Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Tracing-based Scheduling of Isochronous Traffic in Time-Sensitive Networks

Alexander Glavackij

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	M.Sc. David Hellmanns
Commenced:	February 19, 2020
Completed:	August 19, 2020

Abstract

Time-Sensitive Networking (TSN) is a set of standards defined by the IEEE 802.1 TSN Task Group and aims at making IEEE 802.3 Ethernet real-time capable. TSN employs a Time Division Multiple Access (TDMA) scheme to achieve temporal isolation between data streams, and thus, provides bounds on worst-case latencies and real-time guarantees. The TDMA scheme divides access to transmission links into slots. Calculating the slots of the TDMA scheme is referred to as scheduling. Scheduling is known to be NP-complete. Current approaches are mainly based on constraint programming and optimization problems, and thus, do not scale well for large problem instances. In previous work *Scaling TSN Scheduling for Factory Automation Networks* [HGF+20], Hellmanns et al. developed a novel fast scheduling approach to schedule data streams in a TSN network. It significantly outperformed current approaches in execution time. They use a discrete-event network simulator to simulate stream transmissions and to derive a schedule from the simulation results, sacrificing the optimality of the schedule for calculation speed. The authors refer to this approach as *tracing-based scheduling*. We expand the tracing-based scheduling approach to a full-blown scheduling algorithm. To do that, we propose a novel formulation of the scheduling problem: scheduling as a search problem. Using this formulation, we develop four novel scheduling approaches: Tracing-based Scheduling (TBS) Naive approach, TBS Link Time Remaining Time, TBS Link Time Late Streams, and TBS Link Time Monte Carlo Tree Search (MCTS). We evaluate each approach along the following properties: execution rates, scheduling runtimes, and scheduling capabilities. The MCTS approach exhibits the best performance. It outperforms two contemporary schedulers and can schedule the majority of problem instances in under 1,200 s.

Kurzfassung

Time-Sensitive Networking (TSN) ist eine Sammlung von Standards, die von der IEEE 802.1 TSN Task Group definiert wurde und IEEE 802.3 Ethernet echtzeitfähig macht. TSN benutzt ein Zeitmultiplexverfahren (TDMA) um Datenverkehr zu isolieren und somit Echtzeit Garantien geben zu können. Das TDMA Verfahren unterteilt den Zugang zu Übertragungsverbindungen in Zeitschlitze. Die Berechnung dieser Zeitschlitze wird im sogenannten Scheduling durchgeführt. Das Scheduling ist NP-vollständig. Aktuelle Ansätze basieren auf Constraint Programmierung und Optimierungsproblemen und skalieren daher nicht für größere Problem Instanzen. In einer vorausgegangenen Arbeit *Scaling TSN Scheduling for Factory Automation Networks* [HGF+20] entwickelten Hellmanns et al. ein neues Scheduling Verfahren. Das Verfahren benutzt eine diskrete Ereignissimulation um Datenverkehr zu simulieren. Der Schedule wird von den Simulationsergebnissen abgeleitet. Diese Vorgehensweise beschleunigt die Berechnung, der berechnete Schedule ist jedoch nicht zwangsläufig optimal. Die Autoren nennen dieses Verfahren *tracing-based scheduling*. Wir erweitern das tracing-based scheduling zu einem vollständigen Scheduling Algorithmus. Dafür schlagen wir eine neue Formulierung des Scheduling Problems vor: Scheduling als Suchproblem. Mit unserer Formulierung entwickeln wir vier neue Scheduling Algorithmen: Tracing-based Scheduling (TBS) Naive; TBS Link Time Remaining Time; TBS Link Time Late Streams and TBS Link Time Monte Carlo Tree Search (MCTS). Der MCTS Ansatz zeigt die besten Scheduling Fähigkeiten und ist in der Lage, die meisten Problem Instanzen in unter 1,200 s zu lösen.

Contents

1 Introduction **15**

2 Background **17**
2.1 Ethernet 17
2.2 Time-Sensitive Networking (TSN) 20
2.3 Discrete Event Simulation 25
2.4 Tree Search 26

3 Related Work **31**
3.1 Complete Scheduling Approaches 31
3.2 Heuristic Scheduling Approaches 33

4 System Model **37**
4.1 Network Model 37
4.2 Switch Model 38
4.3 Host Model 39
4.4 Delay Model 39
4.5 Traffic Model 40

5 Scheduling Approaches **43**
5.1 Scheduling as a Search Problem 43
5.2 Designing Scheduling Approaches 44

6 Evaluation **53**
6.1 Evaluation Setup 53
6.2 Results 59

7 Conclusion and Outlook **73**

Bibliography **75**

List of Figures

2.1 Format of an Ethernet packet [IEE18a]. 17
2.2 Architecture of Switched Ethernet [Tan81]. 18
2.3 High-view architecture of a switch. 19
2.4 Properties of different data transmission services [Fin18]. 21
2.5 A sequence of transmissions on a single network link. Each packet is transmitted in its dedicated slot, during these slots no other traffic can be transmitted. Isolating packets prevents interference amongst them. 22
2.6 Egress Port architecture of a TSN compliant switch [IEE18b]. 23
2.7 A basic tree with node ids as node names. 26
2.8 One iteration of the MCTS process [BPW+12]. 28

4.1 Graph showing an example factory layout. Backbone nodes are marked with B, Local Area Network (LAN) nodes are marked with L. 38
4.2 Sequence diagram showing the delays a packet encounters on its path. 39

5.1 Visualization of Definition 5.1.3. 44
5.2 Chain resulting from the search with the Naive approach. 46
5.3 Visualizing Definition 5.2.2. 47
5.4 Stream s arrives at its destination after its deadline. Consequently, its remaining time is negative. 48
5.5 Tree visualizing the search process of the Link Time Remaining Time approach. Nodes are numbered by their expansion order. 49
5.6 Tree visualizing the search process of the Link Time Late Stream approach. ls denotes the number of late streams in a stream set. 50

6.1 Activity diagram showing the pipeline structure and the corresponding stages of the benchmarking framework [SWHD20]. The *evaluation of results* stage is not depicted, since it is done manually, and therefore, not part of the activity diagram. 54
6.2 Activity diagram showing the procedure *feasible generator* that generates feasible stream sets. 55
6.3 Graph showing the execution rates for all evaluated schedulers. The TSN SMT and JSSP schedulers executed scheduling for smaller stream set sizes only. The TBS schedulers (except for the MCTS approach) show low execution rates, regardless of the input configuration. Only the MCTS approach did execute all problem instances. 60
6.4 Boxplots showing the runtime distribution for the evaluated schedulers, grouped by the stream set sizes. 62
6.5 Box-plots showing the runtime distribution for the evaluated schedulers, grouped by the size of the topologies. 63
6.6 Graphs showing the ratio of solved stream sets over scheduling time. Note the different x-axis scaling. 64

6.7 Search tree generated by the Link Time Remaining Time approach. The node labels are the rts values defined in Section 5.2.2. Before the search can backtrack, it has to expand every child node, to check if there exists one with a better rts value. As a result, the search can get stuck expanding a large number of child nodes. 66
6.8 Comparing the search trees generated by the Tracing-based Scheduler (TBS) Link Time Remaining Time Scheduler and the TBS Link Time MCTS Scheduler. The MCTS approach builds the search tree asymmetrically, while the Link Time Remaining Time approach expands a single node. 67
6.9 Ratio of solved stream sets for the TBS Link Time Remaining Time scheduler. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$. 68
6.10 Ratio of solved stream sets for the TBS Link Time Late Streams scheduler. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$. 68
6.11 Ratio of solved stream sets for the TBS Link Time MCTS scheduler. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$. 69
6.12 Influence of stream set size on simulation runtime for the *tracing-based simulator*. The black error bars indicate the 90% confidence interval of 20 simulation runs for each stream set size. 70
6.13 Ratio of solved stream sets for the TBS Link Time MCTS scheduler with a 14300 s runtime limit. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$. . . 71

List of Tables

2.1 Priority to traffic class mapping suggested by IEEE Std 802.1Q [IEE18b]. 24
2.2 Table displaying the properties of DFS and BFS search strategies [RN09]. *d* denotes the depth of the least-cost solution. 27

6.1 This table shows the parameters we use for generating benchmark topologies. . . 57
6.2 This table shows the parameters we use for generating stream sets. 58
6.3 Hardware of the evaluation cluster nodes. 59
6.4 Hardware of the evaluation cluster node. 71

List of Algorithms

2.1 Basic tree search algorithm in pseudo-code [RN09]. 26
2.2 Basic MCTS procedure in pseudo-code [BPW+12]. 29

Acronyms

AVB Audio Video Bridging. 22

BFS Breadth-first Search. 27

DEI Drop Eligible Indicator. 18

DES Discrete Event Simulation. 25

DFS Depth-first Search. 27

FCS Frame Check Sequence. 18

GCL Gate Control List. 23

IEEE Institute of Electrical and Electronics Engineers. 15

IIC Industrial Internet Consortium. 40

ILP Integer Linear Problem. 31

IPG Interpacket Gap. 18

IRT Isochronus Real Time. 32

JSSP Job Shop Scheduling Problem. 53

LAN Local Area Network. 7

MAC Medium Access Control. 17

MCTS Monte Carlo Tree Search. 28

PCP Priority Code Point. 18

QoS Quality of Service. 20

SFD Start Frame Delimiter. 17

SMT Satisfiability Modulo Theories. 31

TAS Time-Aware Shaper. 22

TBS Tracing-based Scheduler. 8

TDMA Time Division Multiple Access. 15

TPID Tag Protocol Identifier. 18

TSA Transmission Selection Algorithm. 23

TSN Time-Sensitive Networking. 15

UCB1 Upper Confidence Bound 1. 50

UCT Upper Confidence Bound Tree. 50

VID VLAN Identifier. 18

1 Introduction

Cyber-physical systems (CPS) have become paramount in today's world. CPS generally consist of sensors, controllers, and actuators, which form a computer network. Specific properties can classify network traffic, i.e., the requirements on arrival times of messages or the temporal distribution of messages. Safety-critical alarms are an example of a traffic class with tight upper bounds on latency. At the same, networks often need to carry traffic with no such demands, e.g., configuration messages. We refer to these classes of network traffic as criticality classes. Traditionally, networks carried a single criticality class. As a result, each criticality class requires its dedicated network when transmission of multiple criticality classes is desired. Traffic with strict temporal requirements is referred to as time-sensitive traffic. Nowadays, there is a trend towards converging networks carrying different criticality classes into a single network, creating converged networks.

An example of a converged network is a factory network. Factory networks carry office traffic, which mostly consists of web-traffic and data-traffic. At the same time, they need to carry traffic used to feed control loops, critical to the operation of the plant or machines. In the worst cases, the late arrival of time-sensitive traffic might lead to instability, inaccuracies, or failure. Therefore, it is of utmost importance to meet the temporal requirements of time-sensitive traffic.

In the IT sector, IEEE Ethernet is the de facto standard for Layers 1 and 2 of the OSI Layer model; however, it cannot transmit traffic in real-time. Proprietary extensions to the Ethernet standard adding real-time capabilities, such as PROFINET [TV99], SERCOS [Sch04], and TTEthernet [TTE] already exist. However, these technologies are not interoperable with each other, leading to communication silos. Communication silos impede the exchange of data, and therefore, are a significant roadblock on the way to smart factories. The Institute of Electrical and Electronics Engineers (IEEE), the standardization organization behind Ethernet, set out to develop additional standards, extending Ethernet to support real-time guarantees. These standards, as a whole referred to as Time-Sensitive Networking (TSN), focus on meeting the real-time constraints for time-sensitive traffic. Since Ethernet enjoys widespread acceptance and is implemented on nearly every device, manufacturers are inclined to implement TSN standards. TSN, and thereby real-time capable Ethernet, would be made available for everyone, encouraging interoperability. This makes TSN very relevant for contemporary research since it has the potential to become the standard for real-time networks.

TSN enables IEEE Ethernet networks to carry multiple traffic classes simultaneously, which differ in various characteristics, such as in priority and schedulability. Time-triggered traffic is the traffic class with the highest priority and strict deadlines on the departure and arrival times of messages. These deadlines are required in applications where timing is critical, e.g., motion control of robots. Temporal isolation of data streams is often needed to prevent interference between streams and to adhere to these deadlines. TSN employs a Time Division Multiple Access (TDMA) scheme to achieve temporal isolation. The TDMA scheme divides access to transmission links into slots. During these slots, it ensures that only a single data packet is eligible for transmission, eliminating

contention between data packets. The time slots, were packets can be transmitted, have to be calculated and reserved for each packet individually. The TDMA scheme requires network devices to be time-synchronized to ensure that packets are assigned to their correct transmission slot, i.e., are on the correct device at the correct time.

The calculation and reservation of time slots, such that temporal requirements are met, is called Time-Aware Shaping or scheduling in short. Optimal scheduling is an NP-hard problem [Ull75]. Thus, the calculation of optimal schedules for large amounts of traffic becomes time-consuming and impractical. Hellmanns et al. [HGF+20] proposed a novel approach to schedule large amounts of traffic: using a network simulator, they simulated traffic streams of a given amount of traffic. They derive the schedule for a specific stream from the simulation results. The authors refer to this approach as tracing-based scheduling. This approach proved to be fast; however, no procedures exist if the simulation shows that the streams do not meet their deadline requirements.

The goal of this Bachelor Thesis is to extend tracing-based scheduling by exploring possibilities to handle violations of stream requirements. We develop procedures to treat stream violations and modify the schedule such that streams meet their requirements. Additionally, we verify these procedures in a network simulator. To do this, we first present the technical background of this work. Afterward, we show related research to this thesis. Having presented the technical background and contemporary research, we present our system model. Next, we illustrate our conceptional work and present multiple scheduling approaches. In Chapter 6, we evaluate them by comparing their capabilities to existing scheduling algorithms. Lastly, we conclude our work and give a brief outlook.

2 Background

In this chapter, we delve into the technical background of this work. As mentioned in Chapter 1, Ethernet is the de-facto standard for computer networks. Therefore, we first describe Ethernet, a protocol used to connect several machines into Local Area Networks (LANs). Second, after establishing the necessary background in Ethernet, we present TSN, the standards which aim at making Ethernet real-time capable. Third, we explain how Discrete-Event Simulation works, since the tracing-based scheduling approach uses a discrete event simulator. Fourth, we illustrate heuristic and probabilistic search methods, as our scheduling approaches utilize both.

2.1 Ethernet

This section presents the basics of Ethernet and is based on Tanenbaum's Computer Networks [Tan81]. This work deals with time-sensitive networks, which are an extension of Ethernet networks. Therefore, we first describe what Ethernet is and which duties it performs. Second, we describe the architecture of Ethernet networks and network devices. Lastly, with the previously given background on Ethernet, we present why Ethernet is incapable of providing upper bounds on transmission times and, therefore, cannot provide real-time guarantees.

Ethernet is a protocol standardized and maintained as IEEE 802.3 [IEE18a] by the IEEE. It resides in Layer 1 and Layer 2 of the OSI Layer Reference Model [DZ83] and enjoys worldwide adoption in computer networks. As a Layer 2 protocol, it is responsible for connecting adjacent machines into a LAN and transmitting data between connected machines. It supplies the upper layers with robust and efficient transmission over physical connections with different properties. To do that, it receives packets from the upper layer and encapsulates them into variable-sized chunks, called Ethernet packets. An Ethernet packet consists of a header, a variable-sized payload, and a trailer, cf. Figure 2.1. In the following, we describe each component of the Ethernet packet.

Preamble: The Preamble allows the physical circuitry to reach a steady-state before transmitting the packet.

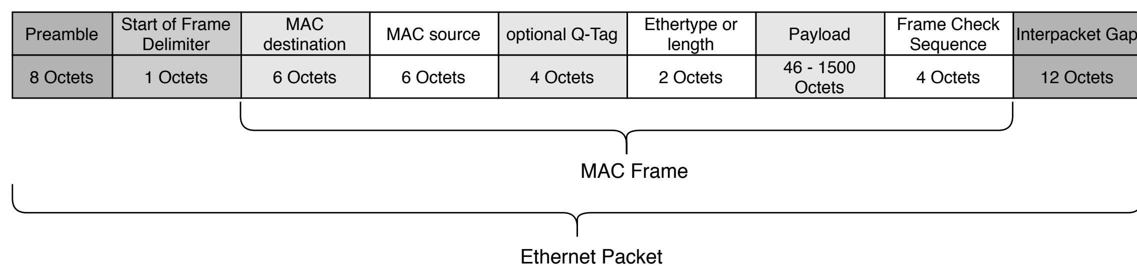


Figure 2.1: Format of an Ethernet packet [IEE18a].

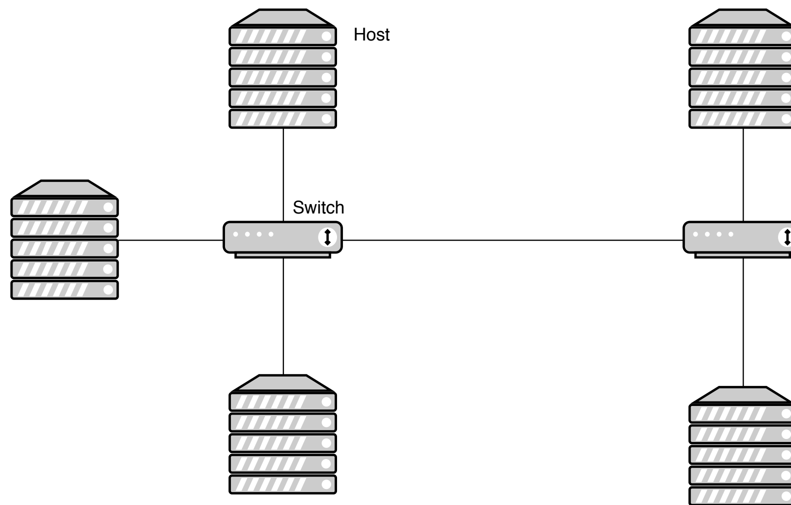


Figure 2.2: Architecture of Switched Ethernet [Tan81].

Start Frame Delimiter (SFD): The SFD is a specific bit-sequence following the Preamble. It denotes the start of the Medium Access Control (MAC) frame.

MAC destination: This field specifies the MAC address of the frame's destination.

MAC source: The source address identifies where the transmission of the frame was initiated.

optional Q-Tag: This is an optional tag added by the 802.1Q [IEE18b] standard. It consists of a Tag Protocol Identifier (TPID), Drop Eligible Indicator (DEI), Priority Code Point (PCP) and VLAN Identifier (VID). Of relevance for this work is the PCP field: it maps packets to traffic classes with different priorities. Packets with higher priority are transmitted before packets with lower priority.

Ethertype or length: This field can take one of two meanings. Depending on its value, it either denotes the length of the payload field or the protocol of the upper layer.

Payload: Payload contains the actual data to be transmitted. The maximum payload size is 1500 Bytes [IEE18a].

Frame Check Sequence (FCS): This field contains a checksum used by the receiver to identify errors during transmission.

Interpacket Gap (IPG): This is a delay added between the transmission of Ethernet packets to provide recovery time for the physical layer below.

As a Layer 1 and Layer 2 protocol, Ethernet controls access to the physical medium transmitting the data. Ethernet can be partitioned by how it controls access to the physical medium: *Classic Ethernet* and *Switched Ethernet* [Tan81]. Classic Ethernet is not relevant in today's networks, since it limits the size of LANs (the length of links is limited to 2500 meters according to 802.3 [IEE18a]). Therefore, we present Switched Ethernet as the relevant LAN configuration.

Switched Ethernet: Figure 2.2 shows the architecture of a Switched Ethernet LAN. Switched Ethernet LANs contain two types of devices: Ethernet Hosts and Layer 2 connecting devices called *Switches*. Devices connect to a switch via ports. A host can send a frame to another host using

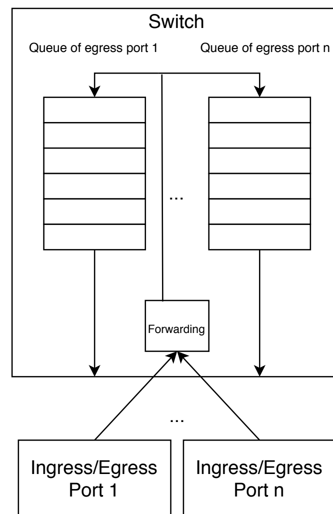


Figure 2.3: High-view architecture of a switch.

the receiving MAC address as the destination MAC address in the packet's header. When a packet arrives at a port on a switch, the switch reads the destination address and forwards the packet to the correct next hop.

Figure 2.3 shows a high-view architecture of a switch. An Ethernet switch consists of the following components: ingress ports, forwarding component, buffer queues, and egress ports. We detail each component in this paragraph. Incoming frames arrive at ingress ports. Afterward, the forwarding element processes incoming frames. The forwarding component reads the frame's header. Based on the destination MAC address, it determines the next hop and forwards the frame to the correct buffer queue. The time it takes the switch to process an incoming packet and make the forwarding decision is referred to as *processing delay*. A switch needs to buffer frames, in case more frames arrive on ingress ports than can be sent out. Queues act as buffers for frames. If a queue is full, the switch does not queue incoming frames and drops them. Egress ports send out frames queued in the buffers. Each egress port has its own queue.

A switch can support two different types of forwarding: *store & forward* or *cut-through*. A switch operating in store & forward mode waits until it receives a packet entirely before it forwards it. In cut-through mode, a switch forwards an incoming packet immediately after its header was received entirely. If the corresponding egress port is busy, the packet is put into the buffer until the egress port is available. Cut-through is a technology not yet standardized; only technical directives exist [Spe20]. Contemporary switches operate in store & forward mode.

If a sender systematically sends out more packets than the directly connected receiver can process, the receiver starts to drop packets. Furthermore, in Switched Ethernet, packets can pass multiple hops before they arrive at their destination. If the network is congested, every hop on the packet's path is a possible source for delay - the packet can even be dropped along its path. A sending Ethernet node does not have a global view of the network congestion since it is only connected to its adjacent nodes. Therefore, no upper bounds on packet delay can be determined - it is possible that a packet never arrives at its destination. As a result, Ethernet cannot give temporal guarantees on data transmission.

2.2 Time-Sensitive Networking (TSN)

Having introduced the basics of Ethernet, we now turn to Time-Sensitive Networking (TSN), since this work investigates TSN networks. We first introduce TSN and describe what TSN is. Afterward, we present the standards that comprise TSN. IEEE 802.1Q introduces Time-Aware Shaping (TAS), the relevant mechanism for this work. Therefore, we go into more detail on time-aware shaping and describe how it functions. TSN networks are comprised of special TSN devices. In contrast to present Ethernet devices, TSN devices implement a special architecture, in order to provide TSN functionalities. Therefore, we lastly illustrate the architecture of TSN devices.

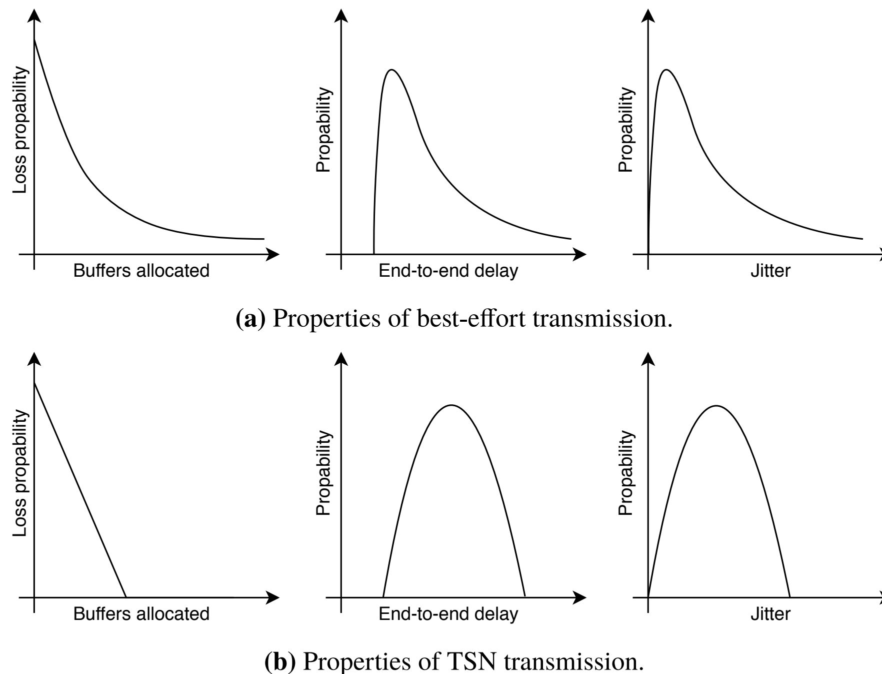
2.2.1 Time-Sensitive Networking Overview

TSN is a set of standards set forth by the IEEE TSN Task Group [IEE20b]. TSN makes it possible to carry data traffic of various applications over an IEEE Ethernet network, while honoring Quality of Service (QoS) requirements [FBG18]. Examples for such applications with different QoS requirements are time-critical applications transmitting TSN traffic and non-time-critical applications transmitting best-effort traffic. Standard Ethernet transmits traffic according to the best-effort paradigm. Figure 2.4 shows the transmission properties of TSN traffic and best-effort traffic [Fin18]. We see that end-to-end latency is unbounded in best-effort transmission. Additionally, packets can be lost, and jitter is unbounded as well. In contrast, TSN provides deterministic data transport of time-sensitive traffic, i.e., guaranteed packet transmission with bounds on latency, low jitter, and low packet loss. TSN achieves this by reserving resources for time-sensitive traffic and applying different traffic shaping techniques. As a result, TSN limits the transmitter of a TSN stream to a set bandwidth. The deterministic data transmission provided by TSN makes TSN applicable in industries where reliable and time-sensitive data transmission is of utmost importance, e.g., in industrial automation. In the following section, we give a brief overview of the standards and services provided by TSN. Afterward, we go into more detail about time-aware shaping, the relevant feature for this work.

2.2.2 Time-Sensitive Networking Standards

The set of standards aggregated by the TSN Task Group comprises the following standards [IEE20b]: IEEE Std 802.1Q-2018, IEEE Std 802.1AS-2020, IEEE Std 802.1AB-2016, IEEE Std 802.1AX-2020, IEEE Std 802.1CB-2017, IEEE Std 802.1BA-2011 and IEEE Std 802.1CM-2018. We now provide a brief overview of these standards and their functionalities in the given order.

IEEE Std 802.1Q-2018 [IEE18b]: IEEE 802.1Q-2018 specifies the operation of switches and switched networks. It includes the principles of operation for switches and maintenance of QoS requirements. It identifies the functions performed by switches and provides an architectural model of the operation of a switch. This standard defines protocols within the architecture of switches that provide capabilities to detect and verify connectivity failures in switched networks. Additionally, 802.1Q-2018 allows switches to provide performance guarantees for time-sensitive and loss-sensitive data transmission. Historically, the TSN Task Group introduced most TSN functionalities as amendments to the 802.-1Q standard. However, these amendments have since been merged into it. The most significant amendment for this work is IEEE 802.1Qbv.



(a) Properties of best-effort transmission.

(b) Properties of TSN transmission.

Figure 2.4: Properties of different data transmission services [Fin18].

IEEE Std 802.1Qbv [IEE16b]: IEEE 802.1-Qbv is an amendment to the IEEE 802.1-Q standard. It provides enhancements to the forwarding process. These enhancements allow transmissions from TSN devices to be scheduled relative to a global network time; hence, the requirement for TSN devices to be synchronized. Scheduling transmissions is referred to as Time-Aware Shaping (TAS).

IEEE Std 802.1AS-2020 [IEE11a]: Ethernet devices do not have a concept of global time. Therefore, clocks of individual devices do not run in sync. Clocks can be out of phase, i.e., not synchronized, and progress at different rates, i.e., they are not syntonized. However, to meet the requirements of time-sensitive traffic, a common time reference for all device clocks is required. This standard provides protocols and procedures to ensure that the data streams meet time-sensitive applications' timing requirements. IEEE Std 802.1AS-2020 leverages the Precision Time Protocol (PTP) defined in IEEE 1588 [IEE08] to synchronize TSN devices.

IEEE Std 802.1AB-2016 [IEE16a]: This standard defines protocols suitable for advertising information to devices attached to the same IEEE Ethernet LAN. It allows network devices to exchange connectivity and management information and to advertise their capabilities.

IEEE Std 802.1AX-2020 [IEE20a]: This standard provides protocols and procedures that allow parallel links to be aggregated into a Link Aggregate Group. The Link Aggregate Group is then treated as if it were a single link. Furthermore, 802.1AX defines procedures to distribute frames over links of a Link Aggregate Group. Aggregate links have a higher aggregate bandwidth than the individual links in the Link Aggregation Group. This leads to improved utilization of available links and improved resilience to failures of individual links.

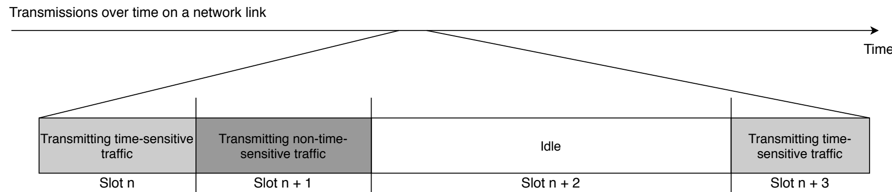


Figure 2.5: A sequence of transmissions on a single network link. Each packet is transmitted in its dedicated slot, during these slots no other traffic can be transmitted. Isolating packets prevents interference amongst them.

IEEE Std 802.1CB-2017 [IEE17]: 802.1CB defines procedures and protocols for network device identification and replication of data traffic for redundant transmission, identification of duplicate traffic, and their elimination. The motivation for duplicating traffic is to increase the probability of successful delivery.

IEEE Std 802.1BA-2011 [IEE11b]: The transmission of audio and video data poses unique demands on networks. This standard defines features, options, configurations, protocols, and procedures of bridges, stations, and LANs to build networks capable of transporting time-sensitive audio and video data traffic. This standard selects features from the IEEE Std 802.1Q and the IEEE Std 802.1AS standards and creates profiles that manufacturers of LAN equipment can use to develop Audio Video Bridging (AVB)-compatible LAN components.

IEEE Std 802.1CM-2018 [IEE18c]: This standard specifies profiles that select features, options, protocols, and procedures of LAN devices to build networks that transport time-sensitive fronthaul streams. Fronthaul streams provide connectivity between cellular base stations. These streams have demanding QoS requirements. Suppliers can use the profiles defined by 802.1CM to produce compliant equipment.

2.2.3 Scheduling and Time-Aware Shaping

IEEE 802.1Qbv introduced Time-Aware Shaping (TAS). Time-Aware Shaper (TAS) as the relevant TSN mechanism for this work. We explain TAS in the subsequent section.

As mentioned previously, the timely arrival of time-sensitive traffic is critical for the correct operation of time-sensitive applications. One approach could be to prioritize time-sensitive traffic over all other traffic classes. However, the prioritization approach causes problems: a time-sensitive frame can still be delayed by a lower priority frame. The time-sensitive frame has to queue until the low priority frame is fully transmitted before it can be transmitted itself. If these delays accumulate over several hops, the latency could ultimately be too large. Furthermore, some applications could require the transmission of multiple separate time-sensitive traffic classes. The prioritization approach does not lend itself to these applications.

The IEEE 802.1Qbv standard utilizes another approach. It employs a TDMA scheme on every transmission link in a TSN network. The TDMA scheme ensures that at specific times, only a single traffic class has access to a transmission link, thereby eliminating interference from other traffic classes. In essence, it creates a protected channel that is used by time-sensitive traffic alone. The TDMA scheme divides access to the transmission links into slots, cf. Figure 2.5. During one slot,

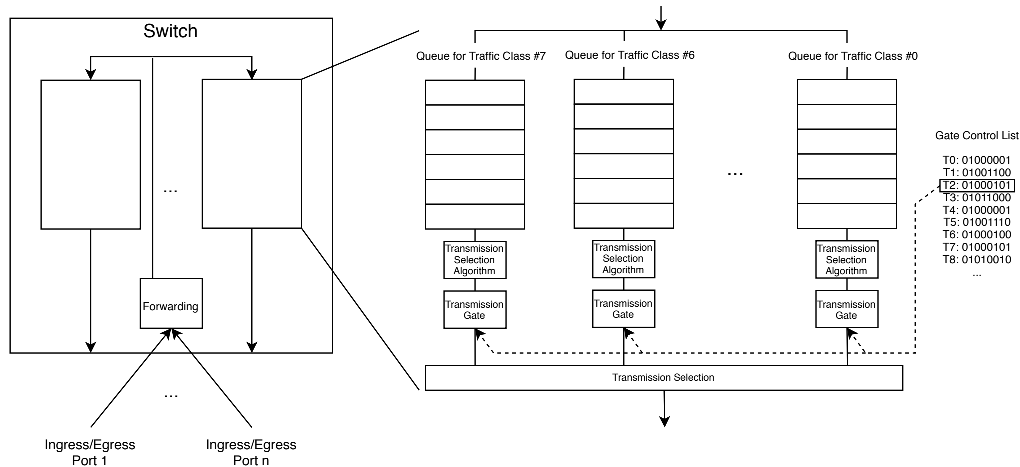


Figure 2.6: Egress Port architecture of a TSN compliant switch [IEE18b].

only one packet of a traffic class can be transmitted. These slots have to be reserved before the actual transmission on the network can take place. Planning and reserving the TDMA slots, such that the transmission meets the requirements of time-sensitive traffic, is called Time-Aware Shaping, schedule synthesis, or scheduling in short. Scheduling is known to be an NP-hard problem [Ull75]; currently, no polynomial scheduling algorithm exists. As a result, scheduling large amounts of packets can become very time-consuming, if not impractical.

2.2.4 TSN Switches

In this section, we present the components introduced by the IEEE 802.1Q and IEEE 802.1Qbv standards for transmitting scheduled traffic. We detail how TSN switches implement the presented TDMA scheme.

Figure 2.3 shows the architecture of a standard Ethernet switch. TSN modifies the architecture and adds several components, cf. Figure 2.6: Priority Queues, Transmission Selection Algorithms (TSAs), Transmission Gates, Gate Control Lists (GCLs) and the Transmission Selection. Each port of the switch is associated with the depicted architecture. We now delve into details of said components.

Priority Queues

After the switch forwards the packet to the correct egress port, it sorts the packet into the corresponding priority queue. Each priority queue holds frames with the associated priority value. The priority of a frame is a function of its PCP value. The PCP value is part of the Q-Tag in the frame's header, as discussed in Section 2.1. Priority encoding and decoding tables stored in the TSN switch map the frame's PCP value to a priority [IEE18b]. These tables can be modified by the network management to support different numbers of priorities. The priorities themselves can then be mapped to traffic classes with different QoS requirements. IEEE 802.1Q itself suggests a priority to traffic class mapping, cf. Table 2.1.

Priority	Traffic Type
1	Background
0	Best Effort
2	Excellent Effort
3	Critical Applications
4	"Video," <100 ms latency and jitter
5	"Voice," <10 ms latency and jitter
6	Internetwork Control
7	Network Control

Table 2.1: Priority to traffic class mapping suggested by IEEE Std 802.1Q [IEE18b].

Transmission Selection Algorithm

The Transmission Selection Algorithm (TSA) is located after the priority queue. It selects packets for transmission according to its selection procedure. The switch only removes a packet from a priority queue and transmits it if it is deemed eligible for transmission by the queue's associated TSA. A trivial TSA, called *Strict Priority*, determines that there is a packet ready for transmission if the associated queue contains at least one frame. Other TSAs implementing more sophisticated selection procedures exist, e.g., the Credit-Based Shaper [IEE18b]; however, they are out of scope for this work.

Transmission Gates

A transmission gate is associated with each priority queue. The state of the gate determines if queued frames can be selected for transmission by the Transmission Selection. For a specific queue, the transmission gate can have one of two states: *open/1* or *closed/0*. An open gate allows the transmission selection to select frames deemed eligible for transmission by the TSA. Conversely, if the gate is closed, the transmission selection cannot select packets from the associated queue.

Gate Control Lists

A GCL controls the states of the transmission gates of an egress port. Every port is associated with a GCL. An entry in a GCL consists of a bit-vector detailing the gate states, and a time interval determining the duration of the gate states. Each entry in a GCL changes the states of the port's transmission gates. The gate operations change the gate's states from *Open* to *Closed* or vice versa. A GCL is finite in length and is repeated periodically after a configurable amount of time T_{cycle}. Scheduling in TSN networks entails calculating the GCLs for every switch. By closing and opening specific gates at specific times, switches can ensure temporal isolation of packets and adherence to the time slots of the TDMA scheme.

Transmission Selection

As its name implies, the Transmission Selection is the component selecting packets for transmission. It checks the queues in descending priority order for a frame to transmit. It can only select a packet in a queue if the corresponding TSA deemed it eligible, and the transmission gate is in the *open* state.

2.3 Discrete Event Simulation

The tracing-based scheduling approach uses a lightweight Discrete Event Simulation (DES). Furthermore, we use a DES to verify our novel scheduling approaches during the evaluation. We lay the theoretical foundation for DESs in this section. We first introduce the notion of DES. Afterward, we detail the main concepts used in DESs. The main concepts are: models, system state, and entities. This section is based on [BCNN10].

A simulation is an approximate imitation of a real-world process or system [BC86]. The behavior of a system over time is analyzed by developing a model of the system. In our case, the system to model is a TSN network. We detail our System Model in Chapter 4. A model has to simplify the real world, while at the same time modeling it as accurately as necessary. A model is predicated on a set of assumptions concerning the operation of the system. If a model is simple enough, it can be solved analytically. However, most systems in the real-world are too complex to be solved analytically. This holds for computer networks. If an analytical solution is not obtainable, numerical methods are employed to simulate the model and generate an artificial history. The system state is a collection of system variables that detail necessary information to describe the system at any point in time [BC86], e.g., the length of a buffer queue. The system state also includes a variable representing time, called *Clock*. The variables and, therefore, the system state, change over time. An entity is any object of a real-world system requiring representation in the model. These entities have attributes and can interact with each other. In this work, the entities in the simulation models are TSN capable switches and hosts.

A discrete-event model is one where the system state only changes at discrete points in time. In contrast, continuous models change continuously over time. The discrete points in time are referred to as event-times. An event marks an occurrence that changes the state of the system. The Event Queue is a data structure holding future events to be executed. The events are ordered by their execution time. When an event occurs, its effects on the system state are calculated. An event can generate future events, which are placed in the Event Queue for later execution. At the start, the simulation initializes the Event Queue with events. After an event was executed, the Clock advances to the time of the next event. This process is repeated until the Event Queue is empty, i.e., the simulation is over, or the simulation reaches a set time-limit.

The tracing-based scheduling approach uses a DES to simulate the transmission of Ethernet packets. It is lightweight in the sense that it only simulates events that are important for schedule synthesis. It uses less overhead than general-purpose discrete-event simulators, and therefore, is faster for our purposes. The discrete event simulator we use for verification is OMNeT++ [OMN19]. OMNeT++ is a modular simulation library. It is primarily used to build network simulations. Out of the box, it does not come with components to simulate Ethernet networks. The INET framework [INE19] provides protocols and other models for communication networks, including Ethernet. However, as

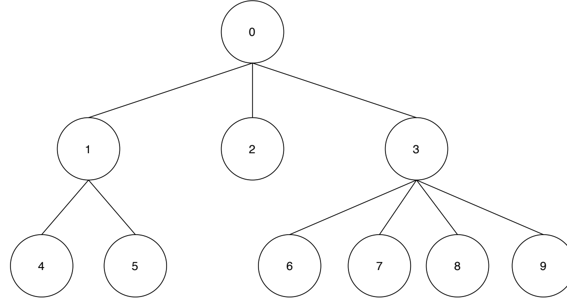


Figure 2.7: A basic tree with node ids as node names.

Algorithm 2.1 Basic tree search algorithm in pseudo-code [RN09].

```
procedure TREE-SEARCH(root_node, queue, value)
    queue ← queue(root_node, queue)                      // Initialization
    while true do
        if queue is empty then return failure
        end if
        node ← dequeue(queue)                            // Step 1
        if node = value then return node                 // Step 2
        end if
        queue ← insert_all(expand(node), queue)          // Step 3
    end while
end procedure
```

of writing, INET does not include TSN components. NeSTiNg [FHC+19] is a simulation model for TSN and enhances INET with TSN-capable components. In this work, we use NeSTiNg to simulate instances of networks and data traffic to verify our novel scheduling approaches.

2.4 Tree Search

The scheduling approaches developed in this work utilize tree search algorithms. Therefore, we present basic tree search algorithms. However, we preface the search algorithms with a brief introduction to trees. Basic tree search algorithms have their limitations concerning time- and space-complexity. To remedy these limitations, researchers developed heuristic and probabilistic tree search algorithms. We give a brief overview of how exactly search algorithms use heuristics to decrease the average search time. Furthermore, we introduce the probabilistic search method we use in this work.

2.4.1 Basic Tree Search Algorithms

Figure 2.7 shows a tree. A tree is a fundamental data structure. It consists of a root node, which itself has sub-trees as its children. Searching these trees for specific values is a well-researched problem in computer science; therefore, many search strategies exist. Listing 2.1 shows the basic tree search algorithm in pseudo-code. We characterize a tree by its depth m and the branching factor

	DFS	BFS
Completeness	Yes (if $m < \infty$)	Yes
Time Complexity	b^m	b^d
Space Complexity	$b \cdot m$	b^d
Optimality	No	Yes

Table 2.2: Table displaying the properties of DFS and BFS search strategies [RN09]. d denotes the depth of the least-cost solution.

b [RN09]. The tree shown in Figure 2.7 has depth $m = 3$ and a non-consistent branching factor. The search algorithm gets the root node of a tree, a queue, and the searched value as inputs. The queue is initialized with the root node. While the queue is not empty, the search algorithm executes the following steps:

Step 1: It pops the next element in the queue.

Step 2: If the popped node is the searched value, the search is successful and the algorithm exits.

Step 3: Otherwise, the algorithm expands the children of the node and inserts them into the queue.

Tree search strategies can be defined by the ordering of the nodes in the *queue*. The basic strategies are: *Breadth-first Search (BFS)* and *Depth-first Search (DFS)*. In BFS the queue is a FIFO queue, in DFS it is a LIFO queue. These strategies are evaluated along the following properties [RN09]:

Completeness: Does the algorithm always find a solution, if it exists?

Time Complexity: The number of nodes generated during the search.

Space Complexity: The maximum number of nodes in memory.

Optimality: Does the algorithm always find the least-cost solution?

Table 2.2 shows the properties for the two basic search strategies. We can see that the DFS finds a solution if it exists. Its time complexity is exponential in the depth of the tree. The time-complexity of the BFS is exponential as well, however, in the depth of the least-cost solution. BFS has exponential space-complexity, DFS has linear space-complexity. The DFS does not always find the least-cost solution, whereas the BFS does. Especially the exponential time-complexity of both strategies leads to challenges in practical applications. If the search space is large, the search becomes time-consuming and impractical. Therefore, researchers introduced probabilistic and heuristic methods to improve average runtimes.

2.4.2 Heuristic and Probabilistic Tree Search

A heuristic is a function that ranks alternatives in search algorithms. It trades optimality and completeness for speed. The heuristic determines the order of the elements in the queue in Listing 2.1. A* is a well-known search algorithm using a heuristic. In A*, the heuristic estimates the remaining cost to reach the search goal. The nodes in the queue are then sorted by the sum of the cost-so-far and the estimated cost-to-go: $total - cost = cost - so - far + estimated\ cost - to - go$ [RN09].

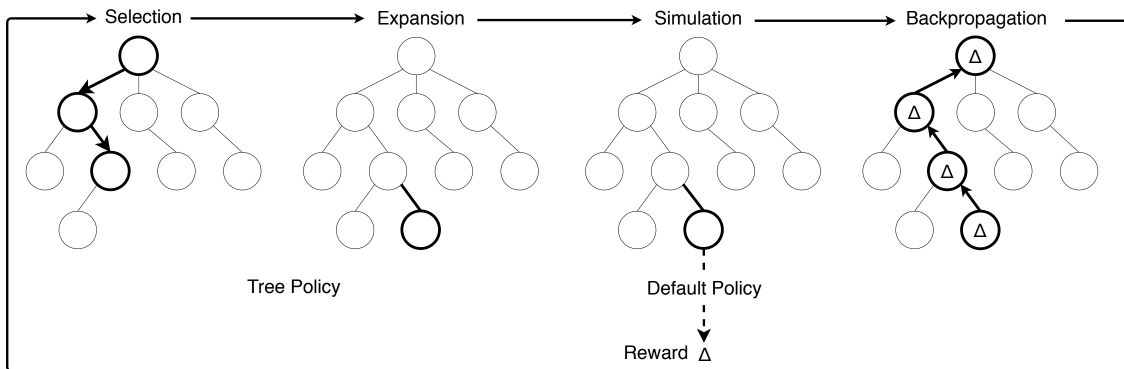


Figure 2.8: One iteration of the MCTS process [BPW+12].

The node with the lowest total cost is then dequeued and expanded. Despite using a heuristic, A* is complete and optimal [RN09]. However, this is not necessarily true for other heuristics. Despite the improvements in search time, the time complexity of A* is still exponential [RN09].

We have seen that search becomes too time-consuming if the search space is large. Monte Carlo methods utilize random sampling of the search space to obtain results in complex environments. They provide approximate solutions, however, with tolerable error bounds and polynomial time-complexity [Was97]. Monte Carlo Tree Search (MCTS) methods have a long history in game-playing algorithms. Recent successes in the game *Go* are responsible for the increased interest in MCTS. Go is a tough game for computers to play since its game trees have high branching factors and are very deep [BPW+12]. Applying MCTS to Go improved the search for advantageous next moves drastically. Because of its great successes in games, researchers apply MCTS to a plethora of other fields, e.g., complex real-world planning and optimization problems. In the subsequent paragraph, we give a brief overview of the MCTS process.

MCTS builds an asymmetric search tree incrementally. Nodes in the tree represent states of the searched environment. The basic algorithm is executed within a computational budget - e.g., time. After exhaustion of the budget, the search is stopped, and the best performing node is returned. One iteration consists of four steps [BPW+12]: *Selection*, *Expansion*, *Simulation* and *Backpropagation*, cf. Figure 2.8.

1. *Selection*: A child selection policy recursively traverses the tree until the most urgent node - according to a utility function - is found. The selection policy starts at the root node in each iteration.

2. *Expansion*: A new child node is added to the previously selected node. The child can be selected randomly or according to a heuristic.

3. *Simulation*: A simulation is executed with the new child node. The simulation results are then used to calculate a reward Δ for the child node.

4. *Backpropagation*: The reward obtained from the simulation is backpropagated along the path the selection policy has taken. The statistics of every node are updated, which are: reward value Δ gathered by expanding that node, and the visit count n.

Algorithm 2.2 Basic MCTS procedure in pseudo-code [BPW+12].

```
procedure MCTS-SEARCH(v_0 : root_node)
    while within computational budget do
        v_selected ←Tree-Policy(v_0)                          // Tree Policy
        Δ ←Default-Policy(v_selected)                         // Default Policy
        Backpropagation(v_selected, Δ)
    end while
return Best-Child(v_0, α)              // Return the best child in the search tree
end procedure
```

To summarize: the MCTS builds the search tree incrementally, i.e., each iteration expands the search tree by one node. For each iteration, the MCTS starts selection from the root node and selects the most urgent node according to a utility function. This node is expanded by adding a child. The MCTS calculates a reward for the new child by executing a simulation with it. Afterward, the reward is backpropagated along the path the selection step has taken. These steps can be grouped into two policies [BPW+12]:

1. *Tree policy*: Selects and creates leaf nodes from the nodes already contained in the search tree. Uses a utility function to select the best node.

2. *Default policy*: Defines how the simulation step is executed.

Algorithm 2.2 shows a basic MCTS algorithm in pseudo-code. After the computational budget is reached, the MCTS returns the best performing child according to the utility function used in the Expansion step.

3 Related Work

After introducing the technical background of this work, we turn to present related research. Scheduling is an established research topic. The earliest scheduling research was conducted on the Job-Shop problem, a well-known optimization problem in operations research. It is relevant since scheduling in real-time networks can be reduced to it. The Job-Shop Scheduling problem considers jobs consisting of operations (typically in a specific order) and machines that execute these operations [Gra66]. A machine can only perform one operation simultaneously. The scheduling problem is defined as follows: find a schedule of operations and machines that satisfies specific constraints, i.e., the order of operations. This problem is equivalent to scheduling in TSN networks. The first TSN scheduling research was published in 2013. However, research on other real-time technologies, e.g., TTEthernet [TTE], was published earlier. Though network architectures of these proprietary technologies differ from the architecture of TSN, the scheduling procedures developed are nevertheless relevant. Since then, different scheduling methods were applied to TSN networks. We classify scheduling approaches into two categories: complete- and heuristic approaches. Complete approaches find a solution, if it exists, whereas heuristic-based approaches are generally faster, but sacrifice completeness for the increase in speed. In the following, we present related research of these two categories.

3.1 Complete Scheduling Approaches

The following section deals with complete scheduling approaches. We further divide them by the method used to formulate the scheduling problem. Typically, researchers formulate the problem either as an optimization problem using an Integer Linear Problem (ILP) formulation or as a Satisfiability Modulo Theories (SMT) problem. In ILP formulations, the constraints are formulated as linear relationships [Wil09]. Afterward, ILP solvers are used to find an optimum with reference to a specific goal, i.e., minimizing overall costs. On the other hand, SMT problems use first-order logic expressions to formulate constraints [BT18]. These constraints can either be satisfied or not; hence, the notion of an optimal solution cannot be applied here. However, both methods find solutions if they exist. If no solution exists, both methods terminate and signal that they cannot find a solution.

The contribution of most papers is two-fold: authors introduce a complete approach to formulate the scheduling constraints. Afterward, they extend their first approach with a heuristic to accelerate the schedule synthesis. They then compare both approaches. In the following, we split the contributions of these papers into their respective categories. We detail the complete approaches in this section; details on the heuristic contributions can be found in Section 3.2.

3.1.1 ILP formulations

This section deals with related research using ILP formulations of the scheduling problem to calculate a schedule. The constraints are formulated as linear relationships [Wil09]. ILP solvers can find an optimal solution.

Hanzálek et al. [HBŠ10] propose a scheduling algorithm that creates static schedules for Profinet IO Isochronus Real Time (IRT) communication. Profinet is a technical standard defining communication over Ethernet. It introduces several traffic classes with different temporal constraints, IRT communication is one of them. IRT traffic has strict bounds on end-to-end latencies. Hanzálek et al. aim to find schedules for IRT traffic with minimal make-span, while respecting further temporal constraints. To do this, they present a formulation of the constraints. Next, they solve the generated constraints with an ILP solver. Lastly, they evaluate their scheduling algorithm on topologies of up to 40 end-hosts and up to 1000 frames. The authors show that optimal schedule calculation only works on topologies with 20 end-systems and 100 tasks in a reasonable amount of time. Duerr et al. [DN16] consider the problem of calculating a TSN schedule for NICs and switches in a TSN network. They map the No-Wait Job-Shop problem to a No-Wait Packet Scheduling (NW-PSP) problem. As a result, they can calculate TSN schedules, which minimize the fragmentation of the network cycle by TSN gate operations. Minimizing fragmentation minimizes the use of guard bands, which potentially decrease the available bandwidth. They show that instances of NW-PSP can be formulated as integer linear problems, which in turn can be solved with ILP solvers. Since NW-PSP is NP-hard, calculating schedules for larger networks does not scale well. The authors attempted to schedule small problem instances with up to 50 streams. However, finding an optimal solution using a state-of-the-art ILP-solver, took over three days. Schweissguth et al. [SDT+17] present an ILP-formation of the Joint Routing and Scheduling problem. Typically, the scheduling of streams is done separately from the routing. This reduces the solution space since streams with identical routes that are not schedulable could be rerouted to make them schedulable. The authors extend the ILP-formulation of the scheduling problem with routing constraints, calling it Joint Routing and Scheduling (JRaS). They compare their approach to a scheduling approach with pre-calculated routes. They show that previously not schedulable stream-sets can be scheduled with their novel approach. However, this comes with a trade-off on calculation time, since the additional routing constraints add complexity. They evaluated their approach on networks with 12 end-systems and up to 40 streams.

3.1.2 SMT formulations

SMT is a decision problem for logical formulas [BT18]. This section deals with related research using SMT formulations of the scheduling problem.

In [Ste10], Steiner proposes formal SMT specifications of scheduling constraints for TTEthernet networks. TTEthernet is a proprietary technology, extending Ethernet with real-time capabilities. Steiner discusses the appropriateness of a general-purpose tool to solve the scheduling problem in TTEthernet networks. To do this, he uses an SMT solver. Afterward, he evaluates the performance of his scheduling tool. Steiner introduces a basic scheduler: it generates all constraints and tries to satisfy them at once. He refers to it as the *one-shot approach*. The author shows that scheduling up to 140 frames takes up to 30 minutes. However, scheduling larger amounts becomes impractical; the calculation time of the basic scheduler increases exponentially. Craciunas et al. [CO14]

propose scheduling approaches to simultaneously synthesize application and network schedules for communicating time-triggered applications in a TTEthernet network. They use SMT to formulate the scheduling constraints and an SMT-solver to solve the constraints. Similar to Steiner [Ste10], they present a one-shot approach. The one-shot scheduler tries to solve all SMT constraints at once. The significant contribution of Craciunas et al. [CO14] in relation to Steiner [Ste10] is that they consider application schedules and the network schedule simultaneously, thus covering the whole solution space. In contrast, Steiner's scheduler solely schedules the network. The authors evaluate their approach on topologies with up to 48 end-systems. The one-shot approach is not able to synthesize a schedule for the smallest evaluated networks in 100 minutes. In [COCS16], Craciunas et al. present research where they adapt their previous scheduling algorithm from [CO14] for TTEthernet networks to TSN networks. Since TSN has a different network architecture than TTEthernet, TSN networks require a different network model. The authors evaluate their complete scheduling approach on networks of up to 7 end-systems and up to 100 streams or 1500 frames. The authors show that the run time of their algorithm increases exponentially with the number of frames.

All complete approaches calculating optimal solutions suffer from the inherent NP-completeness of the scheduling problem. While scheduling small topologies and stream sets of up to 50 end-systems and 100 streams is practical, scheduling larger problem sizes quickly becomes impractical due to the exponential increase in computation time. As a result, researchers have turned to use heuristics to accelerate schedule synthesis.

3.2 Heuristic Scheduling Approaches

The NP-hardness of the scheduling problem makes it difficult to schedule large numbers of streams on large topologies. As previously shown, scheduling over 100 streams becomes very time-consuming with complete approaches. However, the ability to schedule larger topologies and larger number of frames is needed for practical applications, e.g., in large factory networks. For this reason, researchers began enhancing their complete approaches with heuristics. Heuristic approaches trade completeness against calculation speed. We distinguish two types of heuristics: solver-based heuristics and stream-based heuristics. Researchers using solver-based heuristics enhance their constraint solver with a heuristic procedure to accelerate schedule synthesis. Stream-based heuristics leverage properties of streams, or topologies, to reduce the complexity of the scheduling problem. The next subsection deals with scheduling approaches using heuristics. We first detail scheduling approaches using solver-based heuristics. Afterward, we present approaches leveraging stream-based heuristics.

3.2.1 Solver-based Heuristics

Yamada et al. [YN92] propose a genetic algorithm for solving job-shop problems. They use an existing algorithm to generate parent schedules and then define a crossover operation to generate offspring schedules. They show that their approach finds optimal solutions to smaller sized Job-Shop problems (ten jobs and ten machines), in contrast to previous methods, which could not find an optimal solution. Furthermore, they showed that their approach could find near-optimal solutions for larger problem sizes, for which a branch&bound approach cannot compute solutions. In [Ste10]

Steiner proposes SMT scheduling constraints for TTEthernet networks. As shown in Section 3.1.2, he uses a one-shot approach to solve these constraints. Additionally to this basic scheduler, he introduces an incremental scheduler: this scheduling algorithm generates a subset of constraints and invokes the SMT solver on that subset. The incremental scheduler merges the partial solutions to form a complete schedule. The author shows that the incremental scheduler takes up to 30 minutes to schedule 1000 frames. However, scheduling larger amounts of frames is impractical, since the calculation time increases exponentially. In Section 3.1.1, we detailed the complete scheduling approach of Hanzálek et al. [HBŠ10]. They propose a scheduling algorithm that creates static schedules for Profinet IO IRT communication. The authors present a formulation of the constraints and solve the generated constraints with an ILP solver and other heuristic solvers. They evaluate their algorithm on topologies of up to 40 end-hosts and up to 1000 frames. The authors show that optimal schedule generation only works on topologies with 20 end-systems and 100 frames in a reasonable amount of time. Heuristics can solve larger problem sizes of up to 1000 frames in about 10 seconds. Tamas-Selicean et al. [TPS12] draw upon Steiner's [Ste11] research. Therefore, we first present the relevant parts of Steiner's paper, before detailing the contribution of Tamas-Selicean et al. In [Ste11], Steiner discusses static scheduling methods, which allow for time-triggered traffic and non-time-triggered traffic to co-exist in the same network. He introduces the concept of schedule porosity: time intervals, where no time-triggered traffic is transmitted, are inserted into the network cycle. During these phases, devices can transmit non-time-triggered traffic. As a result, jitter and latency bounds can be given even for non-time-triggered traffic. Steiner introduces three ways to integrate phases for non-time-triggered traffic into the network cycle: a-priori, a-posteriori, and by segmenting the network cycle into slots. In a-priori integration, the non-time-triggered phases are given as formal constraints to the scheduling algorithm. The additional constraints increase the complexity of the scheduling problem and the runtime of the scheduling algorithm. In a-posteriori integration, the schedule of time-triggered streams is interleaved with non-time-triggered phases. In contrast to a-priori integration, a-posteriori integration does not increase complexity, since no additional constraints need to be generated. Lastly, schedule porosity can be achieved by segmenting the network cycle into slots. One maximum-sized time-triggered frame and one maximum sized non-time-triggered frame can be communicated during one slot. As a result, the network cycle is fragmented into small periods, where one frame is transmitted in each period. All in all, according to Steiner, up to three levels of schedule porosity can be introduced into a schedule. Tamas-Selicean et al. [TPS12] use the a-priori approach introduced by Steiner to schedule mixed-criticality networks, such that they not only meet the arrival deadlines of time-triggered traffic but deadlines of non-time-triggered traffic as well. To calculate these schedules, they use tabu search as a heuristic and minimize the end-to-end delay of non-time-triggered traffic. The authors show in their evaluation that their approach significantly reduces the end-to-end delay of non-time-triggered traffic. Thus, their approach improves the schedulability of time-triggered and non-time-triggered traffic. Additionally to their complete ILP approach, Duerr et al. [DN16] introduce a heuristic approach for solving NW-PSP instances. They use tabu-search as a heuristic to accelerate schedule calculation. The authors evaluate their heuristic approach on stream sets of up to 1500 streams. Scheduling 1500 streams takes three hours with the heuristic scheduling approach. In contrast, scheduling 50 streams without using heuristics took over three days. Pozo et al. [PRHS15] present a study identifying and evaluating which SMT-solver parameters impact the calculation time of TTEthernet schedules. They use an incremental scheduling approach, similar to Steiner's [Ste10]. They altered multiple parameters of an SMT-solver and evaluated those changes on topologies of up to 50 data-stream links and up to 1000 frames. They show that with specific parameters, the synthesis time can be reduced significantly, up to 75% in best cases. Gavrilut et al. [GP18] propose a schedule

calculation procedure that takes lower priority traffic into account. Lower priority traffic often has constraints on worst-case delays. Conventional scheduling procedures only schedule time-triggered traffic, disregarding the performance of lower priority traffic. Disregarding lower priority streams during scheduling potentially renders them unusable. They use the Greedy Randomized Adaptive Search Procedure (GRASP) as a heuristic during schedule calculation. The authors evaluate their scheduling proposal on topologies of up to 32 end-systems, 35 time-triggered, and 26 lower priority streams. Schedules that previously violate their delay bounds can be scheduled by considering lower priority traffic in the schedule calculation. However, they do not show any performance measurements of their scheduling approach. Considering the performance of lower-priority frames increases the scheduling complexity, and thus, the schedule calculation time.

3.2.2 Stream-based Heuristics

Stream-based heuristics use topology and stream properties to reduce the complexity of the scheduling problem. In the following, we present research using this strategy to accelerate schedule synthesis.

Pozo et al. [PRH19] propose a segmented scheduling approach to the scheduling problem of time-triggered networks. They evaluate their approach on large stream sets of up to 25000 frames, which is significantly larger than previously considered problem sizes. In contrast to other publications, they focus not only on wired networks but also include wireless links in their network model. They calculate schedules using an SMT formulation of the scheduling problem. However, to reduce the problem size, and therefore, reduce calculation time, they employ a divide-and-conquer strategy: they segment the hyper-cycle of the network into several time segments. For each segment, they try to allocate as many frames as possible into a single segment. Then, each segment is scheduled separately. By segmenting the hyper-cycle, they trade the likelihood of finding a schedule against calculation time, since segmenting the hyper-cycle and scheduling each segment reduces the solution space. The authors evaluate their scheduling approach on networks of up to 241 end-systems and 750 streams or 25000 frames. They show that the schedule calculation time increases linearly relative to the number of frames. However, the segmentation of the cycle does not consider dependencies of streams, which might decrease schedulability. Hellmanns et al. [HFG+20] compare different scheduling approaches and analyze stream delays introduced by each approach. They partition streams into two different classes: time-triggered streams with strict bounds on arrival times and cyclic traffic, where packets only need to arrive once per network cycle. Time-triggered streams need to be scheduled individually, which is NP-hard. Cyclic streams can be accumulated and scheduled as a whole, referred to by the authors as class-based scheduling. Accumulating streams and scheduling them together decreases the complexity of the scheduling problem. Furthermore, the authors provide a formula to calculate a schedule for class-based traffic. In essence, the authors reduce the NP-hard scheduling problem to the evaluation of the derived formula. However, the reduction in complexity comes at the expense of lower network utilization, since the formula overestimates the required bandwidth. The class-based approach does not work for time-triggered streams, as they have lower bounds on arrival times. The class-based approach cannot meet these bounds. Hellmanns et al. evaluated their class-based approach on networks of up to 30 end-systems and 30 streams. In [HGF+20], Hellmanns et al. propose a scheduling model for converged networks that supports multiple traffic types. They introduce a new scheduling method referred to as ”hierarchical scheduling”. The authors propose to split the network into hierarchical sub-networks. They schedule

each sub-network and its corresponding streams independently, which reduces the problem size and makes the computation more scalable. Partitioning the network in such a way utilizes a property of streams with low bounds on latency: they typically occur between devices in the same sub-network/hierarchical level. The authors call these streams intra-level streams. Streams that span across multiple sub-networks are called inter-level streams. To schedule these two types of streams, the authors divide the scheduling into two phases: In the first phase, intra-level streams get scheduled independently in each sub-network, using a contemporary scheduling approach. In the second phase, inter-level streams get scheduled. Scheduling in the second phase is done by simulating the inter-level streams with a network simulator and deriving the schedule from the simulation results. The authors refer to deriving the schedule from a simulation as tracing-based scheduling. After all inter-level streams arrived at their destinations, other lower priority traffic classes can be transmitted. As a result, the network cycle is divided into separate phases for different traffic classes. Dividing the cycle into separate phases resembles the a-priori approach introduced by Steiner [Ste11]. Hellmanns et al. test their approach on networks with up to 2500 end-hosts and up to 2000 streams and compare their results to the heuristic scheduling algorithm developed by Duerr et al. [DN16]. Their hierarchical scheduling approach performs up to two magnitudes faster than their baseline. However, this approach does not work when the tracing-based scheduler cannot find a schedule that meets the stream requirements. Currently, no methods exist to alter the tracing-based schedule such that the requirements are met.

Complete scheduling works well for small problem sizes of up to 50 streams. However, schedule computation time increases exponentially, thus, making scheduling large problems impractical. Nevertheless, scheduling larger instances is essential for practical applications. Therefore, researchers employ heuristics to accelerate computation. Heuristics sacrifice completeness for calculation speed. Regardless, in practice, they have shown to provide good results close to the optimum. In [HGF+20], Hellmanns et al. introduce a hierarchical scheduling approach. They propose a new approach called *tracing-based scheduling*. Tracing-based scheduling uses a lightweight network simulator to simulate network traffic. The authors use the simulation results to derive a schedule. Their evaluation shows that tracing-based scheduling can synthesize schedules for very large scheduling instances. Solving these large scheduling instances is not practical with other approaches. However, the tracing-based approach fails if the simulation reveals that streams violate their deadlines. In our work, we aim at expanding the tracing-based approach. We explore procedures to handle streams that violate their deadline. Our goal is to expand the tracing-based approach to a full-blown scheduling algorithm, which considers latency bounds of streams and schedules them accordingly.

4 System Model

As mentioned previously, the tracing-based scheduling approach uses a lightweight network simulator to simulate streams and derive a schedule from the simulation results [HGF+20]. Furthermore, we evaluate the resulting schedules in the simulation framework NeSTiNg [FHC+19]. These simulators utilize models of TSN networks and their components. It is crucial to understand how these simulators model computer networks to understand the limitations and applicability of the results of this work. Therefore, we present our model of a TSN network. In this work, we focus on industrial applications of TSN. Topologies and streams with specific properties are prevalent in these applications. These properties impact the topologies and streams we evaluate in this work. Therefore, we discuss how we model a TSN network, such that our models reflect these properties. Afterward, we present our model of TSN devices. It includes a switch model and a host model. A complete system model includes the network components and the data transmitted by these components. Therefore, we present how we model TSN data streams. These data streams encounter certain delays on links or during processing on nodes. For this reason, we present different types of traffic delays.

4.1 Network Model

We model a TSN network as a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ denotes the set of vertices, while $\mathcal{E}$ denotes the set of edges. $\mathcal{V}$ is comprised of two vertex types: *switches* and *hosts*. More formally, this is:

$$vertex\ type : \mathcal{V} \rightarrow \{switch,\ host\}$$

TSN is intended for use in industrial domains, i.e. Industry 4.0. Networks in these domains follow specific topology layouts. Figure 4.1 shows an example factory network as a undirected graph. A factory network typically consists of a backbone. The backbone is responsible for connecting sub-networks and is usually a ring topology. These sub-networks represent local networks, such as networks connecting the machines of a factory line. Factory lines, as their name implies, are a line topology. Additionally, the end devices of some lines are connected for redundancy, forming a ring topology. We model the factory layout by generating the aforementioned directed graphs. We first generate a backbone ring and select random vertices from it. We refer to these selected vertices as *gateway* nodes. These gateway nodes connect sub-topologies, more specifically lines and rings, to the backbone ring. All generated nodes so far are of the *switch* type. Each switch node is connected to exactly one node of the type *host*.

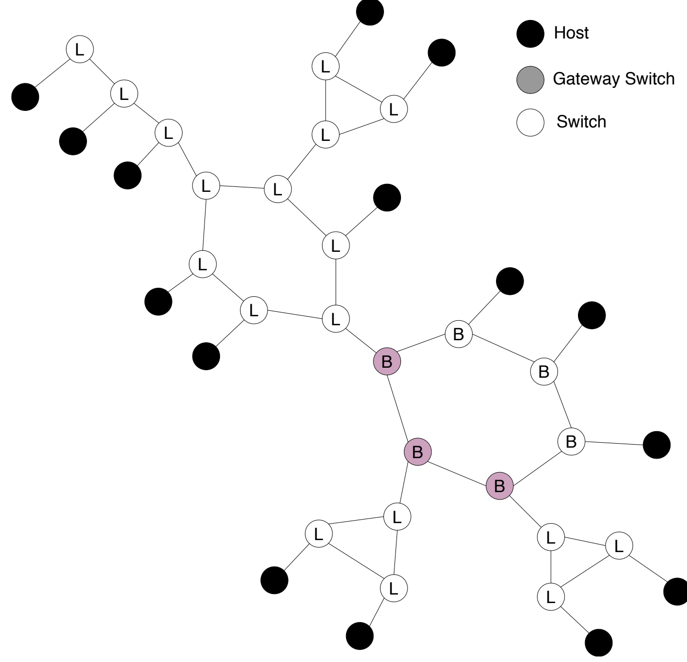


Figure 4.1: Graph showing an example factory layout. Backbone nodes are marked with *B*, LAN nodes are marked with *L*.

4.2 Switch Model

In this work, we use two different event-based network simulators: the tracing-based simulator [HGF+20] and NeSTiNg [FHC+19]. Both simulators model switches. In this section, we elaborate on similarities and differences of switch models in both simulators.

A switch node is a Node $v \in \mathcal{V}$ where $vertex\ type(v) = switch$. Switch vertices have a property defined by the relation below:

$$port : \mathcal{V} \rightarrow \mathbb{N}; v \mapsto x; v \in (u, v) \in \mathcal{E}$$

Since switches are connected via ports, we map outgoing edges onto a specific port. The tracing-based simulator implements switches, as shown in Figure 2.3. It implements the standard IEEE Ethernet switch architecture, and therefore, is not able to simulate TSN networks. However, for tracing-based scheduling, this is not necessary. The tracing-based scheduler simulates the behavior of data streams without the influence of TSN mechanisms. For each data stream, the tracing-based simulator generates a history of events. A data stream history consists of a list of tuples ($event$, $timestamp$). The variable $event$ can take one of the following values: $event \in \{transmission_start, packet_received, packet_processed, packet_queued\}$. The stream history of all streams can then be used to derive a GCL for every switch node.

In contrast to the tracing-based simulator, NeSTiNg does implement the TSN switch architecture. This is necessary since we use NeSTiNg to verify the schedules generated by the tracing-based scheduler. We use *Strict Priority* as the Transmission Selection Algorithm (TSA). To recap, Strict Priority selects a frame for transmission if the associated queue holds at least one frame, cf. Section 2.2.

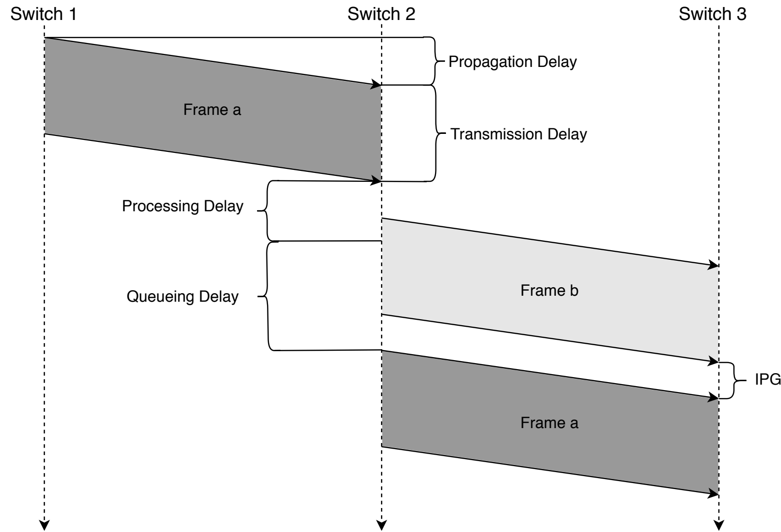


Figure 4.2: Sequence diagram showing the delays a packet encounters on its path.

4.3 Host Model

Hosts are modeled identically in both simulators. Vertices, where $vertex\ type(v) = host$ holds, are host nodes. Opposite to switches, hosts cannot forward packets. They only transmit and receive them. Consequently, host nodes can only be the start and end nodes of data streams. Hosts do not have a processing delay associated, since they do not forward packets. Apart from that, hosts behave like switches. They queue packets before transmitting them and consume received packets.

4.4 Delay Model

A packet encounters multiple delays on its path. It is essential to understand how delays affect packet transmissions to understand and develop scheduling procedures. In this section, we detail the delays encountered by a packet. We differentiate four different delays: *propagation delay* d_{pg}, *transmission delay* d_t, *processing delay* d_{pc} and *queueing delay* d_q, cf. Figure 4.2.

Propagation Delay: Propagation delay is the time it takes the signal to travel from one device to another. The transmission medium carrying the signal determines the delay. The delay can be calculated by dividing the length of the cable with the propagation speed of the signal: $d_{pg} = \frac{cable\ length}{propagation\ speed}$. The propagation speed is dependent on the material of the cable, for copper it is $\sim 2 \times 10^8\,\mathrm{m\,s^{-1}}$. Therefore, the propagation delay for a 40 m long copper wire is: $d_{pg} = \frac{40\,\mathrm{m}}{2 \times 10^8\,\mathrm{m\,s^{-1}}} = 200\,\mathrm{ns}$. Propagation delay is a property of links in a topology. Consequently, we expand our previous definition of TSN topologies, cf. Section 4.1, with the following relation:

$$propagation\ delay : \mathcal{E} \rightarrow \mathbb{N}$$

This relation maps edges to propagation delays. We use 200 ns as the propagation delay for all edges in our topologies.

Transmission Delay: Transmission delay is the time required for the sender to serialize all bits of a packet onto the transmission link. The link speed of the connection and the size of the packet determine the transmission delay: $d_t = \frac{packet\ size}{link\ speed}$. In 1 Gbps Ethernet, the transmission delay for 1 B of data is: $d_t = \frac{1\,\text{B}\cdot 8}{1\,\text{Gbps}} = 8\,\text{ns}$. Therefore, transmitting an Ethernet packet with the maximum payload size (1,500 B) takes $\sim$ 12 µs. Similarly to propagation delay, link speed is a link property. For that reason, we add another property to our TSN network model:

$$link\ speed : \mathcal{E} \rightarrow \mathbb{N}$$

This relation maps edges to link speeds. Gigabit Ethernet is widely adopted today; therefore we set the link speed for all edges to 1 Gbps.

Processing Delay: As its name implies, processing delay is the time it takes a switch to process an incoming packet. The processing delay includes reading the header, determining the next hop of the packet, and checking it for bit-errors. We extend our previous definition of switches with the following relation.

$$processing\ delay : \mathcal{V} \rightarrow \mathbb{N}$$

This relation maps vertices to a processing delay value. For modern Gigabit switches, the processing delay is between 2 µs and 5 µs [DK+14]. For our switch nodes, we choose a processing delay of 2 µs.

Queueing Delay: Queueing delay is the delay a packet incurs when another packet blocks the transmission link. In Figure 4.2, *Frame b* occupies the transmission link. Therefore, *Frame a* queues until *Frame b* is fully transmitted and the IPG elapsed. Afterward, *Frame a* is transmitted.

In this section, we established our Delay Model and set properties to specific values, e.g., the processing delay to 2 µs. However, it is important to note that the scheduling procedures we develop in this work are agnostic to any specific values of topology properties, vertex properties, traffic properties, or delay properties.

4.5 Traffic Model

The punctual transmission of data streams in TSN networks is crucial. Their requirements categorize these data streams. In the following, we unfold which traffic types we model and how we model them.

The Industrial Internet Consortium (IIC) published a white paper describing traffic types in time-sensitive networks [APB+18]. They define eight traffic types. However, only one traffic type is relevant for this work: isochronous traffic. Isochronous traffic refers to traffic transmitted by time-synchronized applications/devices. It is transmitted with small cycle times T_{cycle} of 100 µs $\sim$ 2 ms. Isochronous traffic requires a guaranteed delivery time; otherwise, the late packet is discarded for that cycle, impeding the correct execution of the time-sensitive application. These low cycles times and tight bounds on latencies require no interference from cross-traffic. Payload sizes and deadlines are known beforehand and remain constant. All these demands require isochronous traffic to be scheduled during the network design stage. In TSN, isochronous traffic is referred to as scheduled traffic and has the highest priority.

We refer to a set of data streams as *stream set*: $stream \in \{stream\,1,\ stream\,2,\ \cdots,\ stream\,n\} = \mathcal{S}$. A stream is an entity in our simulation model and models a TSN packet. A stream has certain properties:

$$\begin{aligned}
Source &: \mathcal{S} \rightarrow \mathcal{V} \\
Target &: \mathcal{S} \rightarrow \mathcal{V} \\
Path &: \mathcal{S} \rightarrow \mathcal{P}(\mathcal{V}) \\
Size &: \mathcal{S} \rightarrow \mathbb{N} \\
Start &: \mathcal{S} \rightarrow \mathbb{N} \\
Deadline &: \mathcal{S} \rightarrow \mathbb{N} \\
Cycle\,Time &: \mathcal{S} \rightarrow \mathbb{N}
\end{aligned}$$

Source is the vertex from which the stream originates; conversely, *target* denotes the stream's destination. *Path* specifies the path taken by the stream's packet. *Start* denotes the time at which the source vertex inserts the stream's packet into its queue. The *Deadline* property is the most important property for our purposes. *Cycle Time* denotes the transmission period of the stream. For our purposes, every stream in a stream set has the same *Cycle Time*: $\forall s \in \mathcal{S} : Cycle\,Time(s) = c, c \in \mathbb{N}$. It denotes the time at which the stream has to arrive at its destination. At their core, all scheduling algorithms try to find a start time, such that the stream arrives at its destination before its deadline elapses. The deadlines are pre-determined by the applications using the TSN network. Usually, locally adjacent machines transmit streams with tight deadlines. As a result, their paths are short. In contrast, distant machines transmit streams with large deadlines. Therefore, we choose the deadline as a linear function of the length of the stream's path. We define additional relations that allow us to gather more information about individual streams:

$$\begin{aligned}
Arrival\ time &: \mathcal{S} \rightarrow \mathbb{N} \\
Start\ queueing &: \mathcal{S} \times \mathcal{V} \rightarrow \mathbb{N} \\
Start\ transmission &: \mathcal{S} \rightarrow \mathbb{N}
\end{aligned}$$

Arrival time gets a Stream $s \in \mathcal{S}$ and returns the time at which the destination received the Stream's packet. *Start queueing* additionally gets a Node $v \in \mathcal{V}$ and returns the time when Node v inserted the Stream's packet into its buffer queue. *Start transmission* returns the time at which the source started the transmission of Stream s.

5 Scheduling Approaches

In this chapter, we develop the conceptual foundation for our novel scheduling approaches. First and foremost, we present our formulation of the scheduling problem. The formulation constitutes the basis for all scheduling approaches we develop in this work. Afterward, we develop four novel scheduling approaches: Naive approach, Link Time Remaining Time, Link Time Late Streams, and Link Time Monte Carlo Tree Search approaches.

5.1 Scheduling as a Search Problem

Most scheduling algorithms express the network model as constraints and try to fulfill them to calculate a solution. However, as we have seen, this approach suffers from the inherent NP-hardness of the scheduling problem. We propose a different approach: conceptualizing scheduling as a search problem.

The input of every scheduler is a set of streams that need to be scheduled. As shown in Section 4.5, our stream specification includes a start time and a deadline time for each stream. Schedulers try to find start times, such that the streams obey their deadlines. We denote a Stream s, violating its deadline, as a *Late stream*. Researchers defined the scheduling problem as follows [COCS16]:

Definition 5.1.1
Given a set of streams $\mathcal{S}$, find start times, such that $\forall s \in \mathcal{S} : arrival\ time(s) \leq Deadline(s)$ holds true.

We consider the scheduling problem as a search problem. Before we present our definition, we first introduce the concept of *stream operations*.

Definition 5.1.2
We define a stream operation o, or operation in short, as delaying the start time of a Stream $s \in \mathcal{S}$. More formally: $o(s, x) : Start'(s) = Start(s) + x,\ x \in \mathbb{N}$.

When considering individual streams, delaying the start time of a stream might seem counterproductive: by delaying the stream, we delay its arrival time. However, when we consider the whole stream set, this makes sense. Streams queue in the buffers of switches. A stream might queue after other packets in multiple switches, and therefore, is head-of-line blocked on multiple hops. The queueing might cause the stream to arrive after its deadline. However, this delaying the interfering streams can prevent late arrival. We can delay an interfering stream, such that it queues after the late stream at the critical hop. By arriving after the late stream, the interfering stream does not block the late stream from transmission anymore. Now, the late stream is not queued after the interfering

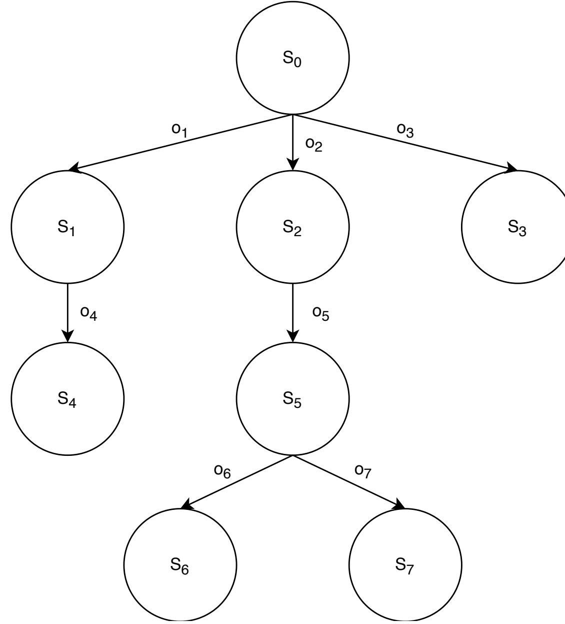


Figure 5.1: Visualization of Definition 5.1.3.

streams and might arrive before its deadline. Note that *stream operations* only exhibit the described effect if queueing occurs. Without queueing, a stream is not head-of-line blocked by other streams. In that case, no interfering streams exist, which can be delayed.

With the definition of stream operations, we expand Definition 5.1.1 as follows:

Definition 5.1.3
Given a set of streams $\mathcal{S}$, find a list of stream operations $(o_1, o_2, \cdots, o_n)$, such that the initial stream set $\mathcal{S}$ is transformed into a stream set $\mathcal{S}_{sol}$, where $\forall s \in \mathcal{S}_{sol} : arrival\ time(s) \leq Deadline(s)$ holds true.

Figure 5.1 shows a visualization of our definition. $\mathcal{S}_0$ is the input stream set, where all streams have start time 0: $\forall s \in \mathcal{S}_0 : Start(s) = 0$. A certain amount of streams in $\mathcal{S}_0$ arrive late. We refer to these streams as *Late streams* and the initial number of Late streams as n_{init_late}. $\mathcal{S}_0$ is the root node of the search tree. Child nodes are expanded by selecting a Stream $s \in \mathcal{S}_0$ and executing an stream operation o on $\mathcal{S}_0$. The child node, e.g., $\mathcal{S}_1$, is then checked for late streams. If no streams arrive late, we found a stream set that fulfills the requirement in Definition 5.1.3 and the scheduling is considered successful. Otherwise, we continue with the applied search strategy. How fast a solution is found depends on the selection strategy for node expansion.

5.2 Designing Scheduling Approaches

In Section 5.1, we formulated the scheduling problem as a search problem. The search starts at an initial stream set $\mathcal{S}_0$, where all start times are set to zero. The search algorithm then expands the children from $\mathcal{S}_0$ by performing *stream operations*. A solution is found when all streams in the stream set of the expanded node arrive timely, i.e., before their deadline. We use the stream

operation path, from the root node to the solution node, to transform the input stream set $\mathcal{S}_0$ into the solution stream set $\mathcal{S}_{sol}$. Scheduling a stream set comes down to finding $\mathcal{S}_{sol}$ as fast as possible. As presented in Section 2.4, different strategies for tree search exist. We detailed the two basic strategies: BFS and DFS. However, the search for $\mathcal{S}_{sol}$ is not trivial: the set of possible stream operations on a stream set $\mathcal{S}_i$ is infinite since a Stream $s \in \mathcal{S}_i$ can be delayed by an arbitrary amount with a stream operation. A DFS or BFS would never terminate. To reduce the search space, we develop selection strategies, which expand nodes that most likely lead to a solution stream set. Furthermore, we develop strategies to calculate reasonable stream operations on nodes in the search tree. The following scheduling approaches employ different strategies. In total, we developed four approaches: *Naive approach*, *Link Time Remaining Time*, *Link Time Late Streams* and *Link Time Monte Carlo Tree Search*. In the following sections, we explain how each scheduling approach operates. The approaches differ in two dimensions: search tree traversal and calculation of reasonable stream operations. We, therefore, structure the subsequent sections in the following manner:

Stream Operation Calculation: Here, we explain how the presented approach calculates the next stream operation to create a new child node. The explanation includes determining the interfering streams and calculating the actual stream operations on the interfering streams.

Tree Traversal: Here, we explain how the presented approach traverses the search tree. Selecting a reasonable node to expand is crucial for the performance of the search.

Discussion: Here, we discuss the presented approach and its advantages, disadvantages, and how we expect the approach to perform.

5.2.1 Naive

In this section, we present the Naive approach.

Stream Operation Calculation

In the following, we refer to streams that interfere and block Late streams as conflict streams. For the Naive approach, we propose the following definition for conflict streams.

Definition 5.2.1
A Stream s conflicts with a Late stream s_{late}, iff the intersection of the streams' paths is not empty, i.e. $s\ is\ in\ conflict\ with\ s_{late} \iff Path(s) \cap Path(s_{late}) \neq \emptyset$.

This is where the Naive approach gets its name from: we delay all streams that share their paths with Late streams, regardless if the Late stream is actually Head-of-line blocked. Applying this expression to every Late stream results in a set of conflict streams $\mathcal{S}_{conflict} \subseteq \mathcal{S}$. We delay the conflict streams by adding 1,000 ns to the start time of every stream:

$$\forall s_{conflict} \in \mathcal{S}_{conflict} : operation(s_{conflict}, 1000).$$

After the stream operations were applied to every conflict stream, the stream set is re-evaluated for Late streams. We aggregate all stream operations applied between evaluations for Late streams into sets of stream operations:

$$O = \{o_i | o_i \text{ is applied between evaluations}\}.$$

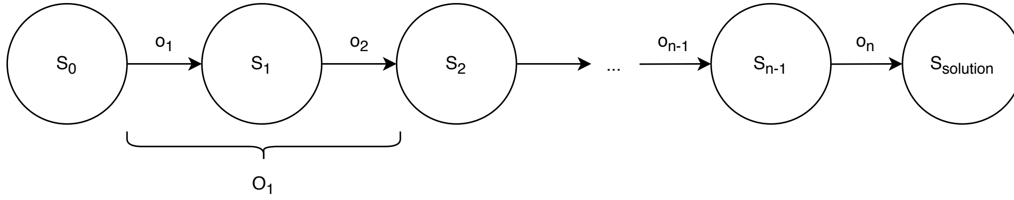


Figure 5.2: Chain resulting from the search with the Naive approach.

Tree Traversal

The Naive approach does not backtrack. The resulting search tree is a chain, cf. Figure 5.2.

Discussion

The Naive approach classifies a stream as a conflict stream if it shares its path with a Late stream. As a result, streams are classified as conflict streams, even when they do not head-of-line block a Late stream. All conflict streams are delayed by 1,000 ns. Thus, even streams that do not block Late streams are delayed. Furthermore, the delay of 1,000 ns might not be sufficient for the conflict stream to queue after the Late stream and not block it anymore. Therefore, we do not expect the Naive approach to perform well. However, we use it as a baseline for the evaluation of our other approaches.

5.2.2 Link Time Remaining Time

In this section, we present the Link Time Remaining Time approach.

Stream Operation Calculation

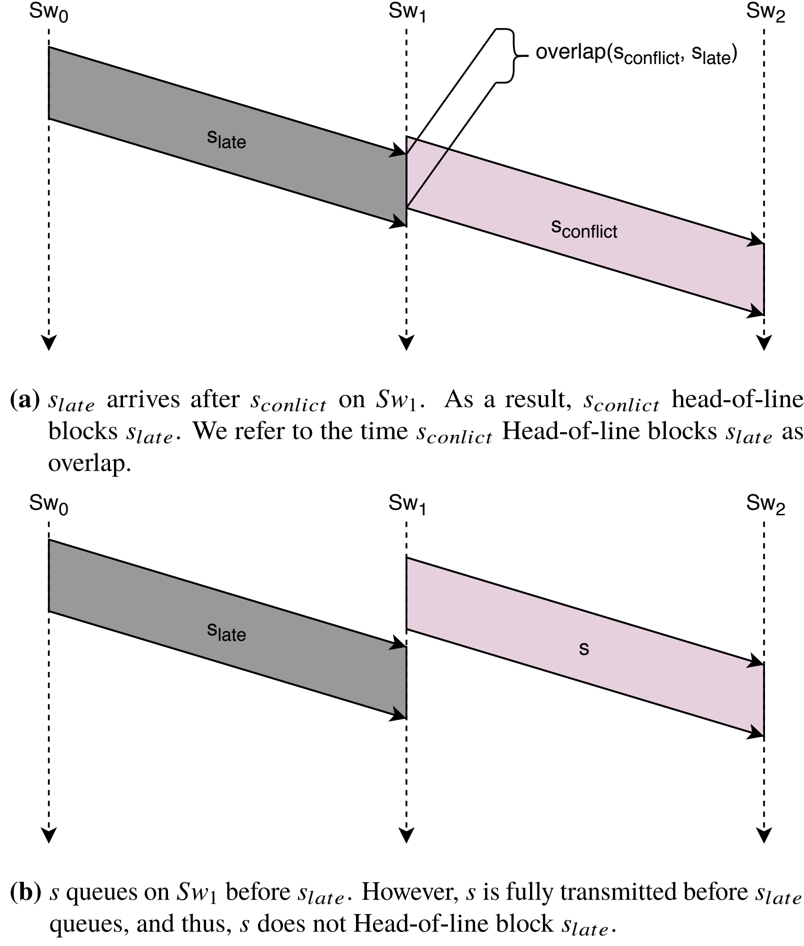
In contrast to the Naive approach, the Link Time approaches consider the actual transmission times of packets when determining conflict streams. Instead of analyzing paths of streams, we analyze the queueing behavior in switch nodes.

Definition 5.2.2
Given are Stream s, Late stream s_{late} and vertex $v \in \mathcal{V}$. We consider Stream s to be a conflict stream, iff s is queued before s_{late} on the same vertex v, and s is not yet fully transmitted by v.

This definition refines Definition 5.2.1, by taking the actual queueing behavior of individual packets into account. If Stream s is fully transmitted when s_{late} queues, it does not block s_{late} from transmission, and therefore, we do not classify it as a conflict stream, cf. Figure 5.3. We calculate an overlap with a Late stream s_{late} and a conflict stream $s_{conflict}$:

$$overlap(s_{conflict}) = start\ queueing(s_{conflict},\ v) - start\ queueing(s_{late},\ v),\ v \in \mathcal{V}$$

The overlap denotes the time $s_{conflict}$ blocks s_{late} from transmission. From the result, we can



(a) s_{late} arrives after $s_{conlict}$ on Sw_1. As a result, $s_{conlict}$ head-of-line blocks s_{late}. We refer to the time $s_{conlict}$ Head-of-line blocks s_{late} as overlap.

(b) s queues on Sw_1 before s_{late}. However, s is fully transmitted before s_{late} queues, and thus, s does not Head-of-line block s_{late}.

Figure 5.3: Visualizing Definition 5.2.2.

derive the Tuple

$$t = (s_{conflict}, overlap(s_{conflict})).$$

We use Tuple t to delay a conflict stream by applying the stream operation $o(s_{conflict}, overlap(s_{conflict}))$. We delay the conflict stream with the calculated overlap so that it queues after the late stream. As a result, the late stream is not blocked by the conflict stream and can be transmitted sooner. We execute this calculation for every stream in the stream set to get a set of tuples:

$$T = \{(s, overlap(s)) | s \text{ is a conflict stream}\}.$$

To determine the actual stream operation $o(s, overlap)$ being used for the node expansion in the search tree, we need to consider the set of tuples T. If we consider the set of conflict streams $\mathcal{S}_{conflict}$, some conflict streams interfere with only one late stream, while others can interfere with multiple late streams. For the stream operation $o(s, overlap)$, we choose the stream $s_{max_conflict} \in \mathcal{S}_{conflict}$ interfering with the highest number of late streams. Using $s_{max_conflict}$, we now create a subset of T ($t_i \in t$ denotes element i in Tuple t):

$$T_{max_conflict} = \{t | t \in T \wedge t_0 = s_{max_conflict}\}.$$

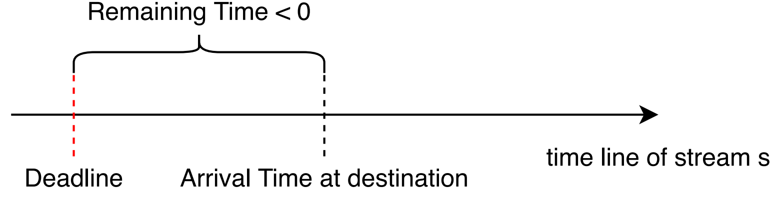


Figure 5.4: Stream s arrives at its destination after its deadline. Consequently, its remaining time is negative.

$T_{max_conflict}$ contains all overlaps for $s_{max_conflict}$. For the stream operation o, we choose the maximum overlap value in $T_{max_conflict}$: $o(s_{max_conflict}, overlap_{max})$). We delay the conflict stream by the largest overlap value, so that the conflict stream definitely queues after all late streams it interferes with.

Tree Traversal

By applying the calculated stream operation, we can expand nodes in the search tree. However, this approach still misses a strategy to find the best next node to expand. In that context, the best node is the node whose expansion is most likely to deliver a solution stream set $\mathcal{S}_{sol}$. We evaluate nodes in this regard by using the *Remaining Time* heuristic.

Definition 5.2.3
Given a Stream s, we define Remaining Time $rt(s)$ as the difference of its $Deadline(s)$ and its $Arrival\ time(s)$ at its destination.

Consequently, streams arriving after their deadline have a negative remaining time, cf. Figure 5.4. We assess stream sets by calculating the sum of remaining times in a stream set:

$$rts = \sum_{s_{late} \in \mathcal{S}} rt(s_{late}).$$

With reference to our definitions in Section 5.1, we consider stream set to be a solution, iff $rts = 0$, i.e., all Late streams s_{late} arrive at or before their deadline. Stream sets with late streams have a negative value for rts.

$$rts = 0 \iff \mathcal{S} \text{ is a solution stream set}$$

We detail the search process using the Remaining Time heuristic with the aid of an example, cf. Figure 5.5. We use the rts value of a stream set as a heuristic for tree traversal during the tree search. The search starts with $\mathcal{S}_0$. We calculate the rts value by simulating $\mathcal{S}_0$ with the tracing-based simulator. The tracing-based simulator generates a packet history for every stream. We use this packet history to calculate the rts value for $\mathcal{S}_0$. Afterwards, we expand this node with o_1 to get node $\mathcal{S}_1$. Again, we calculate the rts value with the tracing-based simulator. The rts value of $\mathcal{S}_1$ is smaller than the rts value of $\mathcal{S}_0$. Therefore, we decide that $\mathcal{S}_1$ is not a node whose expansion is going to deliver the solution stream set $\mathcal{S}_{sol}$. We backtrack to $\mathcal{S}_0$, calculate o_2 and apply it to $\mathcal{S}_0$, which yields $\mathcal{S}_2$. The rts value of $\mathcal{S}_2$ is larger than the rts value of $\mathcal{S}_0$. Therefore, we decide to expand $\mathcal{S}_2$ next. The expansion of $\mathcal{S}_2$, with o_3, yields $\mathcal{S}_3$. Again, its rts value is worse than its parents; therefore, we backtrack to $\mathcal{S}_2$.

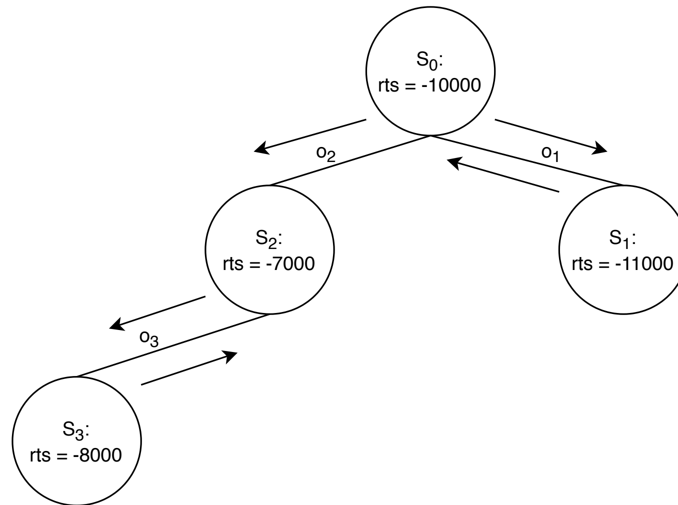


Figure 5.5: Tree visualizing the search process of the Link Time Remaining Time approach. Nodes are numbered by their expansion order.

Discussion

The Link Time Remaining Time approach is greedy. It never expands children with worse rts values than their parents. As a result, this approach might not find a solution, even if it exists. In essence, the Link Time Remaining Time approach tries to maximize the rts value of $\mathcal{S}_0$. However, maximizing rts does not fully translate to minimizing the number of late streams, and thus, finding $\mathcal{S}_{sol}$. Scenarios are possible, where maximizing rts increases the number of late streams. Nevertheless, this approach provides multiple benefits over the Naive approach. First, instead of only considering the paths of streams, the Link Time Remaining Time approach takes the actual queueing times of individual packets into account when calculating conflict streams. As a result, it only classifies streams that actually interfere with late streams as conflict streams. Additionally, this approach calculates an overlap time for each conflict stream. The overlap time represents the period for which a late stream queues behind a conflict stream. The overlap value provides a much better estimate for how much a conflict stream has to be delayed than simply delaying every conflict stream by 1,000 ns.

5.2.3 Link Time Late Streams

In this section, we show the Link Time Late Streams approach.

Stream Operation Calculation

The Link Time Late Streams approach utilizes the same Stream Operation Calculation method as the Link Time Remaining Time approach: it determines a Stream $s_{max_conflict}$, which interferes with the most late streams. It delays $s_{max_conflict}$ by $overlap_{max}$, the largest overlap value.

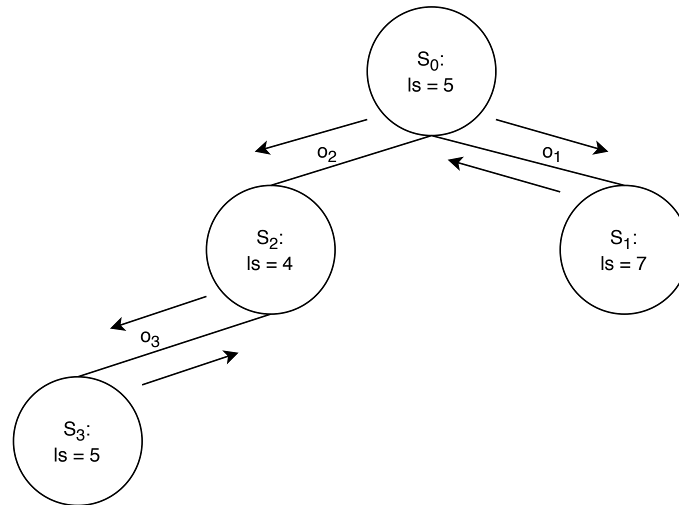


Figure 5.6: Tree visualizing the search process of the Link Time Late Stream approach. ls denotes the number of late streams in a stream set.

Tree Traversal

The Link Time Late Streams approach uses the same Tree Traversal procedure as the Link Time Remaining Time approach, except for the heuristic. It does not use the rts value of a stream set as a traversal heuristic, but uses the number of late streams in the stream set, cf. Figure 5.6. The Link Time Late Streams approach decides to backtrack, if the number of late streams increased in the expanded node. If it decreases, this approach does not backtrack and continues the search at the current node. If the number of late streams remains the same, this approach uses the rts values as a tie-breaker between two stream sets. In this case, this approach behaves like the Link Time Remaining Time approach.

Discussion

The Link Time Late Streams approach shares almost all of its properties with the Link Time Remaining Time approach. However, since it uses the number of late streams as an expansion heuristic, it minimizes the number of late streams, which is the relevant metric to finding a solution stream set. Therefore, we expect this approach to perform better than the Link Time Remaining Time approach.

5.2.4 Link Time Monte Carlo Tree Search

As shown in Section 2.4, MCTS operates differently then the Basic and Heuristic search algorithms we presented in Section 2.4 and developed in this section. Therefore, we utilize a different structure to present our Link Time MCTS approach. Having explained the basic MCTS process in Section 2.4, we structure this section by the separate steps performed in one iteration of the search. These steps are: *Selection*, *Expansion*, *Simulation* and *Backpropagation*. Afterward, we discuss the properties of this approach.

Selection

During the selection step, MCTS selects the most urgent node to expand. In every iteration, the MCTS starts traversing from the root node until it found the most urgent node. A utility function determines the most urgent node. It has to balance exploration (searching previously unexplored areas) and exploitation (searching promising areas). A utility function commonly used in MCTSs is Upper Confidence Bound 1 (UCB1) [ACF02], since it is simple and efficient [BPW+12]. Applying UCB1 to trees, as we do in this work, yields Upper Confidence Bound Trees (UCTs) [KS06]. The UCB1 function is defined as follows:

$$UCB1(j) = \underbrace{\overline{\Delta}_j}_{exploitation} + \alpha \cdot \underbrace{\sqrt{\frac{2 \cdot ln(n)}{n_j}}}_{exploration}.$$

$\overline{\Delta}_j$ denotes the average expected reward from Node j, n_j denotes the number of visits of Node j, n denotes the number of total iterations. To be able to use the UCB1 function, we expand our previous node Definition 5.1.3 in Section 5.1. A Node j in the search tree now consists of: a stream set $\mathcal{S}_j$, a reward Δ_j and the number of visits n_j of Node j. The Simulation subsection details how we calculate the reward for a node in the search tree. The left summand is the exploitation term, while the right summand is the exploration term. We can balance exploration against exploitation with different values of α. Low values for α favor $\overline{\Delta}_j$, and thus, nodes with a high expected reward. High values favor $\sqrt{\frac{2 \cdot ln(n)}{n_j}}$, and thus, nodes which have not yet been visited.

Expansion

After MCTS selects a node, it expands a leaf node from the selected node. With our approaches, we expand leaf nodes by applying a stream operation $o(s, x)$ to a stream set $\mathcal{S}$. In Section 5.2.2, we introduced the set of tuples:

$$T = \{(s, overlap(s)) | s \text{ is a conflict stream}\}$$

T denotes a set of $(stream\ operation, overlap)$ tuples for every conflict stream in stream set $\mathcal{S}$. Applying a Tuple $t \in T$ to $\mathcal{S}$ yields a child stream set. Our MCTS approach takes a uniformly random $t \in T$ and applies it to the selected node from the Selection step.

Simulation

The Expansion step yields stream set $\mathcal{S}_{new}$. During the Simulation step, the MCTS simulates the newly expanded node to calculate a reward Δ. We simulate $\mathcal{S}_{new}$ by using our previously developed Link Time Late Streams approach. We give $\mathcal{S}_{new}$ as an input to the Link Time Late Streams approach and execute the search for a given amount of time. After the search finished, we calculate the reward based on the best (least amount of late streams) stream set the search found:

$$\Delta = r(\text{number of late streams in } \mathcal{S}_{best})$$
$$r(x) = -1 \cdot (10 \cdot x)^2$$

The lower the number of late streams, the higher the reward, i.e. r is a strict monotonically decreasing function (for $x \in [0, \infty]$).

Backpropagation

During the Backpropagation step, the MCTS updates the node statistics of the path taken during the Selection step.

$$\Delta_{j'} = \Delta_j + \Delta$$
$$n_{j'} = n_j + 1$$

The reward value Δ_j of Node j is increased by the calculated reward value Δ, and the number of visits n_j is incremented by one. The UCB1 function uses both values to calculate the average reward $\overline{\Delta}_j$:

$$\overline{\Delta}_j = \frac{\Delta_j}{n_j}.$$

Discussion

MCTS iteratively generates an asymmetric search tree. It expands only the most promising areas of the search tree, according to the UCB1 utility function. The UCB1 function takes the expected reward of Node j and the number of visits of Node j into account. We calculate the reward of a node by applying our previously developed Link Time Late Streams approach. In contrast to the previously developed approaches, the Link Time MCTS approach is not greedy. Therefore, it does not exclude areas of the search tree, as the other approaches do. Consequently, we expect this approach to find solutions to stream sets, where the other approaches failed to find a solution.

6 Evaluation

In Chapter 5, we developed four scheduling approaches: Naive, Link Time Remaining Time, Link Time Late Streams, and Link Time MCTS. In this chapter, we evaluate the capabilities of these approaches. Before presenting the evaluation results, we describe our evaluation setup in order for the reader to understand the results. The evaluation setup includes a framework to automate the evaluation, a procedure to generate input data for our scheduling approaches, and our evaluation parameters. Thereafter, we present our evaluation results in Section 6.2.

6.1 Evaluation Setup

We use a benchmarking framework [SWHD20] to evaluate our scheduling approaches. The framework automates evaluation and, when evaluating multiple schedulers, ensures comparability of the results. The execution pipeline of the framework consists of four stages: *problem instance generation*, *scheduler execution*, *verification of results* and *evaluation of results*. We structure this section along the stages of the execution pipeline. We first provide an overview of the benchmarking framework and the execution pipeline. For our evaluations, we modified the *problem instance generation* stage of the execution pipeline. We describe these modifications in Section 6.1.2. The *scheduler execution* stage already comes with two integrated schedulers: the TSN SMT scheduler and the Job Shop Scheduling Problem (JSSP) scheduler. In our evaluation, we compare our scheduling approaches to these two schedulers, and for this reason, we present their features and properties in Section 6.1.3 and Section 6.1.4, respectively. Lastly, we detail our evaluation parameters and hardware.

6.1.1 Scheduling Benchmarking

As shown in Chapter 3, many scheduling algorithms have been proposed by different research groups. Almost every contribution includes an evaluation of the proposed algorithm. However, research groups predicate their evaluation on different assumptions and execute them in different environments. As a result, comparing evaluation results is very difficult. Nonetheless, comparing results is an essential component of the research process. Schneefuss et al. developed a benchmarking framework to find common ground for comparability and foster direct comparisons of different schedulers [SWHD20]. In this section, we give a brief overview of the benchmarking framework and its execution pipeline. The overview includes a description of its operation and a description of how we use it in our work. Further details are available in the corresponding paper [SWHD20].

The execution pipeline of the framework consists of four stages: *problem instance generation*, *scheduler execution*, *verification of results* and *evaluation of results*. In the following, we illustrate a run-through of the benchmarking pipeline and associate each pipeline step with a stage. Figure 6.1

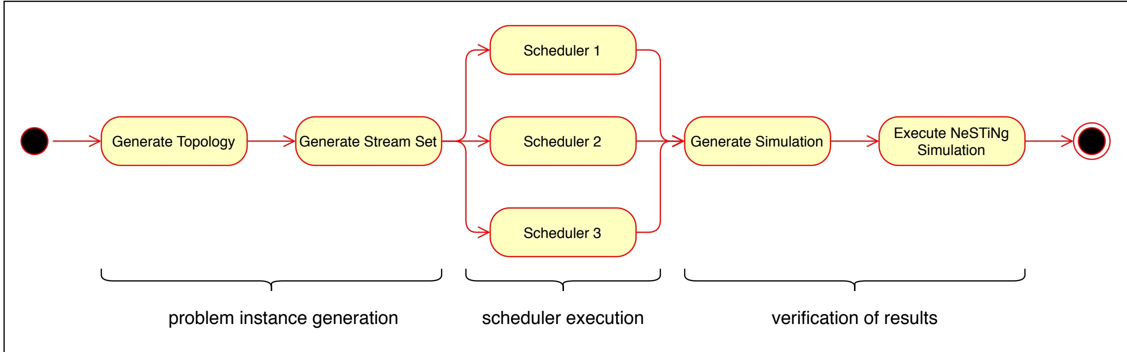


Figure 6.1: Activity diagram showing the pipeline structure and the corresponding stages of the benchmarking framework [SWHD20]. The *evaluation of results* stage is not depicted, since it is done manually, and therefore, not part of the activity diagram.

shows the activities of a run-through of the benchmarking pipeline. First, the pipeline generates multiple topologies. These topologies form the basis for the generation of stream sets. A stream set is associated with a specific topology. Thus, after generating topologies, the pipeline generates corresponding stream sets. However, randomly generated stream sets are not suitable for evaluating the scheduling capabilities of schedulers. The scheduler might not be able to find a solution, either because no solution exists or because the scheduler itself is not capable of finding one. In consequence, making a statement about the scheduler's capabilities is not possible. The framework generates stream sets that have a solution, i.e., are feasible. The *problem instance generation* stage gathers topology generation and feasible stream set generation. After the *problem instance generation* stage, the framework passes the generated data to every integrated scheduler. Each scheduler tries to calculate a schedule for the given problem instance. Schneefuss et al. refer to this step as the *scheduler execution* stage. Afterward, the framework generates and executes a NeSTiNg [FHC+19] simulation. This is the *verification of results* stage. The benchmarking framework provides a user interface to evaluate the simulation results in the *evaluation of results* stage.

6.1.2 Creating Feasible Stream Sets

The benchmarking framework generates feasible stream sets in the *problem instance generation* stage. Generating stream sets is not trivial: they have to follow specific requirements, i.e., number of streams in the stream set. More importantly, the stream sets need to have a solution, i.e., need to be feasible, to evaluate the scheduling capabilities of integrated schedulers properly. Assigning the properties listed in Section 4.5 randomly to each stream does not generally lead to a feasible stream set, e.g., if two streams have contradictory configurations. In this section, we develop a procedure to generate these feasible stream sets. Figure 6.2 shows the activity diagram for our feasible generation procedure. In the following, we detail each activity in the diagram.

The *feasible generator* gets the following inputs:

$\mathcal{T}$: A stream set $\mathcal{S}$ is always associated with a specific topology $\mathcal{T}$, therefore $\mathcal{T}$ is required as an input.

T_{cycle}: Indicates the cycle time of a stream, cf. Section 4.5 for further details.

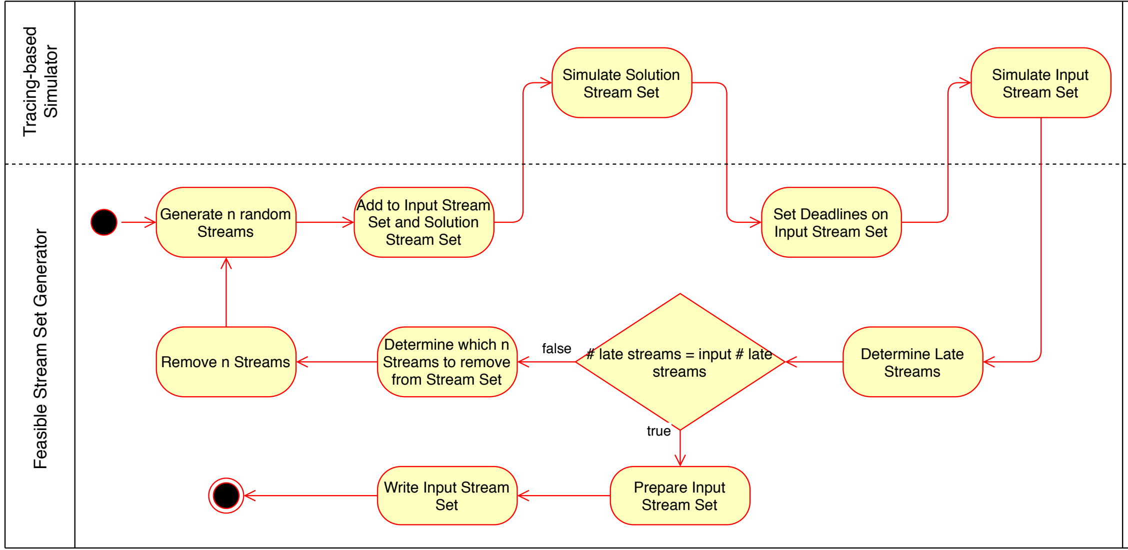


Figure 6.2: Activity diagram showing the procedure *feasible generator* that generates feasible stream sets.

$|S| = n$: Denotes the stream set size.

n_{init_late}: Indicates the initial number of late streams. We elaborate on the function of n_{init_late} later in this section.

The *feasible generator* starts by generating the required number of streams. It generates these streams randomly - i.e., their source, destination, and size are set randomly within the specifications of our Traffic Model. For the $Path$ property of a stream, we calculate the shortest path between a stream's source and destination. The *start times* of the streams are set randomly within the interval $[lower, upper]$. We set $lower = 0$, i.e., a host can transmit a stream as soon as the network cycle starts. For $upper$, we calculate the end-to-end delay of the stream. We set $upper$ to $T_{cycle} - e2e\ delay$, thus, the stream arrives before the network cycle ends.

After generating n streams, the feasible generator adds them to two sets: the Solution Stream Set S_{sol} and the Input Stream Set S_{input}. S_{sol} holds all streams in their feasible configuration - the stream parameters are set such that all deadlines are honored. S_{input} holds all streams with the *start times* set to zero. S_{input} later acts as the input S_0 to our scheduling approaches.

After adding the generated streams to the aforementioned stream sets, the *feasible generator* invokes the tracing-based simulator on S_{sol}. Afterward, the *feasible generator* obtains the arrival times of all streams from the simulation results. We derive the deadlines of the streams from their arrival times calculated by the simulation. As a result, S_{sol} is feasible by design. This is where both stream sets differ. S_{sol} holds the streams in their feasible configuration. In S_{input}, the streams' start times are all set to zero. The different start times of the streams in S_{input} might lead to queueing and blocking effects between streams, which the deadlines do not consider. Consequently, streams could arrive after their deadline. We consider S_{sol} to be a solution for S_{input}.

To determine how many streams arrive too late, we simulate $\mathcal{S}_{input}$ with the tracing-based simulator. We check the event history of every stream and do the following calculation:

$$\forall s \in \mathcal{S}_{input} : Deadline(s) - arrival\ time(s) < 0 \implies s \in \mathcal{S}_{late}.$$

Now the input parameter n_{init_late} comes into play. It determines how many late streams $\mathcal{S}_{input}$ should have. In other words:

$$n_{init_late} = |\mathcal{S}_{late}|.$$

If $n_{init_late} < |\mathcal{S}_{late}|$, we choose a late stream randomly. If $n_{init_late} > |\mathcal{S}_{late}|$, we choose a punctual stream randomly. The generation procedure then removes the selected stream from both stream sets. Afterward, it repeats the process, i.e., it randomly generates the missing streams and adds them to both stream sets. If the current value of late streams is equal to our desired value, the *feasible generator* is finished. To summarize, the *feasible generator* generates a stream set $\mathcal{S}_{sol}$ holding the feasible configuration of all streams, and a stream set $\mathcal{S}_{input}$ holding the input for our schedulers.

We integrate our *feasible generator* into the *problem instance generation* stage of the benchmarking framework. As a result, the benchmarking framework generates feasible stream sets according to the procedure we developed. All schedulers integrated into the framework get the same input topologies and stream sets. This allows direct comparisons between the integrated schedulers. Therefore, we integrate all four scheduling approaches we developed in Chapter 5 into the execution pipeline. Additionally, the scheduling approach of Duerr et al. [DN16] and a scheduler implementing the scheduling constraints proposed in [Ste10] are already pre-integrated into the benchmarking framework. For our evaluation, we use both state-of-the-art schedulers to compare our scheduling approaches. To understand the implications of the evaluation results, it is necessary to understand these schedulers' properties. In the following, we provide quick overviews of both schedulers.

6.1.3 Job-Shop Scheduling Problem (JSSP) Scheduler

In this section, we present the properties and features of the JSSP scheduler. At first, we provide a quick recap of the scientific background of the JSSP scheduler. Afterward, we describe its features and properties.

As already touched on in Section 3.2.1, Duerr et al. [DN16] consider the problem of calculating a TSN schedule for NICs and switches in a TSN network. They map the (No-Wait) Job-Shop Scheduling problem (NW-JSSP) to a (No-Wait) Packet Scheduling (NW-PSP) problem. As a result, they can calculate TSN schedules, which minimize the makespan. Duerr et al. define the makespan of a schedule as the difference between the time the last stream arrives at its destination and the time the first stream is transmitted at its source. The authors formulate instances of NW-PSP as ILP constraints. Additionally to their complete ILP approach, Duerr et al. [DN16] introduce a heuristic approach for solving instances of NW-PSP. They use tabu-search as a meta-heuristic to accelerate schedule calculation. In the following, when we refer to the JSSP scheduler, we refer to the scheduler that employs the heuristic approach to calculate NW-PSP schedules. The typical variant of the Job-Shop Problem allows queueing of jobs on machines, which the authors do not desire. Therefore, they introduce the No-Wait variant of the Job-Shop problem: after a job is started, it cannot be interrupted. This translates to packet scheduling: in NW-PSP, packets do not

Topology Parameter	Values
Topology Size (Number of Switches)	104, 504, 1008
Link Speed	1 Gbps
Propagation Delay	200 ns ≃ 40 m link length
Processing Delay (of Switches)	2 µs
Number of Replicates	10

Table 6.1: This table shows the parameters we use for generating benchmark topologies.

experience queueing. Thus, the JSSP scheduler can only find schedules that exhibit no queueing between streams. The stream sets generated by our *feasible generation procedure* have a solution stream set $\mathcal{S}_{sol}$. However generally, the streams experience queueing in $\mathcal{S}_{sol}$. This might make calculating a schedule more difficult for the JSSP scheduler since a stream set with queueing might only be schedulable if the scheduler considers queueing. Additionally, the JSSP scheduler does not take stream deadlines $Deadline(s)$ into account. It merely tries to place every stream in the network cycle T_{cycle}, such that no queueing occurs. The JSSP scheduler considers scheduling as successful if it places all streams in a schedule, and the makespan of the schedule is smaller than the network cycle: $makespan(schedule) < T_{cycle}$. The JSSP scheduler offers two different routing options: *shortest path routing* and *precalculated routing*. When using *shortest path routing*, the JSSP scheduler calculates the shortest routes itself, in *precalculated routing*, it uses the routes supplied by the input stream set with the property $Path(s)$. We evaluate both routing options in this work.

6.1.4 TSN SMT Scheduler

In this section, we present the TSN SMT scheduler, which the benchmarking framework integrates. We briefly show its features and properties.

In [Ste10], Steiner introduces scheduling constraints formulated in Satisfiability Modulo Theories (SMT). Problems formulated in SMT are decision problems, an SMT solver decides if the constraints are satisfiable or not [BT18]. The scheduler integrated into the benchmarking framework implements the constraints proposed in [Ste10] and uses the z3 solver [MB08] to solve the constraints [SWHD20]. Like the JSSP scheduler, the TSN SMT scheduler offers two routing options as well: *shortest path routing* and *precalculated path routing*. However, in previous work [SWHD20], the TSN SMT scheduler did not yield enough results in shortest path mode; therefore, we exclude this mode from our evaluation.

6.1.5 Evaluation Parameters

In Chapter 4, we introduced our System Model. The System Model includes a network model and a traffic model. The models lacked concrete specifications for any variables since our scheduling approaches are agnostic to specific traffic or topology parameters. In this section, we present our evaluation parameters. We determine the parameters of the topologies we generate, as well as stream set parameters. Furthermore, we elaborate on the hardware we use in our evaluation.

Stream Parameter	Values
Stream Set Size	100, 500, 1000
Payload Sizes	125 B - 625 B in 125 B steps
Cycle Time T_{cycle}	1 ms
Number of Initial Late Streams	1, 2, 3, 5, 7, 10, 13, 17
Number of Replicas	1 and 2

Table 6.2: This table shows the parameters we use for generating stream sets.

Table 6.1 shows the topology parameters we use in our evaluation. We generate topologies with 104, 504, and 1008 switches to cover a wide range of topologies. The generated topologies have the factory layout introduced in Chapter 4. We determine the size of a topology by the number of switches the topology contains since the switches implement the TSN logic and, therefore, dictate the network's complexity. We generate ten replicas of every parameter combination. In total, we evaluate 30 topologies.

Table 6.2 shows the stream parameters for our evaluation. We generate stream sets with 100, 500, and 1000 streams to cover a wide range of stream sets. Our preliminary tests showed that the TSN SMT and the JSSP schedulers struggled to solve large stream sets in a reasonable amount of time. Therefore, we limited the maximum stream set size to 1000 streams. In the white paper [APB+18], the IIC determines the size of isochronous packets to be 30 B - 100 B. However, practitioners are interested in transmitting packets isochronously up to 600 B in size. Therefore, we evaluate payload sizes from 125 B to 625 B in 125 B steps. In Section 6.1.2, we presented our feasible generation procedure. It requires the number of initial late streams n_{init_late} in stream set $\mathcal{S}_{input}$ as an input value. The larger this parameter, the more conflict streams exist in a stream set, and ultimately, the larger the scheduling search tree becomes. Therefore, we expect this parameter to have a significant influence on the search time of our approaches. To reduce the number of parameter combinations, we sample lower values of n_{init_late} more often and decrease the resolution for higher values. We initially planned to generate two replicas for every parameter combination. However, due to time constraints, we generated two replicas for three topologies and one replica for the remaining 27 topologies. This amounts to:

$$
\begin{aligned}
&3\ stream\ set\ sizes \cdot 8\ n_{init_late}\ values \cdot 27\ topologies\\
&+ 3\ stream\ set\ sizes \cdot 8\ n_{init_late}\ values \cdot 2\ replicas \cdot 3\ topologies\\
&= 792
\end{aligned}
$$

evaluation runs for every scheduler.

We execute the evaluation on a cluster consisting of four nodes. Each node is assigned to one scheduler, and executes evaluation runs exclusively for it. Thus, schedulers do not compete for resources, and runtime measurements are comparable between them. To execute the large number of evaluations, we set a time limit for our scheduling approaches: 1,200 s. After 1,200 s, the scheduler is stopped, regardless of if it found a solution. It was not possible to specify a time limit for the TSN SMT and JSSP schedulers. Therefore, we had to enforce reasonable time limits manually. Table 6.3 shows the hardware of the nodes in the evaluation cluster. All schedulers run single-threaded.

CPU	Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz
Memory	4 x 8GiB DIMM DDR4 Synchronous 2666 MHz (0.4 ns)
OS	Ubuntu 18.04.4 LTS
Kernel	4.15.0-111-generic

Table 6.3: Hardware of the evaluation cluster nodes.

6.2 Results

In this section, we present the evaluation results gathered with the presented benchmarking framework. We structure this section by the properties along which we evaluate the scheduling approaches. Some schedulers were able to execute all input sets, while others were not. The execution rate denotes the ratio of executed schedule calculations over started schedule calculations. We first present the execution rates of the schedulers in Section 6.2.1. Another essential property of a scheduler is its runtime behavior. It decides whether a scheduler is suitable for practical applications. We present the runtime behavior of every scheduler in Section 6.2.2. As important as the runtime behavior are the actual scheduling capabilities. Ideally, a scheduler should be able to solve every problem instance to which a solution exists. We show the scheduling capabilities of the schedulers in Section 6.2.3. During our evaluation, we found the Link Time MCTS approach to be the most promising approach. Therefore, we ran additional evaluations with it. We elaborate on the results of these additional evaluations in the dedicated Section 6.2.4. In the subsequent sections, we refer to the scheduling approaches developed in this work as Tracing-based Schedulers (TBS).

6.2.1 Execution Rates

Starting a scheduling algorithm on a problem instance does not necessarily yield a result. We refer to a scheduler execution as an instance of a schedule calculation on an input problem instance. A practical scheduling approach can execute all problem instances. In order for practitioners to use a scheduling algorithm, it has to be practical. We define a scheduler as practical if:

1. It calculates a solution in less than 48 hours. If a scheduler does not terminate within that given time, we consider the execution as failed. Schedules often require re-calculation in practical applications since the configurations of the transmitted data streams are subject to changes. Thus, the schedule calculation needs to be as fast as possible for an approach to be feasible in practical applications.
2. Its memory requirements do not exceed 32 GB. Schedule execution can fail if the hardware requirements of the scheduler exceed the provided capabilities. Failure of execution is not desirable and indicates that the scheduler's hardware requirements might be too high. Therefore, low hardware requirements are desired. Generally, practitioners do not have access to powerful computing clusters. We consider a machine with 32 GB as accessible to most practitioners.

The evaluated schedulers show differences regarding the presented criteria. The execution rate denotes the ratio of executed schedule calculations over started schedule calculations. In the following, we present the execution rates for all schedulers.

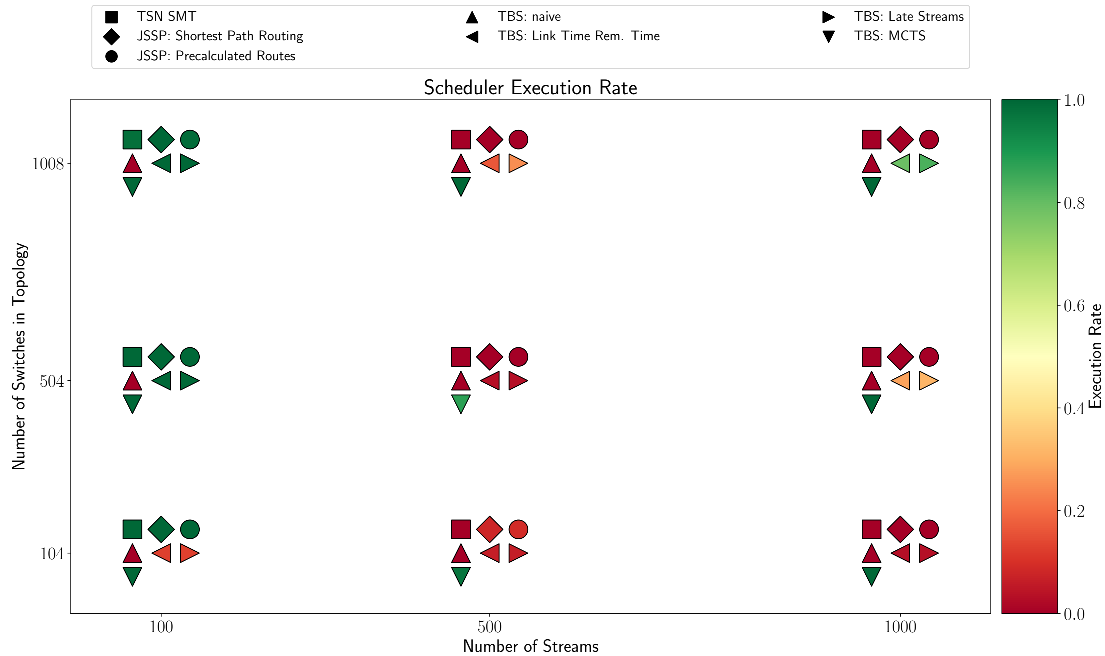


Figure 6.3: Graph showing the execution rates for all evaluated schedulers. The TSN SMT and JSSP schedulers executed scheduling for smaller stream set sizes only. The TBS schedulers (except for the MCTS approach) show low execution rates, regardless of the input configuration. Only the MCTS approach did execute all problem instances.

Figure 6.3 shows the execution rates for all evaluated schedulers. We analyze the execution rates along two dimensions: the size of the evaluated stream set and the topology size. The y-axis shows the topology sizes; the x-axis shows the stream set sizes. These parameters are the two main factors that drive the complexity of the scheduling problem (cf. [Ste10], [CO14]), and therefore, the hardware requirements. The marker type denotes the scheduler; the marker color indicates the execution rate. A green marker indicates a high execution rate, and a red marker indicates a low execution rate.

The TSN SMT scheduler executes scheduling for the smallest stream set size of 100 streams. However, scheduling larger stream sets leads to problems: the TSN SMT scheduler filled the RAM of the cluster node to the point where it became unresponsive. As a result, we abandoned the evaluation of the TSN SMT scheduler on larger stream sets. Figure 6.3 shows this, the execution rate is zero for stream sets larger than 100 streams. We saw a similar decrease in execution rates for both JSSP scheduler variants; however, different factors cause it. The execution of stream sets larger than 100 streams leads to non-polynomial increases in execution times. Many scheduler executions exceeded the time limit of 48 hours we set earlier. Since evaluating the larger stream sets is impossible in a reasonable amount of time, we abandoned the evaluation of stream sets with 1000 streams with the JSSP scheduler. The TBS Naive approach was not able to execute a single stream set without failure. Memory overruns cause the failures: the Naive approach generates a large search tree quickly, regardless of topology or stream set size. This filled the memory of the evaluation node up to the point where the Linux memory manager killed the associated process. We

see similar behavior for the Link Time Remaining Time and Link Time Late streams approaches. They filled the memory up as well, which resulted in low execution rates. Only the MCTS approach shows high execution rates across all topology and stream set sizes.

Our analysis of the execution rates showed that the TSN SMT and JSSP schedulers were not able to execute scheduling for large stream sets. Generally, we assume that practitioners wish to schedule stream sets with more than 100 streams, i.e., in Internet of Things applications where isochronous transmission of large amounts of time-sensitive data is desired. Scheduling such large stream sets is not possible with the two mentioned contemporary schedulers. This fact clearly shows the need for scheduling approaches, which can handle large stream sets. The Naive, Link Time Remaining Time, and Link Time Late Streams approaches showed low execution rates due to large memory consumption. In contrast, the MCTS approach was able to execute all input configurations. Thus, the MCTS approach consumed less memory than the other approaches. As of writing, the benchmarking framework cannot enforce memory consumption limits. Despite the high memory requirements of the Naive, Link Time Remaining Time, and Link Time Late Streams approaches, we are still interested in their scheduling capabilities. We leave the scheduling time limit for all TBS approaches at 1,200 s, to retain comparability between them. If a TBS approach does not find a solution within 1,200 s, we consider the scheduling as unsuccessful. If a TBS approach runs out of memory, it did not find a solution before consuming all memory - in this case, we consider scheduling unsuccessful as well. If a scheduler finds a solution within 1,200 s, it terminates immediately and passes the solution to the benchmarking framework for further evaluation. Leaving the time limit for all schedulers at 1,200 s allows them to use all available resources to find a solution, without enforcing a lower time limit to lower memory consumption. This gives each scheduler the best chance to find a solution.

In this chapter, we touched on time limits and runtimes of schedulers. As previously stated, scheduling runtimes are an essential property of schedulers, especially for determining their practicality. In the following section, we analyze the runtimes of each scheduler.

6.2.2 Scheduling Runtimes

In the previous analysis, the schedulers exhibited different execution rates depending on the input configuration. Knowing the scheduling runtimes and how they are influenced is essential since they decide whether an approach is practical. In this section, we give a comprehensive review of the runtime properties of each evaluated scheduler. Like in Section 6.2.1, we analyze the runtimes along two parameters: stream set size and topology size.

A successful schedule calculation is one where the scheduler finds a schedule, which fulfills all stream requirements. Figure 6.4 shows the runtimes of successful schedule calculations as box-plots. The x-axis of each box-plot shows the stream set size; the y-axis shows the scheduling runtimes. Note the different y-axis scales of the TBS and the TSN SMT and JSSP schedulers. The Naive approach was unable to calculate a single schedule successfully, therefore, we exclude it from the runtime evaluation. The runtimes are grouped by the stream set sizes. The TSN SMT scheduler exhibits runtimes of up to 80,000 s for the smallest evaluated stream sets of 100 streams. As mentioned previously, we abandoned the evaluation of larger stream sets. Runtimes vary greatly and range from 10 s to 80,000 s. The stream set size dictates the scheduling runtimes of the JSSP schedulers. Scheduling the smallest stream sets generally takes less than 5,000 s; however, the runtimes increase

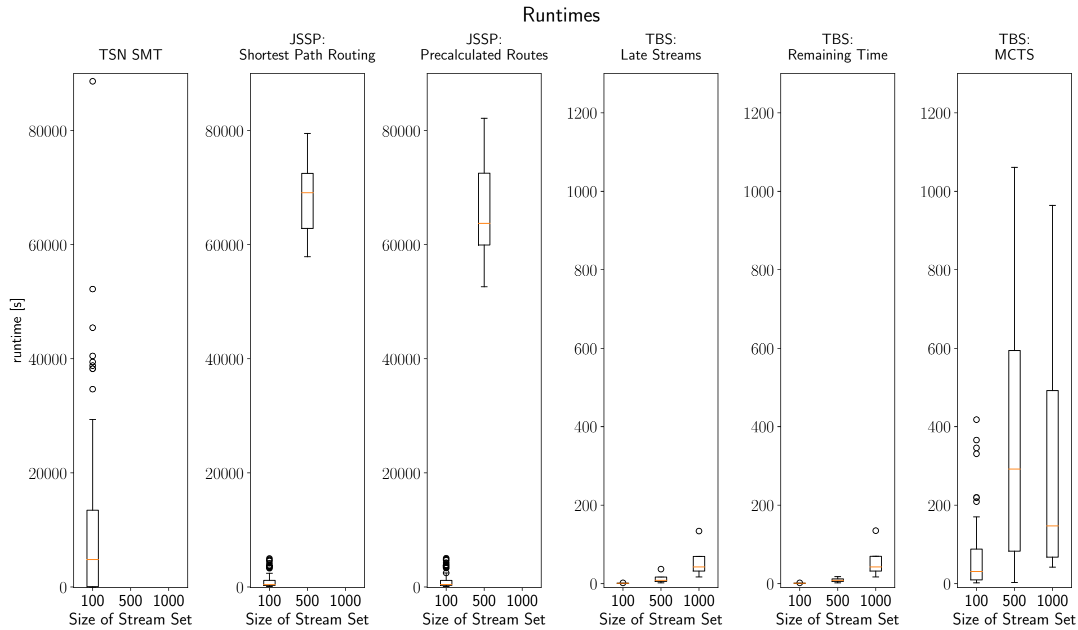


Figure 6.4: Boxplots showing the runtime distribution for the evaluated schedulers, grouped by the stream set sizes.

drastically up to 70,000 s for stream sets with 500 streams. The TBS approaches show very low runtimes (< 100 s) for small stream sets. The runtimes increase for larger stream set sizes, however, only slightly compared to the JSSP schedulers. The TBS approaches solve the largest stream sets with 1000 streams in less than 1,000 s. The MCTS approach shows great variation in runtimes compared to the other TBS approaches. Its scheduling capabilities cause this. It can schedule more stream sets successfully than the other TBS approaches, showing greater variations in runtimes only because it solved more stream sets. We show a detailed review of the scheduling capabilities of each scheduler in Section 6.2.3.

Figure 6.5 shows the runtimes grouped by the topology sizes. In our evaluations, the runtime of the TSN SMT scheduler is dependent on the size of the evaluated topology. The smallest topologies with 104 switches are evaluated in under 1,000 s. The runtimes increase non-linearly for larger topologies. Note that we evaluated the TSN SMT scheduler only on stream sets with 100 streams. Nevertheless, runtimes increase to 80,000 s for topologies with 1008 switches. We conclude that the influence of the topology size on runtime is the cause for the variance in the runtime of the TSN SMT scheduler in Figure 6.4. The JSSP schedulers do not show a correlation between runtime and topology size. The outliers seen in the box-plots are evaluations with large stream set sizes, cf. Figure 6.4. The TBS approaches do not display a correlation between topology size and runtime as well.

The runtimes of the TSN SMT scheduler are influenced by the topology size. Evaluating small topologies and stream sets is very fast. However, evaluation of the largest topologies with 1008 switches took up to 80,000 s. This non-polynomial increase in runtime clearly shows the NP-hardness of the scheduling problem and the limitations of complete approaches. The topology size does not influence the JSSP scheduler as heavily as the TSN SMT scheduler. However, evaluating stream sets

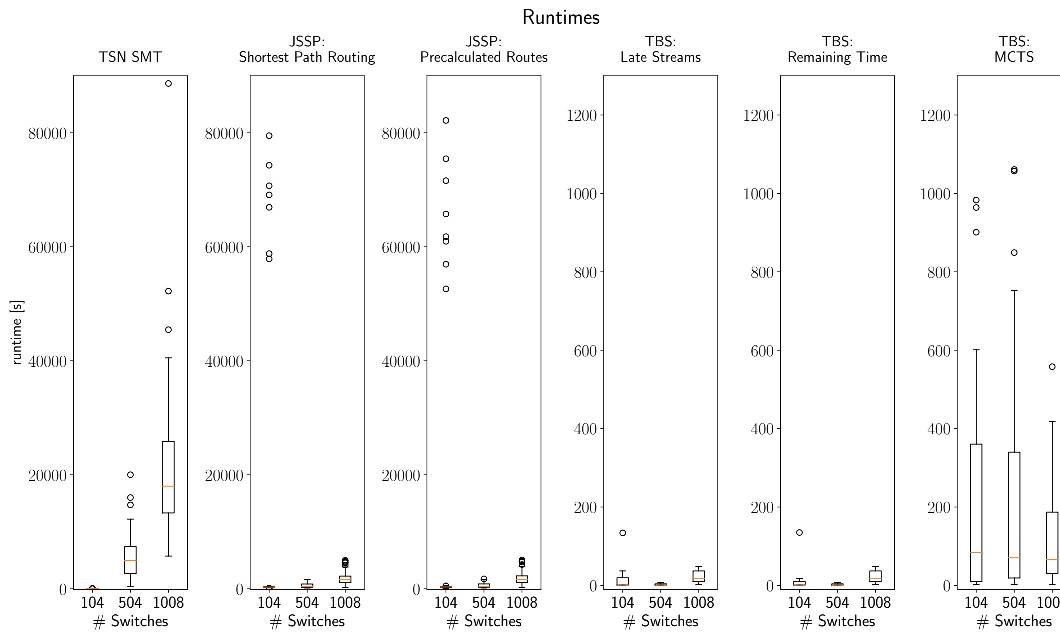


Figure 6.5: Box-plots showing the runtime distribution for the evaluated schedulers, grouped by the size of the topologies.

with 500 streams takes up to 80,000 s. Again, this demonstrates the limitations of contemporary schedulers. The stream set size influences the runtimes of the TBS approaches, however, not as drastically as the runtimes of the TSN SMT and JSSP schedulers. The TBS approaches schedule small stream sets with 100 streams in less than 100 s, large stream sets with 1000 streams in less than 1,000 s.

Merely executing the scheduling is not sufficient. A scheduler should be able to successfully schedule a stream set, i.e., calculate a solution. In the subsequent section, we analyze the scheduling capabilities of the evaluated schedulers.

6.2.3 Scheduling Capabilities

If a scheduler terminates, it has either found a solution or not. The scheduling capabilities of a scheduler are an essential factor for practitioners when deciding which scheduler to use. A well-performing scheduler should be able to find solutions to as many stream sets as possible. In this section, we detail the solving capabilities of each scheduler. We start by describing the visualization we chose to display the solving capabilities. Afterward, we give an overview of each scheduling approach's general properties by visualizing the solving capabilities over the aggregated stream set and topology sizes. Next, we filter the evaluation data to give a more detailed review of each scheduler. Lastly, we provide a summary of the scheduling capabilities.

Figure 6.6 shows the ratio of solved stream sets for every scheduler. The x-axis denotes the scheduling runtime. Note that the x-axis scale is different for every plot. At the start of the schedule calculation, no stream set is solved; therefore, all graphs start at the coordinate origin. As calculation time

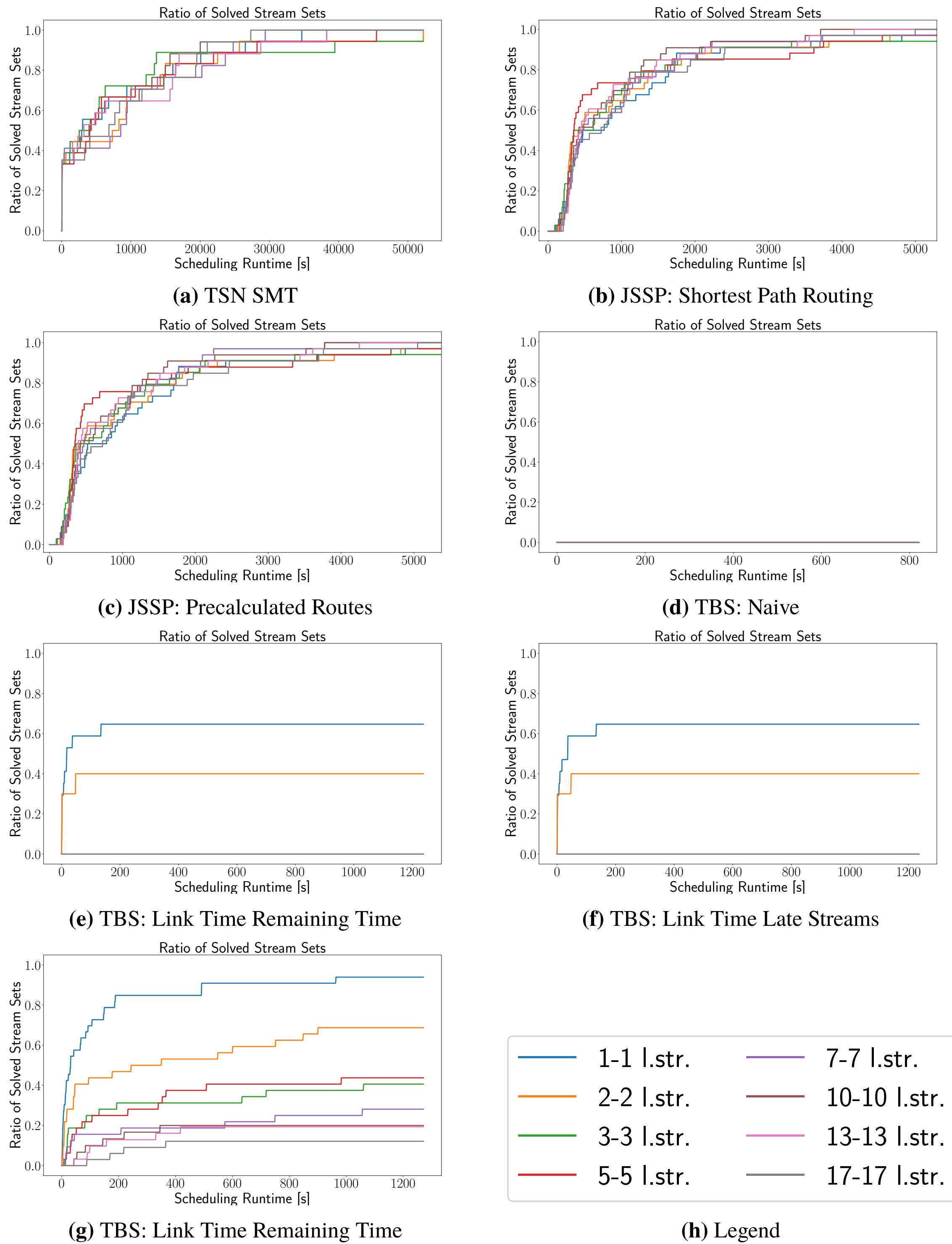


Figure 6.6: Graphs showing the ratio of solved stream sets over scheduling time. Note the different x-axis scaling.

passes, more and more stream sets are solved, the ratio of solved stream sets to total stream sets increases, the graphs start to converge towards a ratio of one. Ideally, a scheduler would solve all stream sets as fast as possible. n_{init_late} denotes the initial number of late streams in a stream set, cf. Section 6.1.2 for details on n_{init_late}. Each line represents the ratio of solved stream sets with a particular initial number of late streams n_{init_late}.

The solving ratios of the TSN SMT and JSSP schedulers are not influenced by n_{init_late}, which can be seen by the simultaneous increase in solved stream sets for all n_{init_late}, cf. Figure 6.6a, Figure 6.6b and Figure 6.6c. This is expected since those scheduling procedures do not use n_{init_late}. Conversely, the TBS approaches are influenced by n_{init_late}, cf. Figure 6.6d - Figure 6.6g. The higher n_{init_late}, the slower the ratio of solved stream sets increases. For larger n_{init_late}, the solving time for the TBS approaches increases, which is expressed by a lower ratio of solved stream sets at large scheduling runtimes. We explain this behavior with the search trees the TBS approaches generate: the larger n_{init_late}, the more conflict streams exist in the stream set. Consequently, more conflict streams have to be delayed to not interfere with late streams. Ultimately, the search tree becomes larger, leading to longer search times and longer scheduling times. The TBS Naive approach is the worst performing approach. It is not able to schedule a single stream set, cf. Figure 6.6d. We expected this behavior in Section 5.2.1, since its heuristics are very coarse. The TSN SMT and JSSP schedulers schedule all evaluated stream sets eventually, however, can take very long to do so. The TBS Link Time Remaining Time and TBS Late Streams approaches are able to schedule the majority of stream sets with $n_{init_late} = 1$ and $n_{init_late} = 2$ in the given runtime of 1,200 s. However, they were not able to find a single solution for larger values of n_{init_late}. The cause of this behavior is two-fold. These scheduling approaches utilize heuristics and are greedy. Because of their greediness, they only expand nodes in the search tree, which improve the heuristic. A solution stream set $\mathcal{S}_{sol}$ might be situated in a subtree, which is never expanded by the greedy search. Additionally, we believe another property of the tree search to be a cause, cf. Figure 6.8a. In Chapter 5, we presented our formulation of the scheduling problem as a search problem. To yield a child node, we apply a stream operation o on a stream set $\mathcal{S}$. The set of possible stream operations determines the number of child nodes and the branching factor of the search tree. We determine the possible stream operations by finding the conflict streams $s_{conflict} \in \mathcal{S}$. The conflict streams interfere with the late streams $s_{late} \in \mathcal{S}$. As a result, the number of conflict streams is directly dependent on the number of late streams. We restrict the possible stream operations to conflict streams. Thus, if a stream set has a low amount of late streams, the amount of conflict streams is low as well, and thus, the set of possible stream operations is small. A low number of stream operations results in a low branching factor of the search tree. A tree with a low branching factor is much easier to search than one with a high branching factor. If n_{init_late} increases, the branching factor of the search tree increases accordingly. The TBS Link Time Remaining Time and TBS Late Streams approaches expand every child node to find one, which improves the heuristic. With a high branching factor, these approaches have to expand a large number of child nodes before deciding to backtrack. Consequently, the TBS Link Time Remaining Time and TBS Late Streams approaches get stuck searching parts of the tree which do not contain a solution, cf. Figure 6.8a. Comparing these approaches with the TBS Link Time MCTS scheduler, we can see significant differences. The TBS Link Time MCTS scheduler is able to find a schedule for 90% of stream sets with $n_{init_late} = 1$. With increasing n_{init_late}, the ratio of solved stream sets drops as well, however, not as drastically as for the other approaches. This difference can, again, be explained with the respective search trees, cf. Figure 6.8. The MCTS approach expands the search tree asymmetrically,

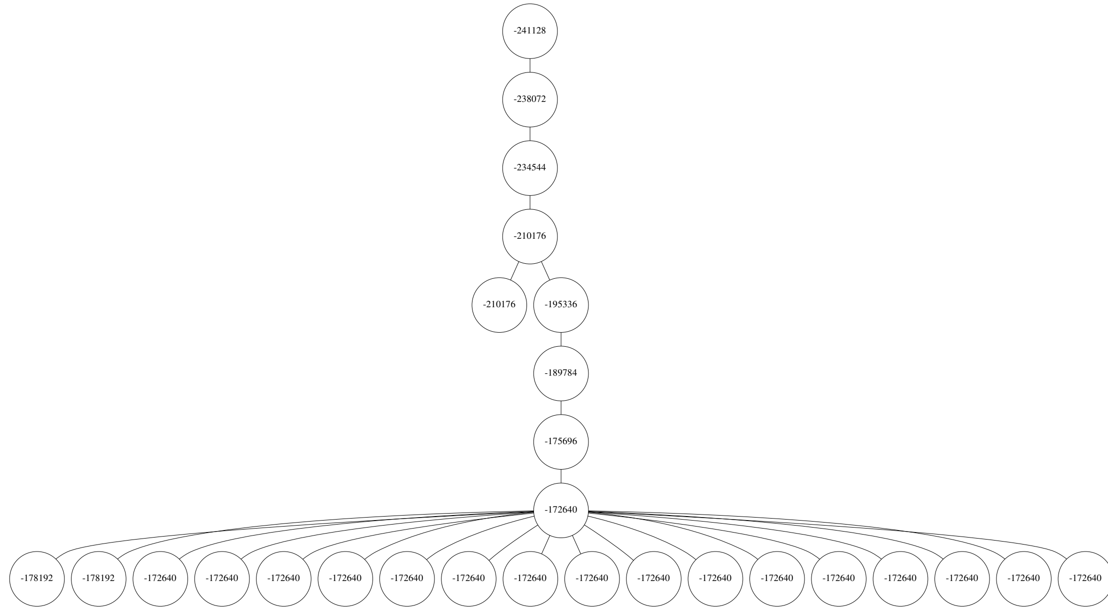
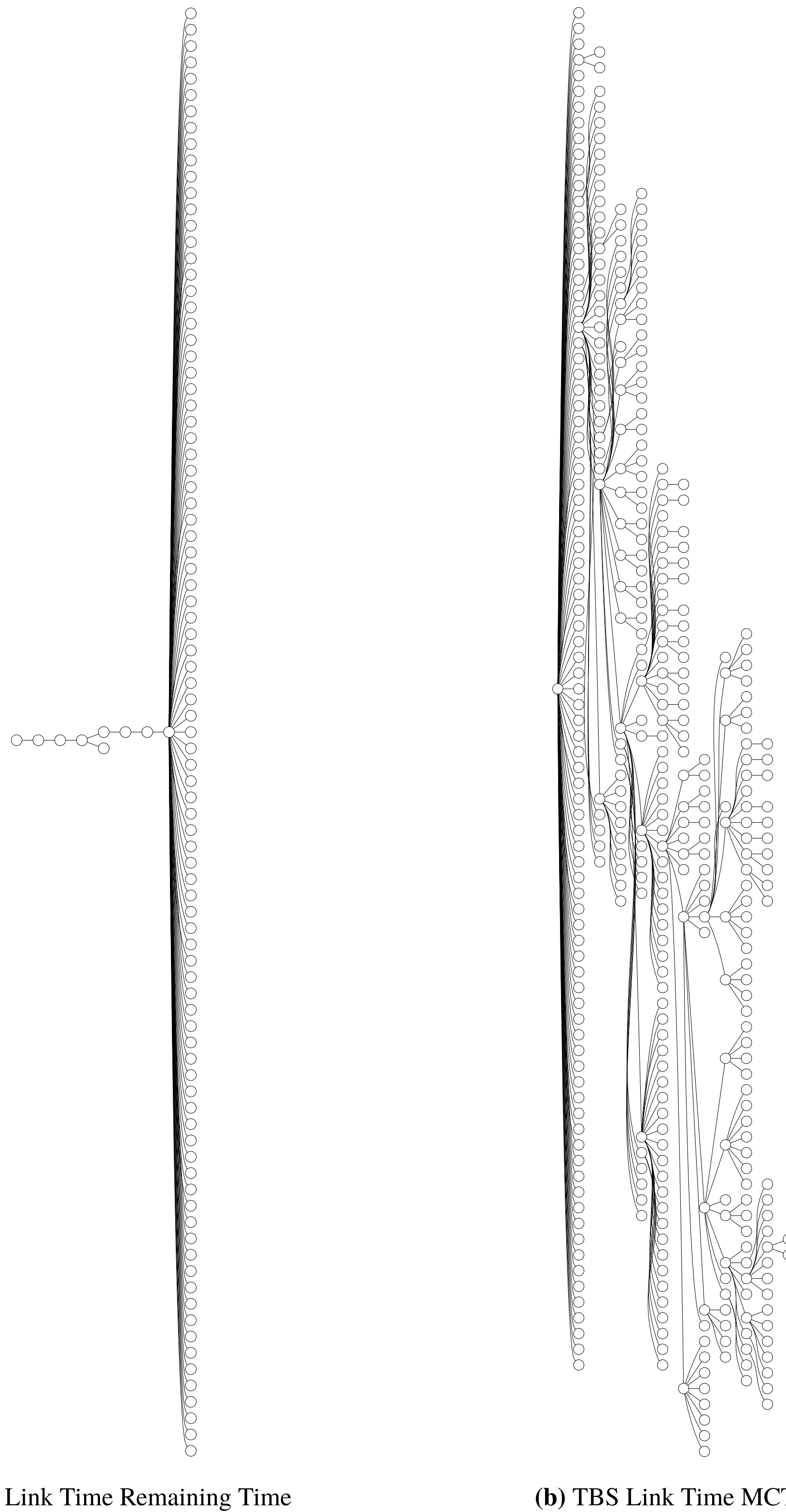


Figure 6.7: Search tree generated by the Link Time Remaining Time approach. The node labels are the rts values defined in Section 5.2.2. Before the search can backtrack, it has to expand every child node, to check if there exists one with a better rts value. As a result, the search can get stuck expanding a large number of child nodes.

cf. Figure 6.8b. It always searches the most promising areas of the tree, and as a result, does not get stuck, even with a high branching factor caused by a large n_{init_late}.

After giving a general overview, we present detailed analyses of the scheduling capabilities of the TBS approaches. We exclude the Naive approach from the following analysis since it did not solve any stream sets. Figure 6.9 shows the ratio of solved stream sets for the TBS Link Time Remaining Time approach for different stream set sizes. Generally, this approach can schedule stream sets with a low number of initial late streams n_{init_late}. It is not able to schedule stream sets with $n_{init_late} > 2$. This holds for all evaluated stream set sizes. The TBS Link Time Remaining Time approach exhibits identical behavior, cf. Figure 6.10.

In contrast to the previous TBS schedulers, the TBS Link Time MCTS scheduler shows improved scheduling capabilities, cf. Figure 6.11. It generally solves more stream sets across all input combinations than the previous TBS approaches. For small stream sets with 100 streams, this scheduler solves every stream set with $n_{init_late} = 1$. For larger n_{init_late}, the ratio of solved stream sets degrades, however, not as fast as for the other TBS approaches. Nearly 40% of stream sets with $n_{init_late} = 17$ are solved in the given runtime of 1,200 s, which is a major improvement to other TBS schedulers. Analyzing the results for stream sets with 500 streams, we see similar behavior. For low values of n_{init_late}, the TBS Link Time MCTS approach can solve the overwhelming majority of stream sets. However, with increasing n_{init_late}, the ratio of solved stream sets degrades faster than for stream sets with 100 streams. This behavior is expedited for the largest stream sets with 1000 streams. For stream sets with 500 and 1000 streams and with $n_{init_late} = 17$, this approach cannot find a solution in the given runtime. We explain the degradation of the ratio of solved stream sets for larger stream sets with the tracing-based simulator's properties. Larger stream sets simply take



(a) TBS Link Time Remaining Time

(b) TBS Link Time MCTS

Figure 6.8: Comparing the search trees generated by the TBS Link Time Remaining Time Scheduler and the TBS Link Time MCTS Scheduler. The MCTS approach builds the search tree asymmetrically, while the Link Time Remaining Time approach expands a single node.

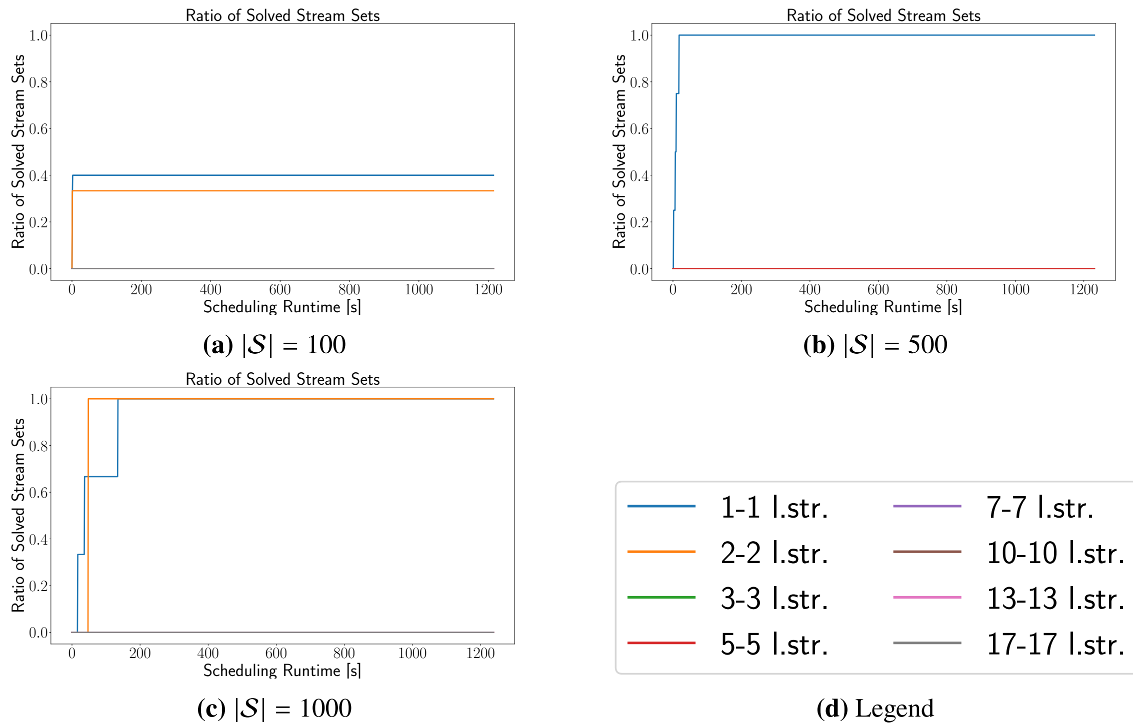


Figure 6.9: Ratio of solved stream sets for the TBS Link Time Remaining Time scheduler. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$.

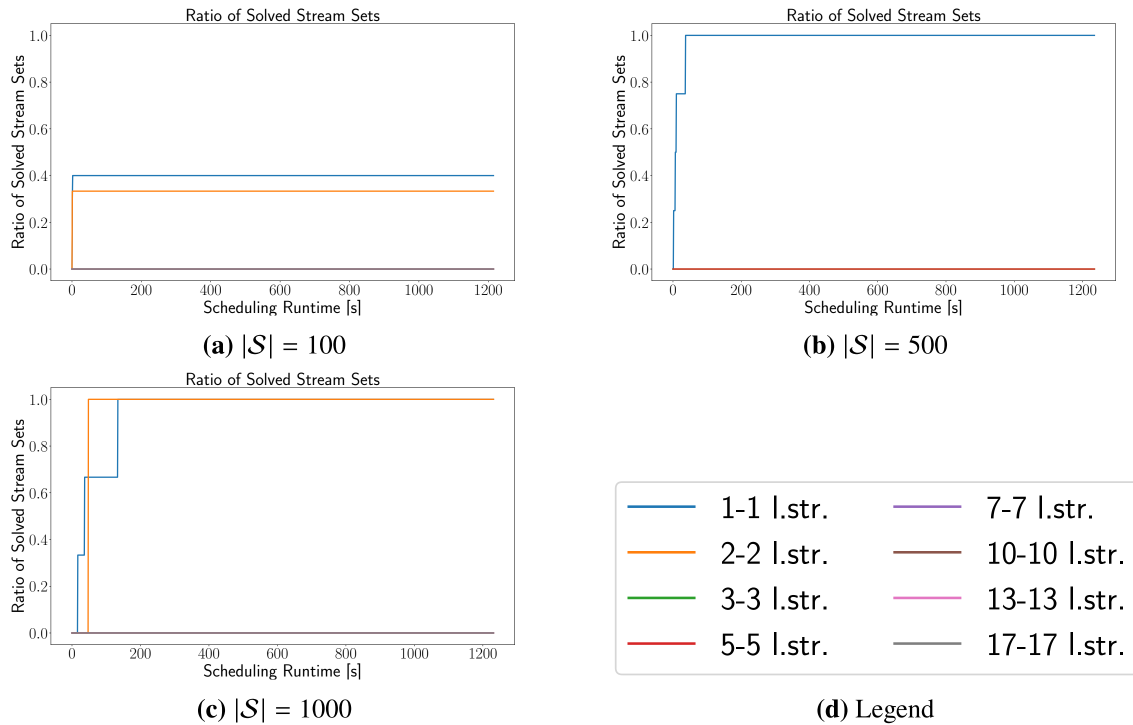


Figure 6.10: Ratio of solved stream sets for the TBS Link Time Late Streams scheduler. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$.

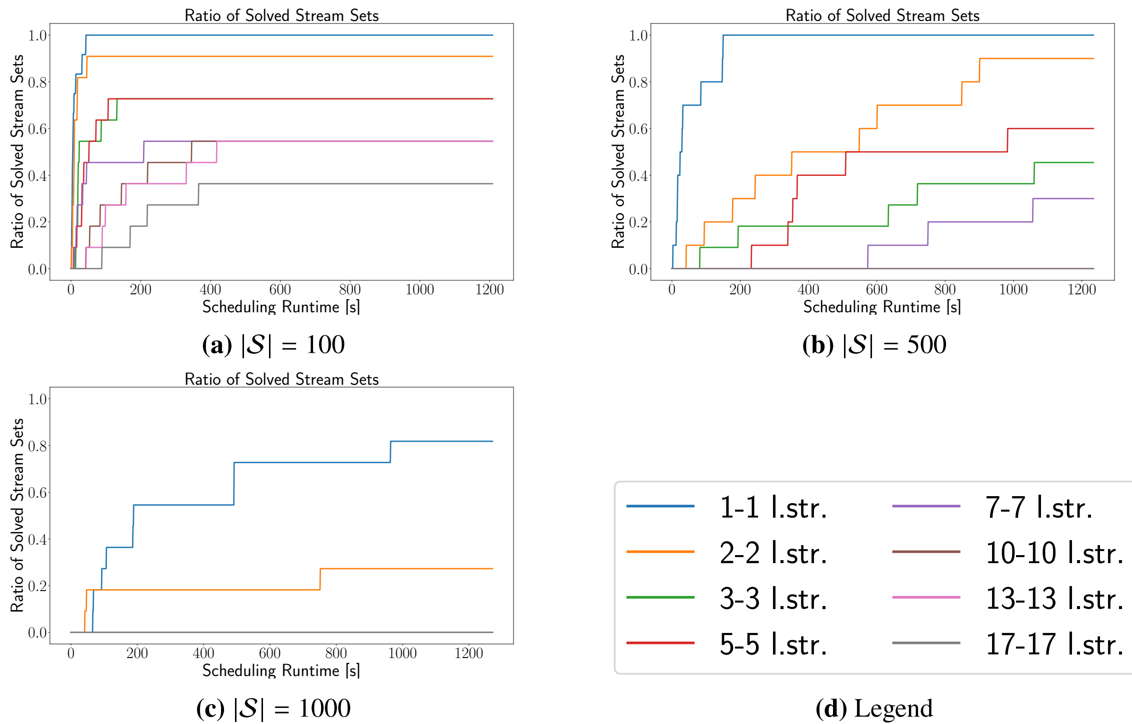


Figure 6.11: Ratio of solved stream sets for the TBS Link Time MCTS scheduler. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$.

longer to simulate, cf. Figure 6.12. The TBS Link Time MCTS approach utilizes the tracing-based simulator in two ways. First, it uses the simulator to determine the Late streams s_{late} in a stream set $\mathcal{S}$. It uses the late streams to establish the conflict streams and stream operations, which it uses to expand child nodes. Second, the TBS Link Time MCTS scheduler uses the tracing-based simulator in the *Default-Policy* of the MCTS to calculate a reward. An increase in simulation time through larger stream sets increases the time it takes the MCTS to perform one iteration of the search. As a result, the TBS Link Time MCTS scheduler performs fewer search iterations in the same amount of runtime - thus, exploring less of the search space. Increasing the runtime of the TBS Link Time MCTS scheduler might remedy this issue and increase the ratio of solved stream sets.

The TSN SMT schedulers can solve small stream sets; however, in some cases, take very long to do so. They do not show a dependence on the number of initial late streams n_{init_late}. The scheduling capabilities of the TBS approaches are influenced by n_{init_late}. The Naive approach was unable to solve a single stream set due to its coarse heuristics and its basic search procedure. The Link Time Remaining Time and Link Time Late Streams approach showed improved behavior. They were able to solve the majority of stream sets with $n_{init_late} = 1$. However, larger values of n_{init_late} lead a substantial decline in solved stream sets. We reasoned that this behavior is caused by how these approaches search the scheduling search tree. They get stuck expanding large numbers of child nodes when deciding if they should backtrack. The MCTS approach showed improved scheduling capabilities for all topology and stream set sizes. In contrast to the previous TBS approaches, it was able to solve stream sets with $n_{init_late} > 2$. However, for large topologies with 1008 switches, it could not solve stream sets with $n_{init_late} > 2$. This is caused by the increase in simulation time when simulating large stream sets with the tracing-based simulator. All in all, the TSN SMT and

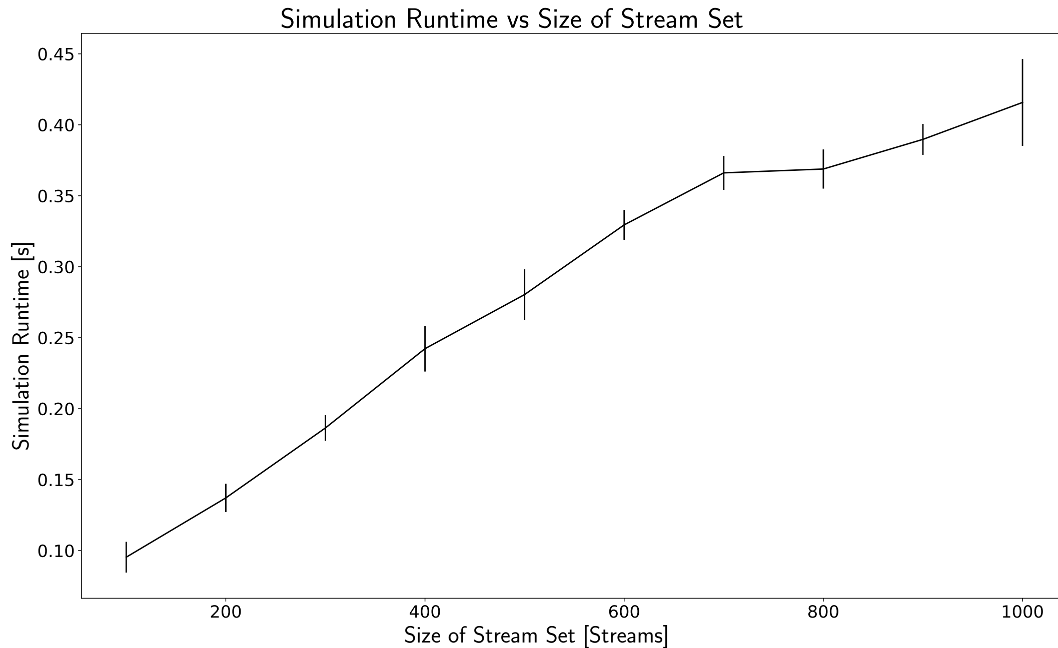


Figure 6.12: Influence of stream set size on simulation runtime for the *tracing-based simulator*. The black error bars indicate the 90% confidence interval of 20 simulation runs for each stream set size.

JSSP schedulers can solve small stream sets. Of the TBS approaches, the MCTS approach showed promising scheduling capabilities. The low time-limit of 1,200 s we set for the TBS approaches reduced the number of solved stream sets, especially for the MCTS approach.

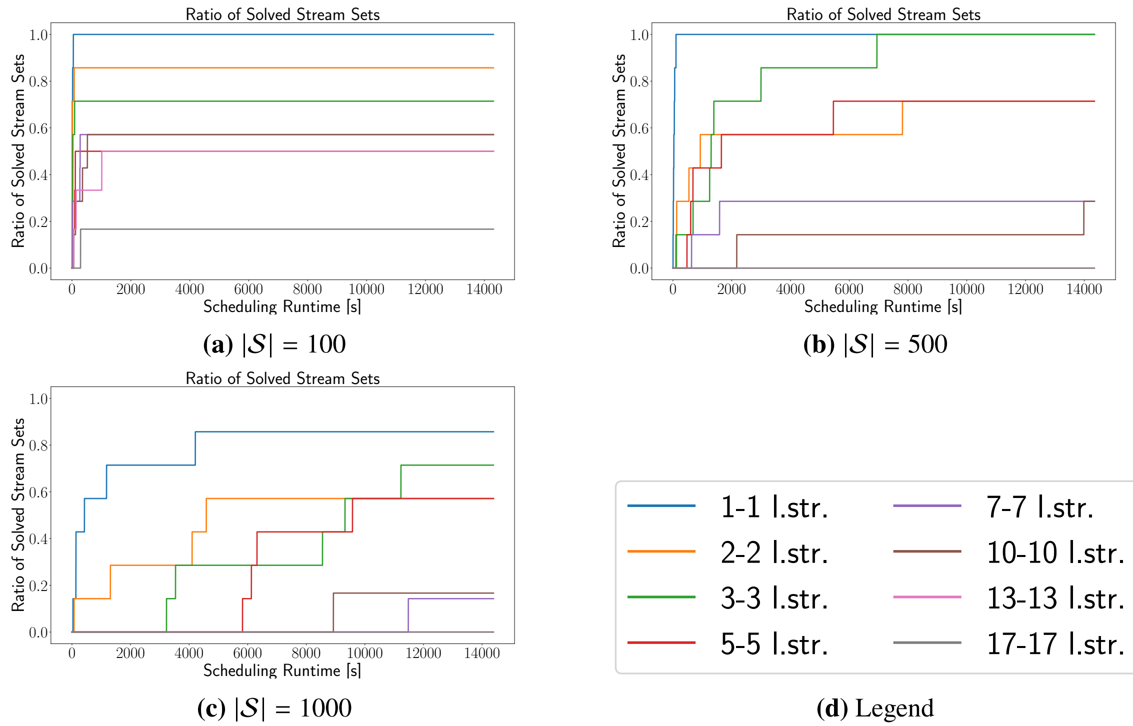
6.2.4 Scheduling Capabilities Link Time Monte Carlo Tree Search

As shown in Figure 6.12, the simulation runtimes of the tracing-based simulator increase with the size of the simulated stream sets, which decreases the amount of stream sets the MCTS approach solves. We argue that increasing the runtime of the MCTS approach might increase the ratio of solved stream sets. In the subsequent section, we perform additional analyses with an increased runtime limit. We use the same input topologies and stream sets we used in the previous analysis. We set the time-limit for these evaluations to 14,300 s (≃4 hours), which is low enough so that a considerable number of scheduling runs could be executed while also being 12 times the previous time-limit of 1,200 s. Due to the increased time-limit, and thus, increased memory consumption, we used an evaluation node with more capable hardware, cf. Table 6.4. Although the node provides 528 GB of RAM, preliminary tests showed that the MCTS approach consumes around 40 GB of memory with the increased time-limit. Note that the CPU of this node has a considerably lower clock rate than the one referenced in Table 6.3, which decreases the performance of single-threaded applications (like the TBS MCTS scheduler).

Figure 6.13 shows the ratio of solved stream sets for the TBS MCTS approach. The ratio of solved stream sets plateaus after 2,000 s for the smallest stream sets of 100 streams, cf. Figure 6.13a. Even for larger n_{init_late}, the TBS MCTS approach is not able to find any more solutions after 2,000 s.

CPU	AMD EPYC 7451 24-Core Processor @ 1198 MHz
Memory	528 GB
OS	Ubuntu 18.04.4 LTS
Kernel	4.15.0-88-generic

Table 6.4: Hardware of the evaluation cluster node.



(a) $|\mathcal{S}| = 100$ **(b)** $|\mathcal{S}| = 500$ **(c)** $|\mathcal{S}| = 1000$ **(d)** Legend

Figure 6.13: Ratio of solved stream sets for the TBS Link Time MCTS scheduler with a 14300 s runtime limit. Each diagram depicts the ratio for different stream set sizes $|\mathcal{S}|$.

Like in the previous analyses with the lower time-limit, the ratio of solved stream sets decreases with larger values for n_{init_late}. For larger stream sets with 500 streams, the increased time-limit does yield more solved stream sets. Stream sets across all n_{init_late} are often solved after 1,200 s. A similar statement can be made for stream sets with 1000 streams. Nearly all stream sets with $n_{init_late} = 1$ are solved, the majority after 1,200 s. The stream sets with $n_{init_late} > 2$ which are solved, are solved after 1,200 s.

Increasing the time-limit did not yield improved results for smaller stream sets. The ratio of solved stream sets plateaus after 2,000 s. This indicates that the MCTS approach finds solutions for some specific stream sets very fast while struggling to find solutions for others. Increasing the time-limit did yield significantly more solved stream sets for larger stream sets. We conclude that increasing the time-limit is advisable for large stream sets.

7 Conclusion and Outlook

In this chapter, we conclude our work and give an outlook on what future work could investigate.

TSN is a set of standards defined by the IEEE TSN Task Group that aims to make Ethernet real-time capable. Ethernet is the de-facto standard for the lower layers of computer networks. TSN employs a TDMA scheme to achieve temporal isolation between data streams. The TDMA scheme divides access to transmission links into slots. Only a single stream has access to the transmission link during a slot. This eliminates interference between streams and the unwanted effects caused by it: unbounded jitter and packet loss. TSN provides deterministic transmission, i.e., guaranteed packet transmission with bounds on latency and jitter. Calculating the time slots of the TDMA scheme is referred to as scheduling. Scheduling is known to be NP-complete [Ull75]; thus, no deterministic polynomial scheduling algorithm exists. However, researchers are continually finding new methods to accelerate the scheduling process. In previous work, [HGF+20], Hellmanns et al. developed a novel scheduling approach to schedule data streams in a TSN network. They use a discrete-event network simulator to simulate these streams and derive a schedule from the simulation results. The authors refer to this approach as *tracing-based scheduling*. However, no procedures in *tracing-based scheduling* exist to handle streams, which violate their deadlines.

The goal of this work was to develop procedures to handle these deadline violations by finding stream configurations that do not violate their requirements. We expanded the tracing-based scheduling approach, proposed by Hellmans et al., to a full-blown scheduling algorithm. The scheduling algorithm finds stream configurations, such that every stream fulfills its latency bounds. Most scheduling approaches express the network model as constraints and try to fulfill these constraints. We propose a different formulation: considering scheduling as a search problem. With this formulation, we can employ search methods to find a correct schedule, i.e., one where all stream requirements are honored. With our formulation, we developed four novel scheduling approaches: Tracing-based Scheduling (TBS) Naive approach, TBS Link Time Remaining Time, TBS Link Time Late Streams, and TBS Link Time Monte Carlo Tree Search (MCTS). Each approach employs a different search routine to find a correct schedule for a stream set.

We evaluated these scheduling approaches with the benchmarking framework [SWHD20]. The benchmarking framework allows comparing different scheduling algorithms. It comes with two state-of-the-art scheduling algorithms pre-integrated: the TSN SMT and JSSP schedulers. Additionally to evaluating our approaches, we compared them to the aforementioned pre-integrated schedulers. We evaluated each scheduler along the following properties: execution rates, scheduling capabilities, and scheduling runtimes. The Naive approach, TBS Link Time Remaining Time and TBS Link Time Late Streams approaches showed low execution rates for large scheduling instances. The generated search tree became too large and required more memory than was available. However, they were able to execute small scheduling instances. Only the TBS MCTS approach was able to execute every problem instance. The Naive approach, TBS Link Time Remaining Time and TBS Link Time Late Streams approaches did schedule the minority of stream sets. The Naive approach was not able

to schedule a single stream set. The Link Time Remaining Time and Late Streams approaches did solve stream sets in certain scheduling instances; however generally, they did not exhibit satisfying scheduling capabilities. The MCTS approach displayed the best scheduling capabilities of the tracing-based approaches. Generally, it was able to schedule the majority of scheduling instances. The ratio of solved stream sets was very high for smaller scheduling instances and decreased for larger ones. We suspected that the low time limit of 1,200 s we set for its scheduling runtime causes this behavior. To further analyze its solving capabilities, we performed additional evaluations on better performing hardware and with longer scheduling runtimes.

We developed a procedure, the TBS MCTS approach, which can handle streams that violate their latency bounds. It finds stream set configurations, such that every stream heeds its latency bounds. Thus, we achieved the goal outlined in this work.

Outlook

In this section, we present possible topics for future work. In Section 6.2.4, we showed that increasing the runtime does yield more solved stream sets for large stream sets. However, for small stream sets, the ratio of solved stream sets plateaued. We suspect that the MCTS approach can find solutions for some stream sets quickly while it struggles to find solutions for others. Future work could look into which stream sets were solved quickly and compare them to stream sets, which were not solved. For all evaluations in this work, the switches operated in *store and forward* mode. However, the tracing-based simulator and our scheduling approaches can calculate schedules for switches operating in *cut-through* mode. Traffic forwarded by switches in *cut-through* mode experiences less delay than in switches in *store & forward*. Using switches in *cut-through* mode might improve the schedulability of stream sets and is interesting to examine in future research. The most promising scheduling approach we developed is the TBS MCTS approach. It utilizes the Monte Carlo Tree Search to find solutions. Future work can focus on optimizing the MCTS. The utility function selecting the node for expansion, and the reward function determine the performance of the MCTS. It might be interesting to implement other utility functions to alter how the MCTS expands the search tree. Additionally, exploring other reward calculation procedures might improve the performance further. The approaches developed in this work are all single-threaded. Parallelizing the tree search might yield further performance improvements. Parallelizing could be done by splitting the search tree into sub-trees and applying a tree search algorithm to every sub-tree in-parallel. Some schedulers can take solving hints to accelerate schedule synthesis. If a TBS algorithm cannot find a solution, it might be worthwhile to use intermediate search results, found by the TBS algorithm, as solving hints for a subsequent scheduler.

Bibliography

[ACF02] P. Auer, N. Cesa-Bianchi, P. Fischer. “Finite-time Analysis of the Multiarmed Bandit Problem”. In: *Machine Learning* 47 (May 2002), pp. 235–256. DOI: 10.1023/A:1013689704352 (cit. on p. 51).

[APB+18] A. Ademaj, D. Puffer, D. Bruckner, G. Ditzel, L. Leurs, M. Stanica, P. Didier, R. Hummen, R. Blair, T. Enzinger. *Iic results white paper: Time sensitive networks for flexible manufacturing testbed-description of converged traffic types*. 2018 (cit. on pp. 40, 58).

[BC86] J. Banks, J. S. Carson. “Introduction to Discrete-Event Simulation”. In: *Proceedings of the 18th Conference on Winter Simulation*. WSC ’86. Washington, D.C., USA: Association for Computing Machinery, 1986, pp. 17–23. ISBN: 0911801111. DOI: 10.1145/318242.318253. URL: https://doi.org/10.1145/318242.318253 (cit. on p. 25).

[BCNN10] J. Banks, J. Carson, B. Nelson, D. Nicol. *Discrete-Event System Simulation*. English. 5th ed. Prentice Hall, 2010. ISBN: 0136062121 (cit. on p. 25).

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43 (cit. on pp. 28, 29, 51).

[BT18] C. Barrett, C. Tinelli. “Satisfiability modulo theories”. In: *Handbook of Model Checking*. Springer, 2018, pp. 305–343 (cit. on pp. 31, 32, 57).

[CO14] S. S. Craciunas, R. S. Oliver. “SMT-Based Task- and Network-Level Static Schedule Generation for Time-Triggered Networked Systems”. In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. RTNS ’14. Versaille, France: Association for Computing Machinery, 2014, pp. 45–54. ISBN: 9781450327275. DOI: 10.1145/2659787.2659812. URL: https://doi.org/10.1145/2659787.2659812 (cit. on pp. 32, 33, 60).

[COCS16] S. S. Craciunas, R. S. Oliver, M. Chmelík, W. Steiner. “Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS ’16. Brest, France: Association for Computing Machinery, 2016, pp. 183–192. ISBN: 9781450347877. DOI: 10.1145/2997465.2997470. URL: https://doi.org/10.1145/2997465.2997470 (cit. on pp. 33, 43).

[DK+14] F. Dürr, T. Kohler, et al. “Comparing the forwarding latency of OpenFlow hardware and software switches”. In: *Fakultät Informatik, Elektrotechnik Informationstechnik, Univ. Stuttgart, Stuttgart, Germany, Tech. Rep. TR* 4 (2014), p. 2014 (cit. on p. 40).

[DN16] F. Dürr, N. G. Nayak. "No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*. Ed. by A. Plantec, F. Singhoff, S. Faucou, L. M. Pinho. ACM, 2016, pp. 203–212. DOI: 10.1145/2997465.2997494. URL: https://doi.org/10.1145/2997465.2997494 (cit. on pp. 32, 34, 36, 56).

[DZ83] J. D. Day, H. Zimmermann. "The OSI reference model". In: *Proceedings of the IEEE* 71.12 (1983), pp. 1334–1340 (cit. on p. 17).

[FBG18] J. Farkas, L. L. Bello, C. Gunther. "Time-Sensitive Networking Standards". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 20–21 (cit. on p. 20).

[FHC+19] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, K. Rothermel. *NeSTiNg: Simulating IEEE Time-sensitive Networking (TSN) in OMNeT++*. Mar. 2019. DOI: 10.1109/NetSys.2019.8854500 (cit. on pp. 26, 37, 38, 54).

[Fin18] N. Finn. "Introduction to Time-Sensitive Networking". In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 22–28 (cit. on pp. 20, 21).

[GP18] V. Gavriluţ, P. Pop. "Scheduling in time sensitive networks (TSN) for mixed-criticality industrial applications". In: *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*. June 2018, pp. 1–4. DOI: 10.1109/WFCS.2018.8402374 (cit. on p. 34).

[Gra66] R. L. Graham. "Bounds for certain multiprocessing anomalies". In: *The Bell System Technical Journal* 45.9 (1966), pp. 1563–1581 (cit. on p. 31).

[HBŠ10] Z. Hanzálek, P. Burget, P. Šůcha. "Profinet IO IRT Message Scheduling With Temporal Constraints". In: *IEEE Transactions on Industrial Informatics* 6.3 (Aug. 2010), pp. 369–380. ISSN: 1941-0050. DOI: 10.1109/TII.2010.2052819 (cit. on pp. 32, 34).

[HFG+20] D. Hellmanns, J. Falk, A. Glavackij, R. Hummen, S. Kehrer, F. Dürr. "On the Performance of Stream-based, Class-based Time-aware Shaping and Frame Preemption in TSN". In: Feb. 2020. DOI: 10.1109/ICIT45562.2020.9067122 (cit. on p. 35).

[HGF+20] D. Hellmanns, A. Glavackij, J. Falk, R. Hummen, S. Kehrer, F. Dürr. "Scaling TSN Scheduling for Factory Automation Networks". In: Mar. 2020. DOI: 10.1109/WFCS47810.2020.9114415 (cit. on pp. 3, 16, 35–38, 73).

[IEE08] IEEE. "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (2008), pp. 1–300 (cit. on p. 21).

[IEE11a] IEEE. "IEEE Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks". In: *IEEE Std 802.1AS-2011* (2011), pp. 1–292 (cit. on p. 21).

[IEE11b] IEEE. "IEEE Standard for Local and metropolitan area networks–Audio Video Bridging (AVB) Systems". In: *IEEE Std 802.1BA-2011* (2011), pp. 1–45 (cit. on p. 22).

[IEE16a] IEEE. "IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery". In: *IEEE Std 802.1AB-2016 (Revision of IEEE Std 802.1AB-2009)* (2016), pp. 1–146 (cit. on p. 21).

[IEE16b] IEEE. “IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic”. In: *IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015)* (2016), pp. 1–57 (cit. on p. 21).

[IEE17] IEEE. “IEEE Standard for Local and metropolitan area networks–Frame Replication and Elimination for Reliability”. In: *IEEE Std 802.1CB-2017* (2017), pp. 1–102 (cit. on p. 22).

[IEE18a] IEEE. “IEEE Standard for Ethernet”. In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600 (cit. on pp. 17, 18).

[IEE18b] IEEE. “IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks”. In: *IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014)* (2018), pp. 1–1993 (cit. on pp. 18, 20, 23, 24).

[IEE18c] IEEE. “IEEE Standard for Local and metropolitan area networks – Time-Sensitive Networking for Fronthaul”. In: *IEEE Std 802.1CM-2018* (2018), pp. 1–62 (cit. on p. 22).

[IEE20a] IEEE. “IEEE Standard for Local and Metropolitan Area Networks–Link Aggregation”. In: *IEEE Std 802.1AX-2020 (Revision of IEEE Std 802.1AX-2014)* (2020), pp. 1–333 (cit. on p. 21).

[IEE20b] IEEE. *Time-Sensitive Networking (TSN) Task Group*. 2020. URL: https://1.ieee802.org/tsn/ (visited on 07/22/2020) (cit. on p. 20).

[INE19] INET. *INET Framework*. 2019. URL: https://inet.omnetpp.org/ (visited on 11/13/2019) (cit. on p. 25).

[KS06] L. Kocsis, C. Szepesvári. “Bandit Based Monte-Carlo Planning”. In: vol. 2006. Sept. 2006, pp. 282–293. DOI: 10.1007/11871842_29 (cit. on p. 51).

[MB08] L. de Moura, N. Bjørner. “Z3: an efficient SMT solver”. In: vol. 4963. Apr. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 57).

[OMN19] OMNeT++. *Simulation Manual - OMNeT++ version 5.5*. 2019. URL: https://doc.omnetpp.org/omnetpp/manual/ (visited on 11/13/2019) (cit. on p. 25).

[PRH19] F. Pozo, G. Rodriguez-Navas, H. Hansson. “Methods for Large-Scale Time-Triggered Network Scheduling”. In: *Electronics* 8.7 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8070738. URL: https://www.mdpi.com/2079-9292/8/7/738 (cit. on p. 35).

[PRHS15] F. Pozo, G. Rodriguez-Navas, H. Hansson, W. Steiner. “SMT-based synthesis of TTEthernet schedules: A performance study”. In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. June 2015, pp. 1–4. DOI: 10.1109/SIES.2015.7185055 (cit. on p. 34).

[RN09] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597 (cit. on pp. 26–28).

[Sch04] E. Schemm. “SERCOS to link with Ethernet for its third generation”. English. In: *Computing and Control Engineering* 15 (2 Apr. 2004), 30–33(3). ISSN: 0956-3385. URL: https://digital-library.theiet.org/content/journals/10.1049/cce_20040205 (cit. on p. 15).

[SDT+17] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, G. Mühl. "ILP-Based Joint Routing and Scheduling for Time-Triggered Networks". In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. RTNS '17. Grenoble, France: Association for Computing Machinery, 2017, pp. 8–17. ISBN: 9781450352864. DOI: 10.1145/3139258.3139289. URL: https://doi.org/10.1145/3139258.3139289 (cit. on p. 32).

[Spe20] J. Specht. *On Standardization of Cut-Through Forwarding (CTF)*. Jan. 24, 2020. URL: http://www.ieee802.org/1/files/public/docs2020/new-specht-cut-through-tech-0120-v01.pdf (visited on 07/08/2020) (cit. on p. 19).

[Ste10] W. Steiner. "An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks". In: Nov. 2010, pp. 375–384. DOI: 10.1109/RTSS.2010.25 (cit. on pp. 32–34, 56, 57, 60).

[Ste11] W. Steiner. "Synthesis of Static Communication Schedules for Mixed-Criticality Systems". In: *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. Mar. 2011, pp. 11–18. DOI: 10.1109/ISORCW.2011.12 (cit. on pp. 34, 36).

[SWHD20] P. Schneefuss, M. Weitbrecht, D. Hellmanns, F. Dürr. "Benchmarking TSN Schedulers". In: (2020) (cit. on pp. 53, 54, 57, 73).

[Tan81] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1981. ISBN: 0-13-165183-8 (cit. on pp. 17, 18).

[TPS12] D. Tamas-Selicean, P. Pop, W. Steiner. "Synthesis of Communication Schedules for TTEthernet-Based Mixed-Criticality Systems". In: *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '12. Tampere, Finland: Association for Computing Machinery, 2012, pp. 473–482. ISBN: 9781450314268. DOI: 10.1145/2380445.2380518. URL: https://doi.org/10.1145/2380445.2380518 (cit. on p. 34).

[TTE] TTEthernet. *Time-Triggered Ethernet*. URL: https://www.sae.org/standards/content/as6802/ (visited on 07/12/2020) (cit. on pp. 15, 31).

[TV99] E. Tovar, F. Vasques. "Real-time fieldbus communications using Profibus networks". In: *IEEE Transactions on Industrial Electronics* 46.6 (1999), pp. 1241–1251 (cit. on p. 15).

[Ull75] J. D. Ullman. "NP-Complete Scheduling Problems". In: *J. Comput. Syst. Sci.* 10.3 (June 1975), pp. 384–393. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80008-0. URL: https://doi.org/10.1016/S0022-0000(75)80008-0 (cit. on pp. 16, 23, 73).

[Was97] R. L. Wasserstein. "Monte Carlo: Concepts, Algorithms, and Applications". In: *Technometrics* 39.3 (1997), pp. 338–338. DOI: 10.1080/00401706.1997.10485133 (cit. on p. 28).

[Wil09] H. P. Williams. *Logic and Integer Programming*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 0387922792 (cit. on pp. 31, 32).

[YN92] T. Yamada, R. Nakano. "A Genetic Algorithm Applicable to Large-Scale Job-Shop Problems." In: vol. 2. Jan. 1992, pp. 283–292 (cit. on p. 33).

All links were last followed on August 10, 2020.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature