuristics.

The Kashinath et al. [31] simulator generates random topologies with priority and deadline characteristics and tests whether they are schedulable or not. We capture unschedulable topologies by instrumenting their system to collect all information that we will need for our analysis. Careful analysis of their simulator architecture allows us to identify the proper location to position our instrumentation and the proper run context to extract data. Each pass through schedulability experiment code then results in a file being generated in the event that the topology was not schedulable. This network info file (netInfoFile) contains the delay information for each switch in each flow as well as the information used to compute that. Our heuristic evaluation system takes the netInfoFile as input and generates a revised netInfoFile as output.

For our focused problem of assessing improvements to schedulability, it is simpler to write our system from scratch rather than trying to reuse simulator code directly. Because of this, we have the task of ensuring that we implement delay calculations exactly as their system does and this is done by writing unit tests that compare our results to those saved in the input netInfoFile.

Each dataset is comprised of one thousand netInfoFiles each representing unschedulable topologies. To generate these, we wrap their schedulability experiment

code in a driver which will run it repeatedly. If the run results in a schedulable topology, it is ignored. We also filter out topologies that fall into one of three situations. First, if the propogation delay is greater than the deadline for a flow, then it is non-reparable and ignored. Second, if any of the failed flows are high priority, we deem this topology as likely unfixable and it is ignored. Third, our heuristics involve swapping priority between two flows at a particular switch and this only makes sense to do if those flows share a forward link, so topologies that do not satisfy this constraint are ignored as well. The remaining unschedulable topologies contain one or more failed flows and are candidates for our heuristics to improve.

The Kashinath et al [31] simulator generates topologies that contain a certain number of flows. Each of our datasets contains netInfoFiles that all have the same number of flows. Varying a parameter in the simulator allows that number to vary from 6 to 15, yielding ten datasets.

Each run of our heuristic evaluation system applies its process to each of the thousand netInfoFiles in the dataset and gathers statistics on how often schedulability is achieved as well as how many adjustments are made by the heuritistic on each particular topology.

5.3.1 Simulation Setup

The simulator of Kashinath [31] is instrumented to provide the input to our system in the form a networkInfoFile. Each networkInfoFile contains information on how many flows are present by listing the flow ids ([0,1,2,3,4,5]). For each flow id, the file

contains the flow's priority, propagation delay, end-to-end deadline, packet processing time, period, total path delay, and whether that flow was schedulable or not. Also, the path of the flow is represented as a list of host and switch ids (["h11", "s1", "s5", "s4", "h42"]). For each switch in each flow's path, information represented includes the fifo delay, interference delay, the delay so far along that path (which is the sum total of accumulated delays to that point), and the slack, where slack is the remaining time available before the flow's deadline is used up.

The Kashinath [31] simulator both generates topologies and computes delay characteristics. Our simulator uses their topologies as inputs and then computes the FIFO and interference delay characteristics. These computations use the period and packet processing times that are associated with each flow, found in the networkInfoFile. See Figure 5.3.

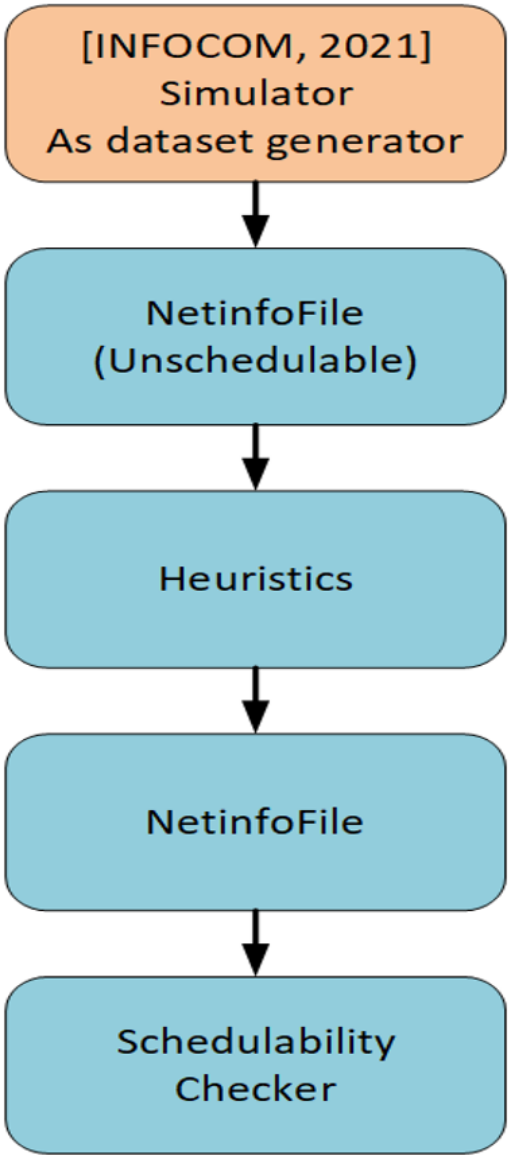

Figure 5.3: Simulation Setup

5.3.2 Result

5.3.2.1 Performance of all Spatially Varying Static Priority Heuristics

Our results from looking at only SWAP spatially varying static priority heuristics are shown in Figure 5.4.

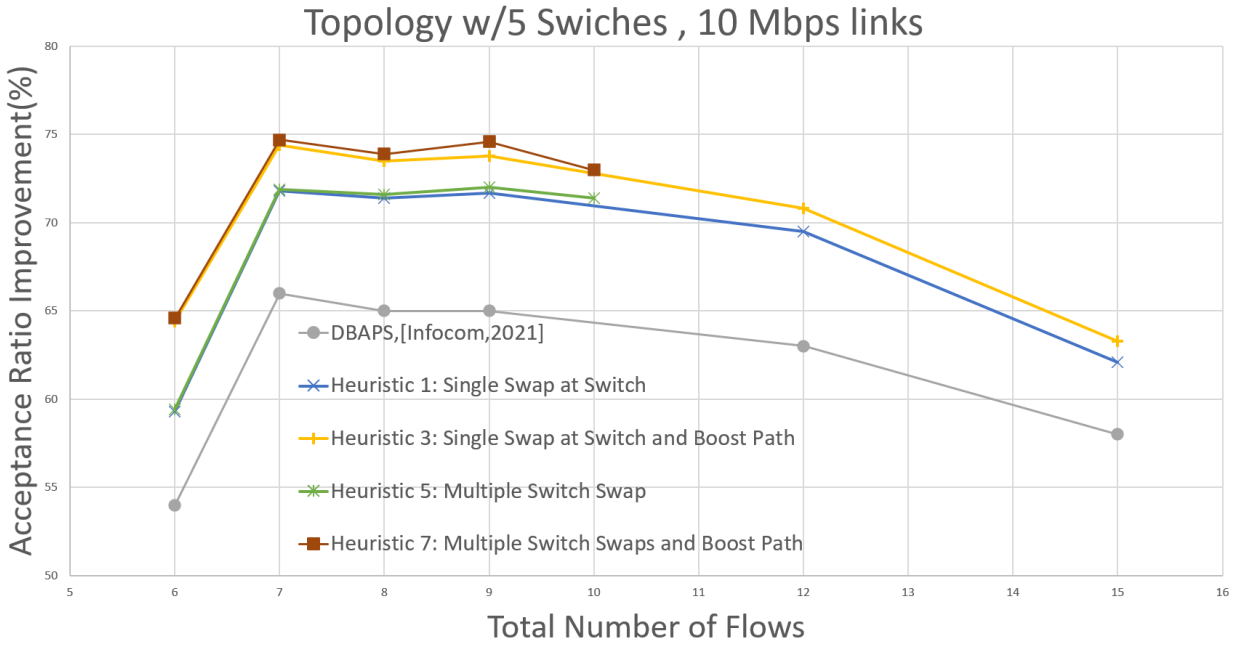

Figure 5.4: Compare Performance of all Spatially-Varying Static Priority Heuristics

5.3.2.2 Performance of all Boost Heuristics

The heuristic result from only boost variants is displayed in Figure 5.5.

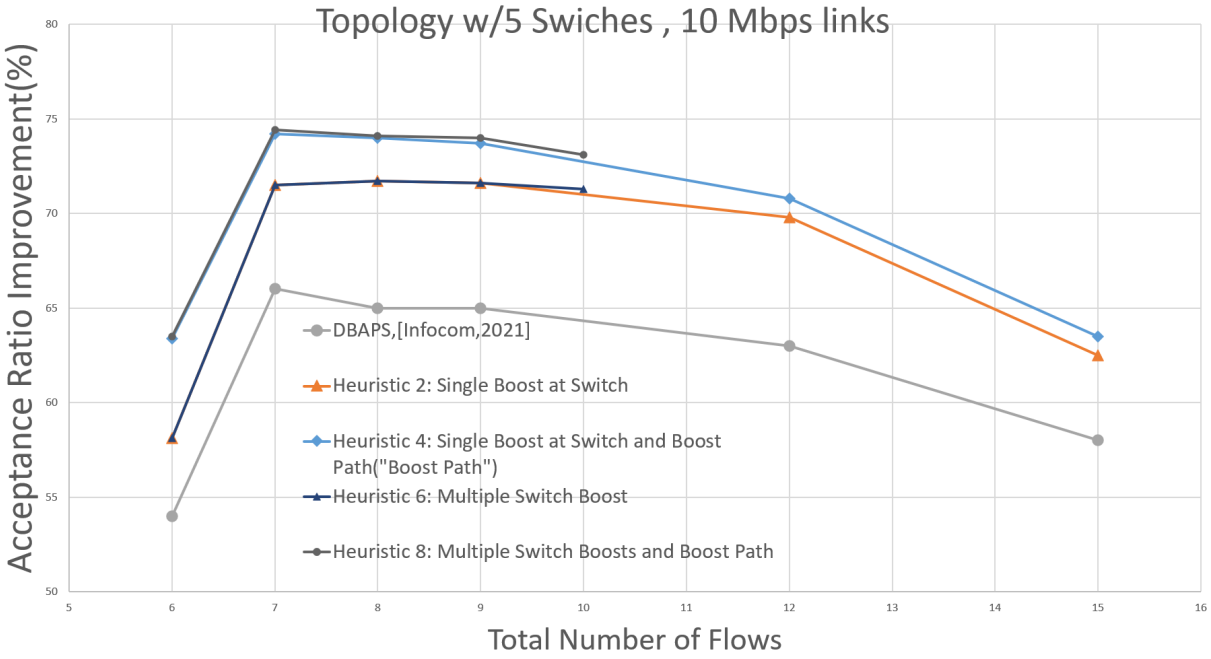

Figure 5.5: Compare Performance of all Boost Heuristics

5.3.2.3 Performance of all Heuristics

Analysis reveals that Heuristic 8 (Multiple Switch Swaps and Boost Path) has the best performance among the Heuristics - an increase of up to **10.6 %** see Table 5.1. In this Heuristic, a priority swap at multiple switches is combined with a wholesale adjustment of priority at the remaining switches of the failed flow, using the priority value that was originally held by the swap target flow in all switches in the path (boost). In these Heuristic, the FIFO delay are increase and this lead to minimize

the performance (See Figure 5.6).

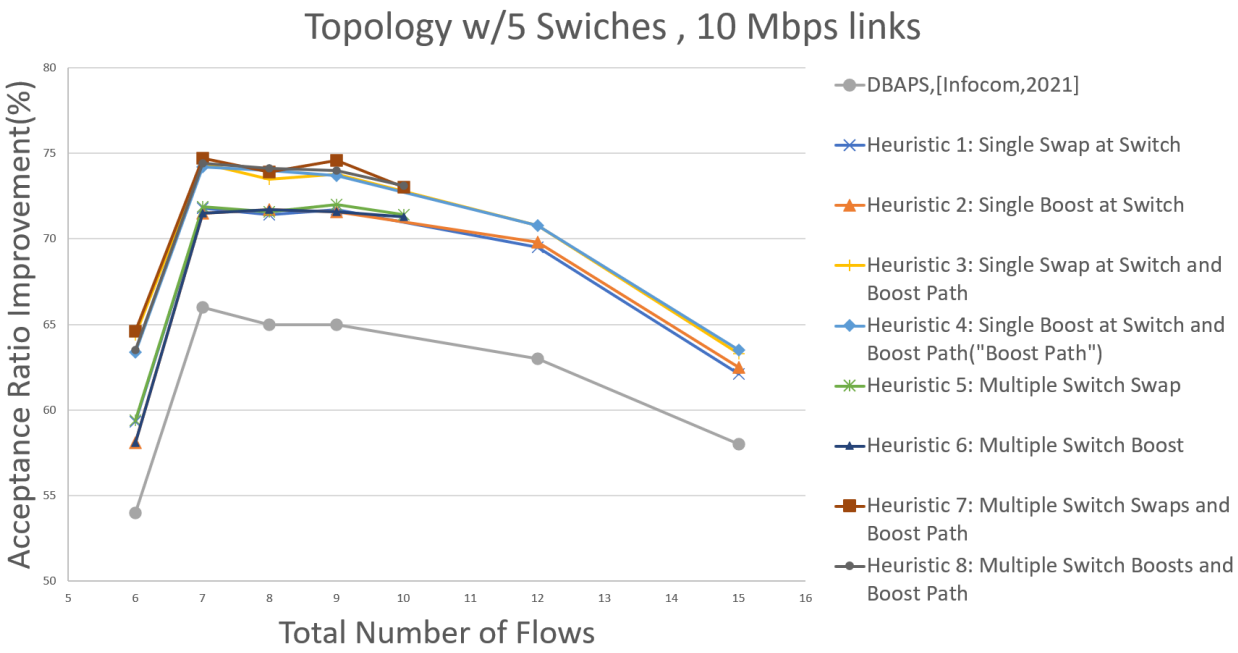

Figure 5.6: Compare Performance of all Heuristics

Table 5.1: Compare Performance of all Heuristics

Heuristics Name in Paper	Improvement Ratio(%)
Heuristic 1: Single Swap at Switch	(4.1 % - 6.7 %)
Heuristic 2: Single Boost at Switch	(4.1 % - 6.8 %)
Heuristic 3: Single Swap at Switch and Boost Path	(5.3 % - 10.4 %)
Heuristic 4: Single Boost at Switch and Boost Path("Boost Path")	(5.5 % - 9.4 %)
Heuristic 5: Multiple Switch Swap	(5.4 % - 7 %)
Heuristic 6: Multiple Switch Boost	(4.1 % - 6.7 %)
Heuristic 7: Multiple Switch Swaps and Boost Path	(8 % - 10.6 %)
Heuristic 8: Multiple Boost Swaps and Boost Path	(8.1 % - 9.5 %)

Chapter 6: Least-Slack Prioritization

6.1 Research Question

A Real-Time Network (RTN) aims to ensure that packets in a network are able to make it to their specific destination all while maintaining their Quality of Service requirements [55]. However, this presents us with a challenge - how to correctly schedule all packets in a system in such a way as to satisfy end-to-end deadlines of all packets.

Additionally, as the network load and user demands of a network increase, the difficulty of scheduling packets in this network also increases. Furthermore, as a network's internal makeup consists of multiple switches, packets may encounter different conditions at each hub that impact its traversal. Our focus is specifically on networks that are abnormally sensitive to perturbation and have guarantee requirements regarding packet delivery times. Networks that fit this description include: avionics, automobiles, industrial control systems, power substations, and manufacturing plants, to name a few.

We can use a metric of the performance constraints applicable to a network by identifying the flow priorities and the deadlines in play. The flow priority dictates the packet processing hierarchy at each switch in a network. The delays caused here can affect whether a flow's deadline requirement can be achieved. While it would be ideal for all flows to meet their requirements, not all possible priority and

deadline variations will be guaranteed to meet these requirements [56]. Rather than considering the deadline constraints from a total path point of view as was done previous works we instead consider a per-hop deadline budget constraint, computed by dividing the deadline for the entire path by the number of switch-to-switch hops in that path. Flow priorities are then derived from these per-hop deadlines and these priorities and may be used to influence search order in our algorithms.

We consider two approaches to leveraging per-hop deadlines in static path assignment. As a reference algorithm, we use Kashinath's [31] Delay and Bandwidth Aware Path Selection (DBAPS-DM), which use an end-to-end deadline-monotonic Prioritization. First, with Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (DBAPS-LSP), we investigate whether we can improve (DBAPS-DM) by allowing it to leverage priorities derived from per-hop deadlines. Second, with Greedy Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (GDBAPS-LSP), we investigate whether a first assignment of prioritise as path can perform as well or better than DBAPS.

6.1.1 Motivating Example

Our example topology contains two flows that are not able to be scheduled when (DBAPS-DM) is used to assign paths. We show that by using per-hop deadlines to dictate flow priority, the topology is then schedulable.

The example topology shown in Figure 6.1 has six flows, and flow priority is assigned according to end-to-end deadlines. Two flows are not schedulable with

this approach. The failed flows are flow $f2$ and flow $f3$. Flow $f2$ originally had a priority of 1 with a deadline requirement of 1.92ms, but a total delay of 2.021ms. Additionally, flow $f3$ also has a priority of 1 with a deadline of 2.08ms, but takes 2.0971ms to reach its target.

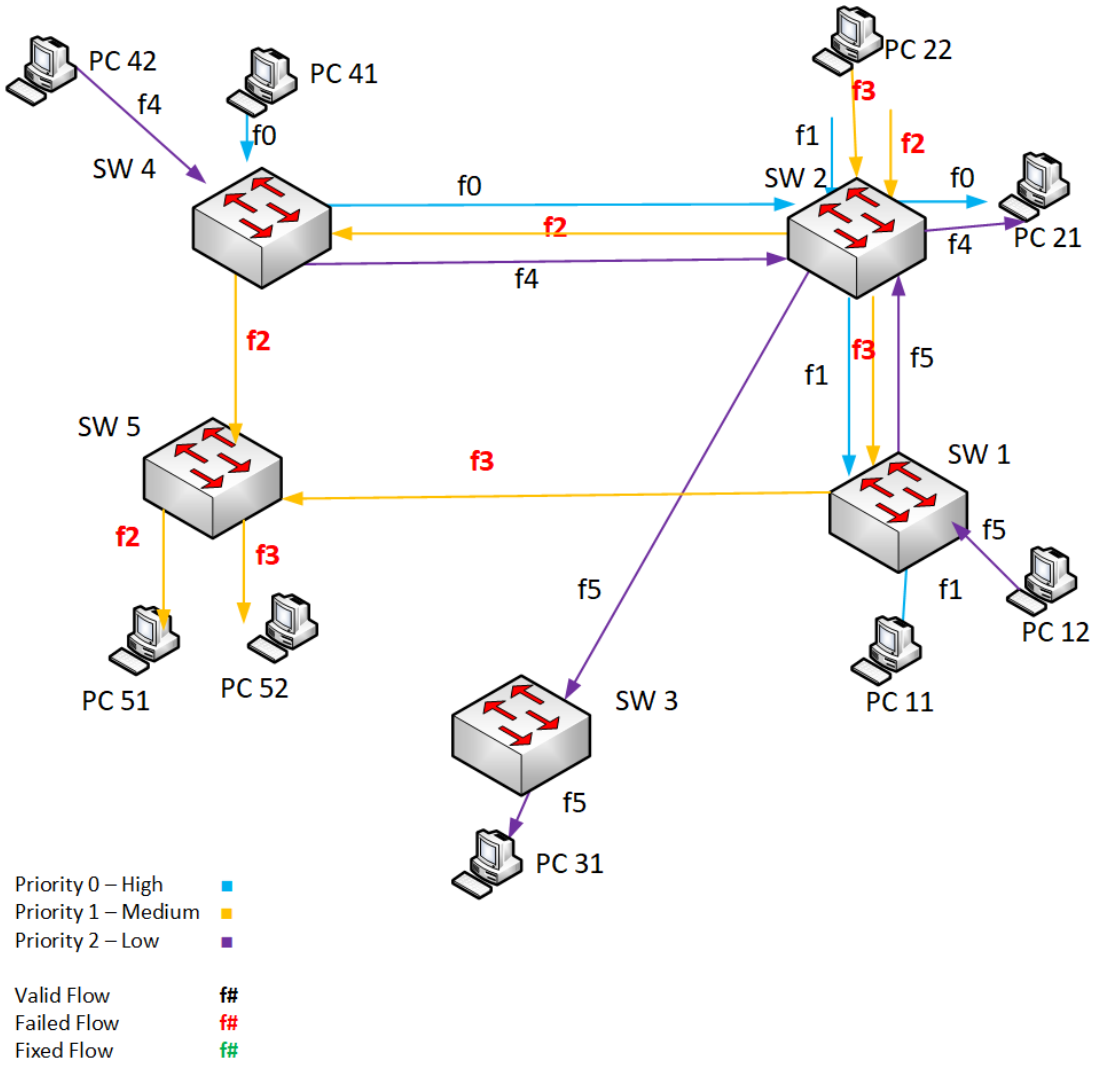

Figure 6.1: Original Topology: The 5 switch topology used for this experiment. Each switch would also connect to the controller via a management port (not shown). Within this topology, Flow 2 and Flow 3 is unable to reach its destination.

When the network flows are instead prioritized according to their per-hop deadlines, the network becomes schedulable. Here, both flow $f2$ and flow $f3$ are assigned low per-hop deadlines (0.96ms and 1.04ms respectively). These low per-hop dead-

lines result in the priority of both flows being increased to priority 0. Flow $f2$ now has a total delay of 1.8857ms and flow $f3$'s delay was shortened to 1.8717ms. This allows schedulability, even though the delays of other flows may have increased, as the increased delay may still be within the end-to-end deadline. See Figure 6.2.

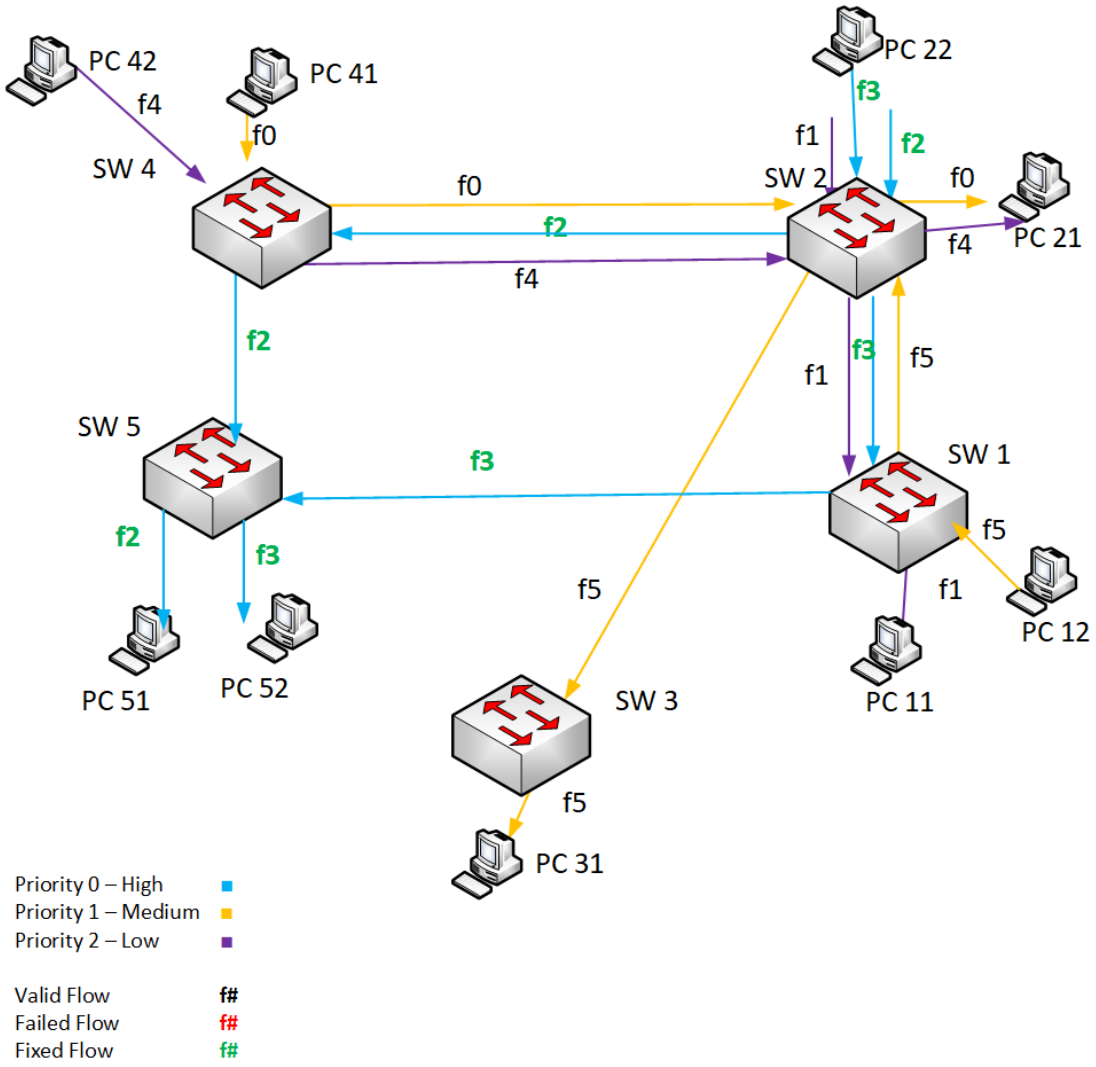

Figure 6.2: Improved Topology: The 5 switch topology used for this experiment. This version shows the resulting corrections made by our algorithm, allowing both Flow 2 and Flow 3 to be scheduled.

6.2 Proposed Solution

To find a solution to network scheduling and deadline guarantees, previously developed algorithms use local priority assignment in order to improve and reduce resource

competition [20]. Park's proposed schedulability conditions [44] are utilized, in tandem with the high-level network view provided by an SDN. Kashinath [31] expands on a different approach used by Kumar et al, which applies a multi-constraint optimization technique to optimally determine static priorities for flows (DBAPS-DM). Flows are assigned to queues, allowing prioritization at the switch level, which can improve schedulability. [31, 34] The algorithm serves as our reference algorithm in assessing performance of our algorithms.

6.3 Static Priority Assignment Algorithms

For our two algorithms, we take a least slack first approach to setting flow priorities using per-hop deadlines as a guiding constraint in our attempt at improved network utilization.

6.3.1 DBAPS-DM Algorithm.

For DBAPS-DM, path assignment is performed using interference index as a measure. The interference index takes into account both bandwidth and deadline constraints. First, path candidates are determined for each flow. A search is performed through the path candidates looking for the one with the lowest interference index for assignment to it's corresponding flow. This search is repeated until all flows have a path assigned. In each pass, the algorithm iterates over flows by order of their ID, as determined by the order of their creation. (See Algorithm 1)

6.3.2 DBAPS-LSP Algorithm.

In this Algorithm, we modified DBAPS to allow it to accept the flow priorities we derive using per-hop deadlines. We compute per-hop deadlines as follows: for each flow, find the shortest path, ignoring bandwidth constraints. These paths are then used to compute the per_hop_deadline values of each flow, where the flows with the highest priority are assigned to the shortest per_hop_deadline. The prioritized flows are then passed to modified DBAPS-DM to dictate the order in which it will search for the path candidate with the shortest interference index. (See Algorithm 10)

Algorithm 10 Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (DBAPS-LSP).

Input: The network $N(V, E)$, set of flows F, delay and bandwidth utilization constraints on links $\mathfrak{D}_k = [\mathfrak{D}_k(u,v)]_{\forall(u,v)\in E}$, $\widetilde{\mathfrak{D}}_k = [\widetilde{\mathfrak{D}}_k(u,v)]_{\forall(u,v)\in E}$ and $\mathfrak{B}_k = [\mathfrak{B}_k(u,v)]_{\forall(u,v)\in E}$, $\widetilde{\mathfrak{B}}_k = [\widetilde{\mathfrak{B}}_k(u,v)]_{\forall(u,v)\in E}$, for each flow $f_k \in F$, respectively, and the delay and bandwidth bounds $D_k \in \mathbb{R}^+$ and $\widehat{B}_k \in \mathbb{R}^+$, respectively, and positive constant $X_k \in \mathbb{Z}$, $\forall f_k \in F$.

Output: The path vector $\boldsymbol{\mathcal{P}} = [\mathcal{P}_k]_{\forall f_k \in F}$ where $\mathcal{P}_k$ is the path if the delay and bandwidth constraints ($\mathfrak{D}_k(\mathcal{P}_k) \leq D_k$ and $\mathfrak{B}_k(\mathcal{P}_k) \leq \widehat{B}_k$) are satisfied for f_k, or False otherwise.

```
1: for each f_k ∈ F (starting from higher to lower priority) do
2:     /* Relax bandwidth constraint and solve */
3:     Find the shortest path
4:     if SolutionFound then   /* Path found for f_k */
5:         Use those paths to compute per hop deadline values for each flow
6:         Set flow prios with highest priority assigned to shortest per hop deadline
7:         /* To control the order it visits flows in its search for a shortest interference index path
       to assign in that pass */
8:         Pass the prioritized flow ids to the Algorithm 1
9:         if SolutionFound then
10:            /* Path found by relaxing delay constraint */
11:            P_k := P* /* Add path to the path vector */
12:        else
13:            P_k := False   /* Unable to find any path for f_k */
14:        end if
15:    else
16:        P_k := False   /* Unable to find any shortest path for f_k */
17:    end if
18: end for
```

6.3.3 GDBAPS-LSP Algorithm.

This Algorithm is simpler than DBAPS in that it pays attention to bandwidth , and deadline constraints but is greedy. For each flow, we choose the shortest path available that still has enough bandwidth given prior path assignments. Those paths are then used to compute per hop deadline values for each flow. We then assign the highest flow priority to the shortest per-hop deadline flow. This is described in Algorithm 11.

Algorithm 11 Greedy Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (GDBAPS-LSP).

Input: The network $N(V, E)$, set of flows F, delay and bandwidth utilization constraints on links $\mathfrak{D}_k = [\mathfrak{D}_k(u,v)]_{\forall(u,v)\in E}$, $\widetilde{\mathfrak{D}}_k = [\widetilde{\mathfrak{D}}_k(u,v)]_{\forall(u,v)\in E}$ and $\mathfrak{B}_k = [\mathfrak{B}_k(u,v)]_{\forall(u,v)\in E}$, $\widetilde{\mathfrak{B}}_k = [\widetilde{\mathfrak{B}}_k(u,v)]_{\forall(u,v)\in E}$, for each flow $f_k \in F$, respectively, and the delay and bandwidth bounds $D_k \in \mathbb{R}^+$ and $\widehat{B}_k \in \mathbb{R}^+$, respectively, and positive constant $X_k \in \mathbb{Z}$, $\forall f_k \in F$.

Output: The path vector $\boldsymbol{\mathcal{P}} = [\mathcal{P}_k]_{\forall f_k \in F}$ where $\mathcal{P}_k$ is the path if the delay and bandwidth constraints ($\mathfrak{D}_k(\mathcal{P}_k) \leq D_k$ and $\mathfrak{B}_k(\mathcal{P}_k) \leq \widehat{B}_k$) are satisfied for f_k, or False otherwise.

```
1: for each f_k ∈ F (starting from higher to lower priority) do
2:     /* Abiding bandwidth constraints and solve */
3:     Find the shortest path
4:     if SolutionFound then   /* Path found for f_k */
5:         Use those paths to compute per hop deadline values for each flow
6:         Set flow prios with highest priority assigned to shortest per hop deadline
7:         if SolutionFound then
8:             /* Path found by relaxing delay constraint */
9:             P_k := P* /* Add path to the path vector */
10:        else
11:            P_k := False   /* Unable to find any path for f_k */
12:        end if
13:    else
14:        P_k := False   /* Unable to find any shortest path for f_k */
15:    end if
16: end for
```

6.4 Evaluation

After path assignments are performed using the three approaches (DBAPS-DM,DBAPS-LSP,GDBAPS-LSP), the default Kashinath [31] schedulability logic is used to determine schedulability. We assess performance by computing the percentage of schedulable runs for each approach using a number of parameterizations for network topology and bandwidth.

6.4.1 Simulation Setup

The simulator of Kashinath [31] uses the end-to-End Deadline-Monotonic algorithm (DBAPS-DM) for assigning paths. The algorithm takes into account both bandwidth and deadline constraints by using those measures to compute an interference index for every path that is deemed a candidate for a given flow. The algorithm then searches for the candidate with the lowest interference index and assigns that path to its associated flow. The search order is naive in that it follows the flow id rather than any particular measure that would compel a certain prioritization. Once a path is assigned, the search is repeated until all flows have paths assigned to them.

We modified the simulator to allow us to try two additional path assignment approaches, each applied to the same topology as was used for DBAPS-DM. For each topology, we would try the four approaches to see for each approach, how many would be schedulable versus unschedulable.

The first approach we added was called Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (DBAPS-LSP). In this ap-

proach we first assign paths using an simple algorithm that is naive in that it does not take into consideration bandwidth constraints for priority assignment. Instead it uses the shortest available path for any given flow. It then computes the per-hop deadline for each flow, and assigns flow priorities with the highest priority assigned to the path with shortest per-hop deadline. We then re-assign the paths by using a modified DBAPS-DM that allows the flow priorities to control the order in which candidate paths are considered.

The second approach we added was called Greedy Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (GDBAPS-LSP). In this approach we assign paths using an algorithm uses the shortest path available that abides bandwidth constraints where prior path assignments consume the bandwidth needed by the flow. We use those paths to calculate the per-hop deadline for each flow and assign flow priorities with the highest priority assigned to the path with shortest per-hop deadline. See Figure 6.3.

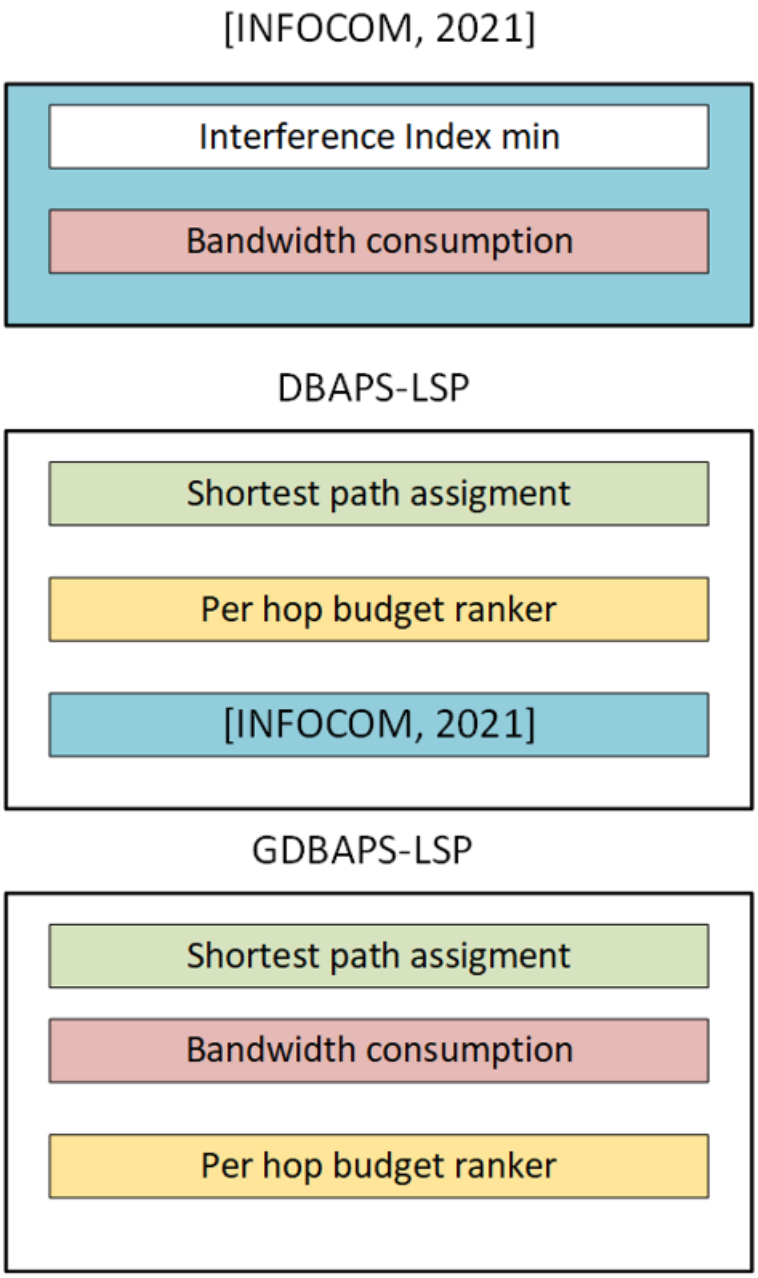

Figure 6.3: Simulation Setup

6.4.2 Result

Our analysis of the different algorithms used reveals that the Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (DBAPS-LSP) algorithm resulted in the most flows scheduled, of the three algorithms tested. Greedy Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (GDBAPS-LSP) was the second best performing algorithm, with Delay and Bandwidth Aware Path Selection with Deadline Monotonic Priorities (DBAPS-DM) being the worst performing. (See Figure 6.4)

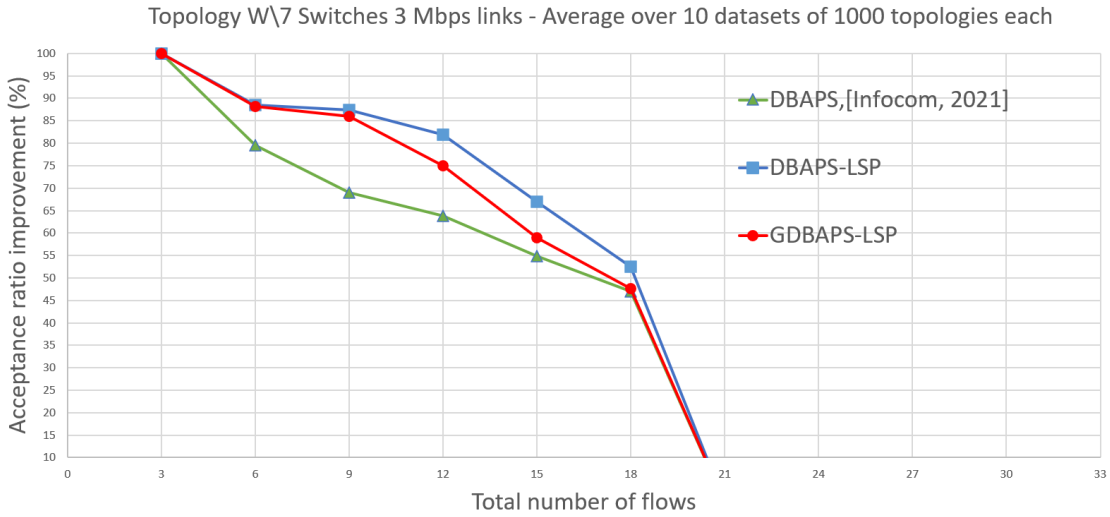

Figure 6.4: 7 Switches 3 Mbps - Average over 10 datasets of 1000 topologies each

Additionally, analysis revealed that the two algorithms DBAPS-LSP and GDBAPS-LSP perform significantly better when the priority (queues) levels used are not fixed at three priorities. We found that adding additional priority levels improved the acceptance ratio in these two algorithms, with DBAPS-LSP having the best perfor-

mance. (See Figure 6.5 and Figure 6.6)

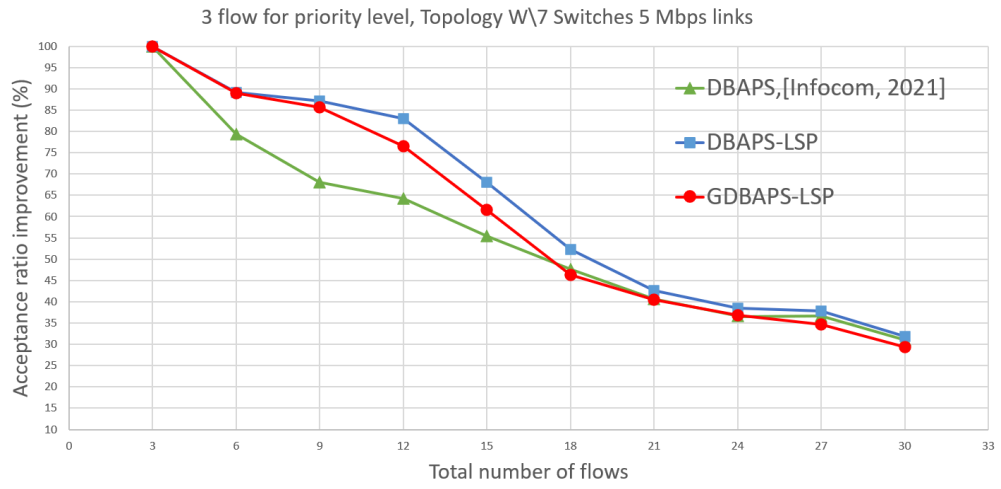

Figure 6.5: Performance of all Algorithms , 3 priority levels , 7 Switches, 5 Mbps

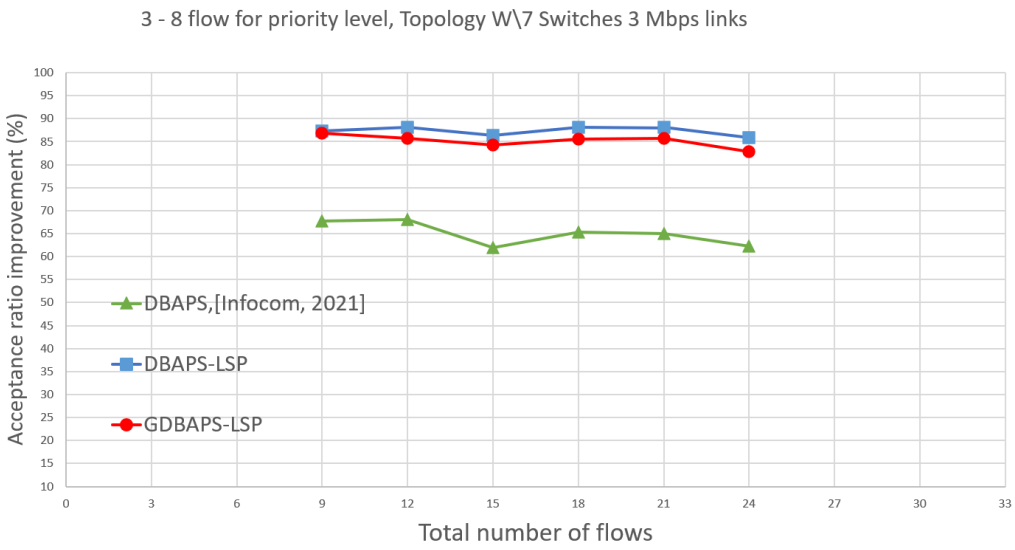

Figure 6.6: Performance of all Algorithms with 3 - 8 level priority 7 Switches, 5 Mbps

The path layout algorithms DBAPD-LSP and GDBAPS-LSP are both more topology-aware than [31] in that they take into account flow hop count. Figure 6.7 reveals that this leverage gives them better performance than [31], by up to 16% in the case of GDBAPS-LSP and up to 18% in the case of DBAPS-LSP. Also rep-

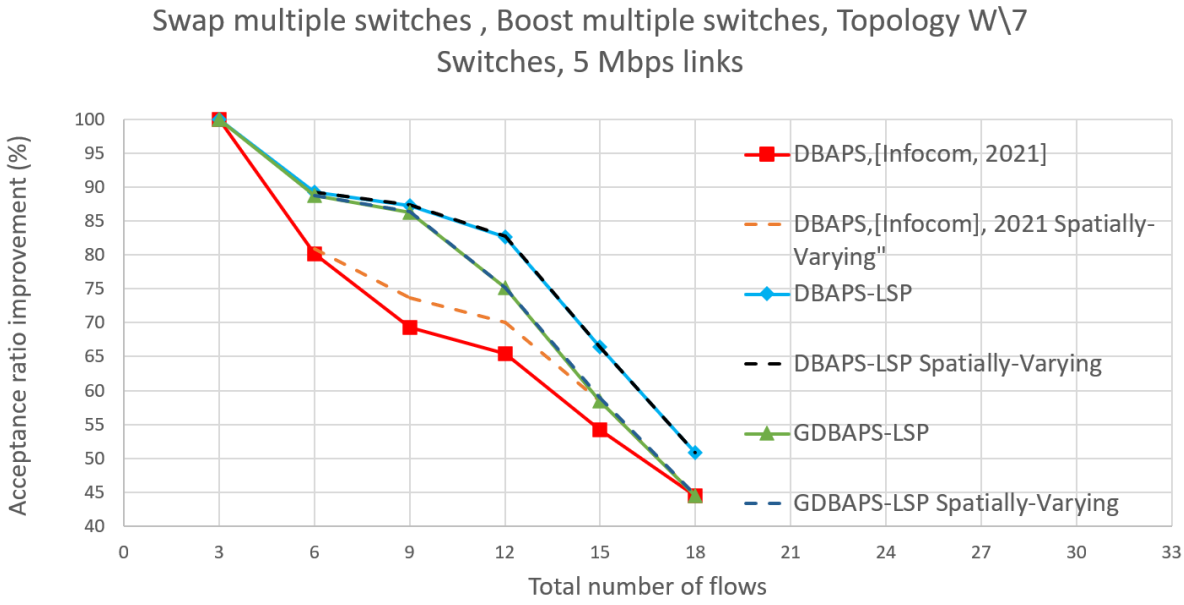

Figure 6.7: Performance of path layout algorithms with Spatially Varying Algorithm applied

resented in Figure 6.7 is the effect of applying the Spatially Varying Priorities as an after-step to the three path assignment approaches to determine if schedulability can be improved. We see that the [31] algorithm is improved upon by up to 5% in acceptance ratio. However, we see that no improvement is made by applying it to either DBAPS-LSP or GDBAPS-LSP. This indicates that the more topology-aware path layout approaches are already making better use of the available slack in the network.

Chapter 7: Future Work

7.1 Local Swap Priority

The most pressing future improvement to this work would be a proper implementation of this system using a modification of a commercial SDN controller [1, 4]. For our evaluation experiment we used Kashinanth et al.'s [31] simulation framework and refactoring by Python3 [2, 3] . It remains to be seen how well our approach could be integrated into a production system using existing technology. Doing so would provide an immediate benefit as a new scheduling system using controller-switch cooperation could be achieved.

An expansion on this work could address a failing flow's path. This work focused on changing the local priorities of a flow as a method of improvement. Future work could address not just the alteration of switch priorities, but could attempt changes to a failed flow's path in order to improve its deadline performance.

Another valuable addition to this research would be a mechanism to mix routing and scheduling together. In our system, we allowed routes to be set by the framework, but in a real significant improvement could be gained if a controller's intelligence were leveraged to route flows along faster paths, dynamically, where required. For example, our implementation was able to track delay for each flow inside each switch, along with the maximum and minimum delay. This information could be leveraged by a well-designed controller. An adaptation of the cost system based on real-time

data might be possible wherein the controller could re-route flows to faster paths based on cost or other factors.

An interesting expansion of this work would be to introduce even more intelligence to queuing priority packets [32]. One approach which has been discussed, but has yet to be properly expanded and considered, is an approach wherein high priority packets are dynamically pushed on to queues with the smallest current size. This might allow for high priority packets to be allocated to those queues where they are most likely to receive quick service.

Lastly, our approach needs to be compared with other approaches to determine its effectiveness and efficiency.

7.2 Dynamic Priority

Qian et al. introduced an algorithm that aids in routing and determining the forwarding paths for real-time message moves in distributed calculating domains while also creating a scheduler that helps in enforcing a message forwarding policy on network devices. In addition to that, this routing algorithm takes into consideration network resource demands of real-time messages and deadlines. Due to this, no results of real-time messages are allowed to be dropped due to the network contention that performs when being controlled by the message scheduler while also no real-time messages unable to meet their deadlines. However, this scheme requires modifications of switch scheduling mechanisms and hence cannot be realized with COTS hardware. Comparatively, our proposal does not require a hardware modification

and can work with COTS SDN switches. One direction is to explore ways to achieve this with COTS hardware.

7.3 Leverage Proposed Switch Capabilities

P4 (Programming Protocol-independent Packet Processors) is a language in development by a wide array of researches in industry and academia. It's primary purpose is to provide a unified programming paradigm in which software can be written to control the data plane directly [12]. The language is as-yet incomplete, but very interesting work is ongoing which will allow for event-based controls like the algorithms we have proposed.

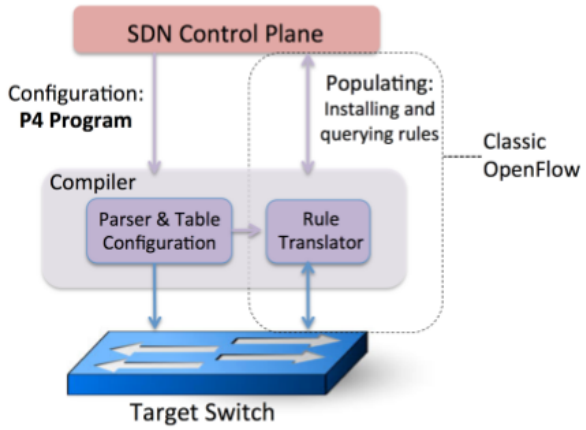

Figure 7.1: A P4-configured switch [12]

Additionally, a framework called Inband Network Telemetry has been developed which could give switches access to the kind of information they would need to make useful decisions about packet prioritization [23].

From these two developments we think the next step in using these new methods to improve the networking landscape - we must work with the P4 language to ensure that the specification of this important new development is flexible and robust enough to handle these efficient new prioritization algorithms.

Chapter 8: Conclusion

8.1 Spatially-Varying Locally Static Priorities

Network flows in Real-Time (RT) systems need to meet stringent end-to-end deadlines for safe and reliable operation of such system.

The historical and simple approach to support this demand has been to increase the number of switches as load increases. However, switches are expensive. Furthermore, large, complex networks are hard to manage due to hardware integration of the control and data plane.

We propose using SDN to achieve the flexibility and dynamism we require. With SDN, the software control layer is decoupled from the data plane. Such software systems are relatively easy to manage, compared to traditional networks. They also provide higher compatibility with a wider range of hardware manufacturers. To solve this problem, there are two approaches Spatially-Varying Locally Static Priorities and Static Priority Assignment

In Spatially Varying Static Priority Scheduling real time traffic flows was improved. We analyze end-to-end deadlines in a real-time system to ensure that all packets reach their destination meeting the static end-to-end deadlines set before the environment is run. To schedule flows, we use a deadline-monotonic scheduling algorithm. In order to better achieve the goal of meeting end-to-end deadlines we contrast a system with static global priorities for flows with one where priorities are

assigned at switches, giving local priorities to flows again based on the deadline-monotonic scheduling algorithm, instead of using [34]. We used heuristic algorithms to improve schedulability for their framework, but leverage our ability to refine the priorities switch-by-switch within each flow. This allows our algorithms to make more surgical improvements.

The analysis of this study reveals that Heuristic 7 (Multiple Switch Swaps and Boost Path) has the best performance among the algorithms - an increase of up to 10.6%. In this algorithm, a priority swap at a multiple switch is combined with a wholesale adjustment of priority at the remaining switches of the failed flow, using the priority value that was originally held by the swap target flow happen in all switches in the path (boost).

8.2 Least-Slack Prioritization

We compared the performance of two path assignment algorithms to the reference algorithm, Delay and Bandwidth Aware Path Selection with Deadline Monotonic Priorities (DBAPS-DM). Varying flow counts and keeping the number of priority levels (i.e. ,queues) constant at 3, our Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (GDBAPS-LSP) algorithm performed up to 17% better than (DBAPS-DM), with the most improvement in the range of 6 to 15 flows in the network. Our Delay and Bandwidth Aware Path Selection with Least-Slack (per-hop budget) Prioritization (DBAPS-LSP) algorithm performed up to 19% better than DBAPS-DM, again in the 6 to 15 flow range.

Taking a second approach of increasing the number of priority queues as flow counts increase (by keeping the number of flows per priority level constant at 3, and varying the priority levels from 3 to 8), we see consistent improvement across all flow counts from 9 to 24. The improvement for GDBAPS-LSP varied from 18% to 23%. The improvement for DBAPS-LSP varied from 20% to 25%. These improvements are significant enough to consider applying these algorithms in the proper contexts.

Finally, we determined that applying the Spatially Varying Priorities Algorithm 8 as an after-step to the three path assignment approaches only improves schedulability in the case of DBAPS [31] , where the improvement seen was up to 5% in acceptance ratio, with the most improvement occurring in the 9 - 15 flow range. No improvement is made by applying it to either DBAPS-LSP or GDBAPS-LSP,as these more topology-aware approaches are already making better use of the available slack in the network.

Bibliography

[1] Grotto Networking basic network simulations and beyond in python. `https://www.grotto-networking.com/DiscreteEventPython.html`.

[2] Networkx python package. `https://networkx.github.io/`.

[3] python3 python documentation. `https://docs.python.org/3/`.

[4] SimPy event discrete simulation for python. `https://simpy.readthedocs.io/en/latest/`.

[5] ARINC Specification 664, Part 7, Aircraft Data Network, Avionics Full Duplex Switched Ethernet (AFDX) Network. 2003.

[6] Namwon An, Taejin Ha, Kyung-Joon Park, and Hyuk Lim. Dynamic priority-adjustment for real-time flows in software-defined networks. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 144–149. IEEE, 2016.

[7] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal*, 8(5):284–292, 1993.

[8] Abdullah Aydeger, Kemal Akkaya, Mehmet H Cintuglu, A Selcuk Uluagac, and Osama Mohammed. Software defined networking for resilient communications in smart grid active distribution networks. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2016.

[9] Haowei Bai, Mohammed Atiquzzaman, and William A Ivancic. Running integrated services over differentiated service networks: quantitative performance measurements. In *ITCom 2002: The Convergence of Information Technologies and Communications*, pages 11–22. International Society for Optics and Photonics, 2002.

[10] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the complexity of network management. In *NSDI*, pages 335–348, 2009.

[11] Guillem Bernat, Alan Burns, and Albert Liamosi. Weakly hard real-time systems. *IEEE transactions on Computers*, 50(4):308–321, 2001.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Computer Communication Review*, 44:87–95, 2014.

[13] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[14] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking (ToN)*, 17(4):1270–1283, 2009.

[15] H. Charara, J. L. Scharbarg, J. Ermont, and C. Fraboul. Methods for bounding end-to-end delays on an AFDX network. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*, pages 10 pp.–202, 2006.

[16] Shigang Chen and Klara Nahrstedt. On finding multi-constrained paths. In *Communications, 1998. ICC 98. Conference Record. 1998 IEEE International Conference on*, volume 2, pages 874–879. IEEE, 1998.

[17] Shigang Chen and Klara Nahrstedt. An overview of quality of service routing for next-generation high-speed networks: problems and solutions. *IEEE network*, 12(6):64–79, 1998.

[18] Tamer Dag and Oral Gokgol. A priority based packet scheduler with deadline considerations. In *4th Annual Communication Networks and Services Research Conference (CNSR'06)*, pages 8–pp. IEEE, 2006.

[19] Hampton C Gabler and John Hinch. Evaluation of advanced air bag deployment algorithm performance using event data recorders. In *Annals of Advances in Automotive Medicine/Annual Scientific Conference*, volume 52, page 175. Association for the Advancement of Automotive Medicine, 2008.

[20] Laurent George, Paul Muhlethaler, and Nicolas Rivierre. Optimality and non-preemptive real-time scheduling revisited. 1995.

[21] Jochen W Guck and Wolfgang Kellerer. Achieving end-to-end real-time quality of service with software defined networking. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 70–76. IEEE, 2014.

[22] Jochen W Guck, Amaury Van Bemten, Martin Reisslein, and Wolfgang Kellerer. Unicast qos routing algorithms for sdn: A comprehensive survey and performance evaluation. *IEEE Communications Surveys & Tutorials*, 20(1):388–415, 2017.

[23] Anton Gulenko, Marcel Wallschläger, and Odej Kao. A practical implementation of in-band network telemetry in open vswitch. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2018.

[24] Shengyan Hong, Thidapat Chantem, and Xiaobo Sharon Hu. Meeting end-to-end deadlines through distributed local deadline assignments. *2011 IEEE 32nd Real-Time Systems Symposium*, pages 183–192, 2011.

[25] Jay Hyman, Aurel A Lazar, and Giovanni Pacifici. Mars: The magnet ii real-time scheduling algorithm. *ACM SIGCOMM Computer Communication Review*, 21(4):285–293, 1991.

[26] National Instruments. Controller Area Network (CAN) Overview.

[27] Jeffrey M Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1):95–116, 1984.

[28] Raj Jain and Subharthi Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[29] P. Jayachandran and T. Abdelzaher. Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *2008 Euromicro Conference on Real-Time Systems*, pages 233–242, 2008.

[30] Praveen Jayachandran and Tarek F. Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Real-Time Systems*, 40:290–320, 2008.

[31] Ashish Kashinath, Monowar Hasan, Rakesh Kumar, Sibin Mohan, Rakesh Bobba, and Smruti Padhy. Safety critical networks using commodity sdns. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2021.

[32] Daniel I. Katcher, Shirish S. Sathaye, and Jay K. Strosnider. Fixed priority scheduling with limited priority levels. *IEEE Transactions on Computers*, 44(9):1140–1144, 1995.

[33] Deepak Kumar, Joy Kuri, and Anurag Kumar. Routing guaranteed bandwidth virtual paths with simultaneous maximization of additional flows. In *IEEE International Conference on Communications, 2003. ICC'03.*, volume 3, pages 1759–1764. IEEE, 2003.

[34] Rakesh Kumar, Monowar Hasan, Smruti Padhy, Konstantin Evchenko, Lavanya Piramanayagam, Sibin Mohan, and Rakesh B Bobba. End-to-end network delay guarantees for real-time systems using sdn. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–242. IEEE, 2017.

[35] Ian Land and Jeff Elliott. Architecting arinc 664, part 7 (afdx) solutions. XILINX, 2009.

[36] Kilho Lee, Minsu Kim, Hayeon Kim, Hoon Sung Chwa, Jinkyu Lee, and Insik Shin. Fault-resilient real-time communication using software-defined networking. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 204–215. IEEE, 2019.

[37] Dan Levin, Marco Canini, Stefan Schmid, and Anja Feldmann. Incremental sdn deployment in enterprise networks. *ACM SIGCOMM Computer Communication Review*, 43(4):473–474, 2013.

[38] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[39] M Migliore, V Martorana, and F Sciortino. An algorithm to find all paths between two nodes in a graph. *Journal of Computational Physics*, 87(1):231–236, 1990.

[40] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 501–521, 2016.

[41] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Mps-can analyzer: Integrated implementation of response-time analyses for controller area network. *Journal of Systems architecture*, 60(10):828–841, 2014.

[42] Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Integrating mixed transmission and practical limitations with the worst-case response-time analysis for controller area network. *Journal of Systems and Software*, 99:66–84, 2015.

[43] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[44] Moonju Park. Non-preemptive fixed priority scheduling of hard real-time periodic tasks. In *Proceedings of the 7th International Conference on Computational Science, Part IV: ICCS 2007*, ICCS '07, pages 881–888, Berlin, Heidelberg, 2007. Springer-Verlag.

[45] Jon M Peha and Fouad A Tobagi. Cost-based scheduling and dropping algorithms to support integrated services. *IEEE Transactions on Communications*, 44(2):192–202, 1996.

[46] Thomas Pfeiffenberger and Jia Lei Du. Evaluation of software-defined networking for power systems. In *2014 IEEE International Conference on Intelligent Energy and Power Systems (IEPS)*, pages 181–185. IEEE, 2014.

[47] Tao Qian, Frank Mueller, and Yufeng Xin. Hybrid edf packet scheduling for real-time distributed systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 37–46. IEEE, 2015.

[48] Tao Qian, Frank Mueller, and Yufeng Xin. A linux real-time packet scheduler for reliable static sdn routing. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[49] Ahmed Rahni, Emmanuel Grolleau, and Michael Richard. Feasibility analysis of non-concrete real-time transactions with edf assignment priority. 2008.

[50] Myriana Rifai, Dino Lopez-Pacheco, and Guillaume Urvoy-Keller. Coarse-grained scheduling with software-defined networking switches. *ACM SIGCOMM Computer Communication Review*, 45(4):95–96, 2015.

[51] Mehrin Rouhifar and Reza Ravanmehr. A survey on scheduling approaches for hard real-time systems. 2015.

[52] Fragiskos Sardis, Massimo Condoluci, Toktam Mahmoodi, and Mischa Dohler. Can qos be dynamically manipulated using end-device initialization? In *Communications Workshops (ICC), 2016 IEEE International Conference on*, pages 448–454. IEEE, 2016.

[53] Robert Sedgewick. *Algorithms in c, part 5: graph algorithms*. Pearson Education, 2001.

[54] Jon Spencer, Olwen Worthington, Robert Hancock, and Eleanor Hepworth. Towards a tactical software defined network. In *2016 International Conference on Military Communications and Information Systems (ICMCIS)*, pages 1–7. IEEE, 2016.

[55] Marco Spuri. Analysis of deadline scheduled real-time systems. 1996.

[56] Marco Spuri. Holistic analysis for deadline scheduled real-time distributed systems. 1996.

[57] Lars-Erik Thorelli. An algorithm for computing all paths in a graph. *BIT Numerical Mathematics*, 6(4):347–349, 1966.

[58] Ken Tindell, H Hanssmon, and Andy J Wellings. Analysing real-time communications: Controller area network (can). In *RTSS*, pages 259–263. Citeseer, 1994.

[59] Chenhui Xu, Bing Chen, and Hongyan Qian. Quality of service guaranteed resource management dynamically in software defined network. *Journal of Communications*, 10(11):843–850, 2015.

[60] Shu-Ngai Yeung and John Lehoczky. End-to-end delay analysis for real-time networks. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 299–309. IEEE, 2001.

[61] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. Rsvp: A new resource reservation protocol. *Netwrk. Mag. of Global Internetwkg.*, 7(5):8–18, September 1993.

[62] Xiaoyuan Zhu, Hui Zhang, Dongpu Cao, and Zongde Fang. Robust control of integrated motor-transmission powertrain system over controller area network for automotive applications. *Mechanical Systems and Signal Processing*, 58:15–28, 2015.