Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Optimizing ILP-based Joint Scheduling and Routing for Time-Aware Shaping in Factory Automation Networks

Lucas Haug

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Kurt Rothermel
Supervisor:	M.Sc. David Hellmanns
Commenced:	April 28, 2020
Completed:	October 28, 2020

Abstract

With the rise of Industry 4.0 and Internet of Things (IoT), the need for deterministic real-time communication is higher than ever before. In the past, fieldbus systems have dominated the field of time-critical applications. However, they are incompatible among each other and are not able to transmit time-sensitive and non-time-sensitive traffic over the same medium. The widespread usage of IEEE Ethernet Networks and the growing need for real-time communication lead to the IEEE *Time-sensitive Networking (TSN)* Standards. These TSN Standards extend IEEE Ethernet networks with real-time capabilities. They provide multiple priority levels and a *Time Division Multiple Access (TDMA)*-based gating mechanism for each switch. However, they do not define how to calculate the TDMA schedules. There are already different approaches, which solve the Scheduling- or the *Joint Routing and Scheduling (JRaS)* problem for TSN. These approaches are either complete and suffer from a high runtime or they are heuristic and do not guarantee to find a feasible solution. In this work, we improve upon an already existing *Integer Linear Programming (ILP)*-based JRaS approach. For this, we develop different optimizations of different categories, which reduce the complexity of the problem or make use of an ILP-solver's specific capabilities. We evaluate our different optimizations individually and in combination in order to find the best combination for two different switching types. These switching types are known as Store-and-Forward switching, which is the default switching type provided in the IEEE Ethernet standards and Cut-Through switching, which is an optimization commonly used in industrial networks. Additionally, we benchmark our optimized ILP-based approach against other schedulers. With our best optimization combination, we are able to reduce the runtime by about 80 % compared to the base ILP-model.

Kurzfassung

Mit dem Aufschwung von Industrie 4.0 und Internet of Things (IoT) ist der Bedarf an deterministischer Echtzeitkommunikation höher als je zuvor. In der Vergangenheit haben Feldbussysteme den Bereich der zeitkritischen Anwendungen weitgehend geprägt. Sie sind jedoch untereinander inkompatibel und nicht in der Lage, zeitkritischen und nicht zeitkritischen Verkehr über dasselbe Medium zu übertragen. Die weit verbreitete Verwendung von IEEE-Ethernet-Netzwerken und der wachsende Bedarf an Echtzeit-Kommunikation führten zu den IEEE *Time-sensitive Networking (TSN)* Standards. Diese TSN Standards erweitern IEEE-Ethernet-Netzwerke um Echtzeitfähigkeiten. Sie bieten mehrere Prioritätsstufen und einen auf Zeitmultiplexing (*engl. Time Division Multiple Access, TDMA*) basierenden Gating-Mechanismus für jeden Switch. Sie definieren jedoch nicht, wie die TDMA-Schedules zu berechnen sind. Es gibt bereits verschiedene Ansätze, die das Scheduling- oder das *Joint Routing and Scheduling (JRaS)* Problem für TSN lösen. Diese Ansätze sind entweder vollständig und leiden unter einer hohen Laufzeit oder sie sind heuristisch und garantieren nicht, eine gültige Lösung zu finden. In dieser Arbeit verbessern wir einen bereits bestehenden *Integer Linear Programming (ILP)*-basierten JRaS-Ansatz. Dazu entwickeln wir verschiedene Arten von Optimierungen, die die Komplexität des Problems reduzieren oder die spezifischen Fähigkeiten eines ILP-Lösers nutzen. Wir evaluieren unsere verschiedenen Optimierungen einzeln und in Kombination, um die beste Kombination für zwei verschiedene Switching-Typen zu finden. Diese Switching-Typen sind bekannt als Store-and-Forward, der Standard-Switching-Typ, der in den IEEE-Ethernet-Standards vorgesehen ist, und als Cut-Through-Switching, eine Optimierung, die

häufig in industriellen Netzwerken verwendet wird. Zusätzlich vergleichen wir unseren optimierten ILP-basierten Ansatz mit anderen Schedulern. Mit unserer besten Optimierungskombination sind wir in der Lage, die Laufzeit um etwa 80 % im Vergleich zum ILP-Basismodell zu reduzieren.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

CBS Credit-Based Shaper. 23

CRC Cyclic redundancy check. 18

CSP Constraint Satisfaction Problem. 29

DEI Drop Eligible Indicator. 19

DFS Depth-First search. 26

ETS Enhanced Transmission Selection. 23

FBS Filtered Beam Search. 32

FCS Frame Check Sequence. 18

GCL Gate Control List. 7, 23, 24, 41

ILP Integer Linear Programming. 3, 15, 17, 24

IoT Internet of Things. 3, 15

IRS Iterative Resource Scheduling. 32

JRaS Joint Routing and Scheduling. 3, 15, 29, 37, 41, 77

JSSP Job Shop Scheduling Problem. 8, 29

LCS Longest Common Subsequence. 53

lhs left-hand side. 25

MCTS Monte Carlo Tree Search. 33

PCP Priority Code Point. 18

PTP Precision Time Protocol. 22

PTT Parameter Tuning Tool. 7, 59

rhs right-hand side. 25

SMT Satisfiability Modulo Theory. 30

TAS Time-Aware Shaping. 22, 24

TCI Tag Control Information. 18

TDMA Time Division Multiple Access. 3, 7, 24

TG Task Group. 15, 22

TPID Tag Protocol Identifier. 18

TS Transmission Selection. 7, 20, 22, 23, 24

TSA Transmission Selection Algorithm. 7, 22, 23

TSN Time-sensitive Networking. 3, 15, 17, 22, 77

VID VLAN Identifier. 19

VLAN Virtual Local Area Network. 18

1 Introduction

Deterministic real-time communication is a key-requirement for safety in industrial networks. The need for such real-time networks is growing rapidly with the Internet of Things (IoT) and Industry 4.0. Many real-time networks process sensor data to trigger actions with effects to the physical world, thus they require small latencies and a guarantee that specific packets are always fully transmitted within a specified time window. This includes applications such as industrial automation networks, where malfunction due to late messages could lead to a high economical damage and expose human beings to dangerous situations. Another example is in-car communication, where real-time data transfer is crucial for the safety of humans.

Many systems have evolved in the past to satisfy the needs of time-critical applications. Different fieldbuses, such as CANbus, have been used for a long time now and are still the most commonly used systems in the automotive field. With the growth of Ethernet networks, multiple Ethernet-based solutions, such as PROFINET and TTEthernet [SAE16] were used. One advantage of those Ethernet-based solutions is the possibility to transmit time-sensitive and non-time-sensitive traffic over the same medium. Despite the use of the same transmission medium, different Ethernet-based solutions are incompatible among each other, but intercompatibility is mandatory for many applications such as IoT networks.

The growing need of real-time networks and widespread usage of existing IEEE Ethernet networks lead to the foundation of the IEEE Time-sensitive Networking (TSN) Task Group (TG), to extend IEEE Ethernet with real-time capabilities. The TSN standards define different priority levels for Time Division Multiple Access (TDMA), which is implemented as a gating mechanism in each TSN-capable device in the network. To use TSN's real-time capability, routes and schedules, which are used to set up the gates of switches, need to be calculated for each frame. Calculating the routes and schedules for a specific network topology and a set of streams means, that each frame's route and the exact time at each switch needs to be precalculated to ensure, they meet their respective real-time requirements. Solving this scheduling problem is NP-hard [Ste10] and therefore requires an efficient and optimized approach to provide a solution within a reasonable amount of time.

There is already work on solving the *Joint Routing and Scheduling (JRaS)* problem for TSNs. However, most of these approaches do not reduce the problem's complexity beforehand. We use an already existing approach based on *Integer Linear Programming (ILP)* as the basis of our work. The goal of our work is to develop and explore different optimization approaches to reduce the runtime of ILP-based schedulers. In this work, we present approaches of three different categories. The first category reduces the ILP by reducing the underlying network topology model. Approaches of the second category provide changes to the ILP generation procedure. These approaches either remove unnecessary variables or restrict the values of variables to a specific interval. The last category makes use of additional functionalities of the Gurobi ILP-solver.

In the following Chapter 2, we first give a brief overview of the technical background and technologies used in this work. Following that, we discuss approaches of already existing work with their respective advantages and disadvantages in Chapter 3. Our System model in Chapter 4 provides the formal representation for our network and streams based on graphs. In our System Model, we also introduce assumptions on the configuration of our network and provide a description of the base ILP-model by Hellmanns et al. [HDHK18]. Chapter 5 contains our problem statement, which formalizes the problem we aim to solve with our optimized ILP-based approach. In the main part in Chapter 6, we then present our approaches and methods with which we aim to achieve our goal of reducing the runtime of ILP-based schedulers for Time-sensitive Networks. We then analyze the capabilities of these optimization approaches and compare them to other schedulers in Chapter 7. Finally, we conclude our work in Chapter 8.

2 Background

In this chapter, we give an overview over the underlying technologies used in this work. This chapter consists of four technical sections. The first section deals with general Ethernet technologies, with the main focus on the forwarding process. In the second section, we extend the first block with Time-sensitive Networking (TSN) specific technology. Thereafter, we describe the concept and parts of Integer Linear Programming (ILP). Finally, in the last block, we define our graph definition and discuss two different graph algorithms.

2.1 IEEE Ethernet

In this section, we cover IEEE Ethernet related background, as it contains basic knowledge about networks and its devices. This knowledge is required in order to understand TSN-specific topics. The main focus of this section is the forwarding process of an Ethernet switch. One prerequisite to understand the forwarding process is to know the different parts of an Ethernet frame, hence we start this chapter with the explanation of an Ethernet frame's structure. Following that, we introduce the forwarding steps in an Ethernet switch. We then explain two different switching types, which define, when a switch starts to forward a frame. Finally, based on the above steps, we define the different delay types of Ethernet switching.

2.1.1 Ethernet Frames

In this section, we discuss the parts of an Ethernet frame based on Figure 2.1, with a main focus on the IEEE 802.1Q header extension. The following description of the Ethernet frame is based on the IEEE 802.3 Standard [IEE18a]:

Source/Destionation MAC The header starts with a *Destination MAC* field containing the MAC address of the receiver. This destination MAC address can either be the MAC address of a single receiver or a multi-/broadcast MAC address. Similarly, the *Source MAC* field holds the MAC address of a frame's sender.

EtherType Following the source address is an *EtherType* field, which normally specifies the type of protocol inside the *Payload* field. This field can also be used to announce the length of the following payload.

Payload The payload contains the main message by the sender. It has a maximum size of 1,500 B

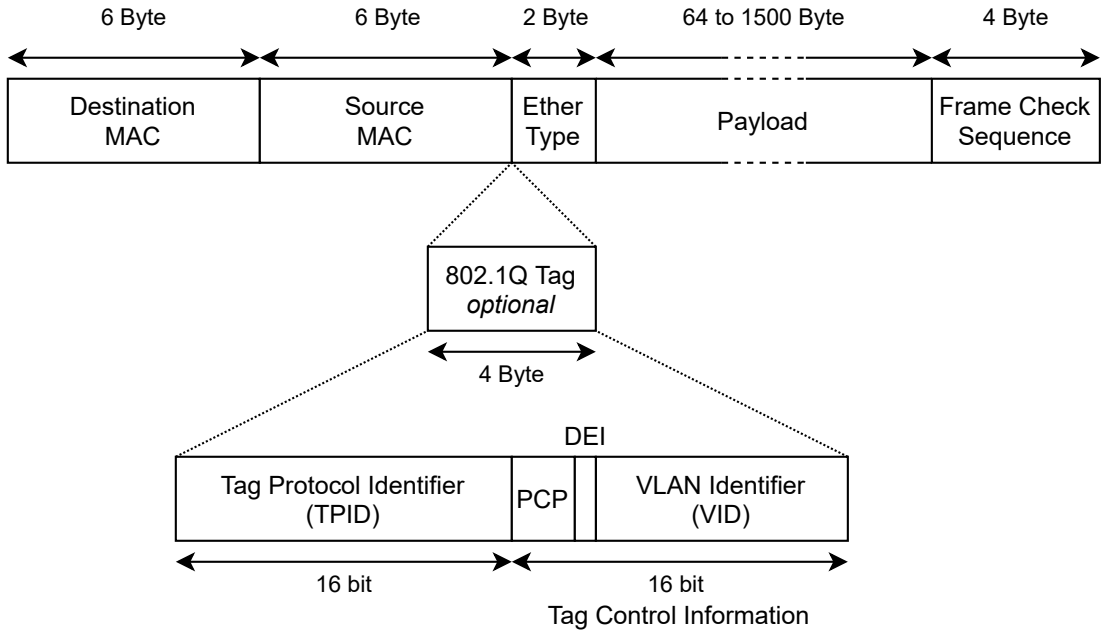


Figure 2.1: Ethernet Frame and VLAN tag [IEE18a; IEE18b].

Frame Check Sequence (FCS) The last value of an Ethernet frame is a 4 B *Frame Check Sequence (FCS)*, which contains a checksum to identify corrupt frames. The FCS is located at the end of a frame, because it uses Cyclic redundancy check (CRC), which enables the switch to calculate the checksum while receiving the frame and compare it to the received checksum in the end [Wil93].

VLAN tag

The IEEE 802.1Q tag, also known the *Virtual Local Area Network (VLAN)* tag is an extension to the base IEEE 802.3 Ethernet frame and has a size of 4 B. It is defined in the IEEE 802.1Q Standard [IEE18b]. The VLAN tag consists of two main parts with a size of 16 bit each. The first part is the *Tag Protocol Identifier (TPID)* and the second part is the *Tag Control Information (TCI)*, which contains three different control values. We discuss these parts in the following. We call frames containing a VLAN tag *tagged* frames, whereas we refer to frames without the VLAN tag as *untagged*.

Tag Protocol Identifier (TPID) The first 16 bit of the VLAN tag are reserved for the TPID, which is a fixed HEX value of 0x8100. In a tagged frame the TPID takes the position of the EtherType field of an untagged frame. This is why the fixed HEX value 0x8100 is a reserved EtherType value signaling an inserted VLAN tag.

Priority Code Point (PCP) The TCI starts with a 3 bit *Priority Code Point (PCP)* field, which allows the sender to specify the priority of the frame based on eight different priority classes. A higher PCP value normally means, the frame has a higher priority.

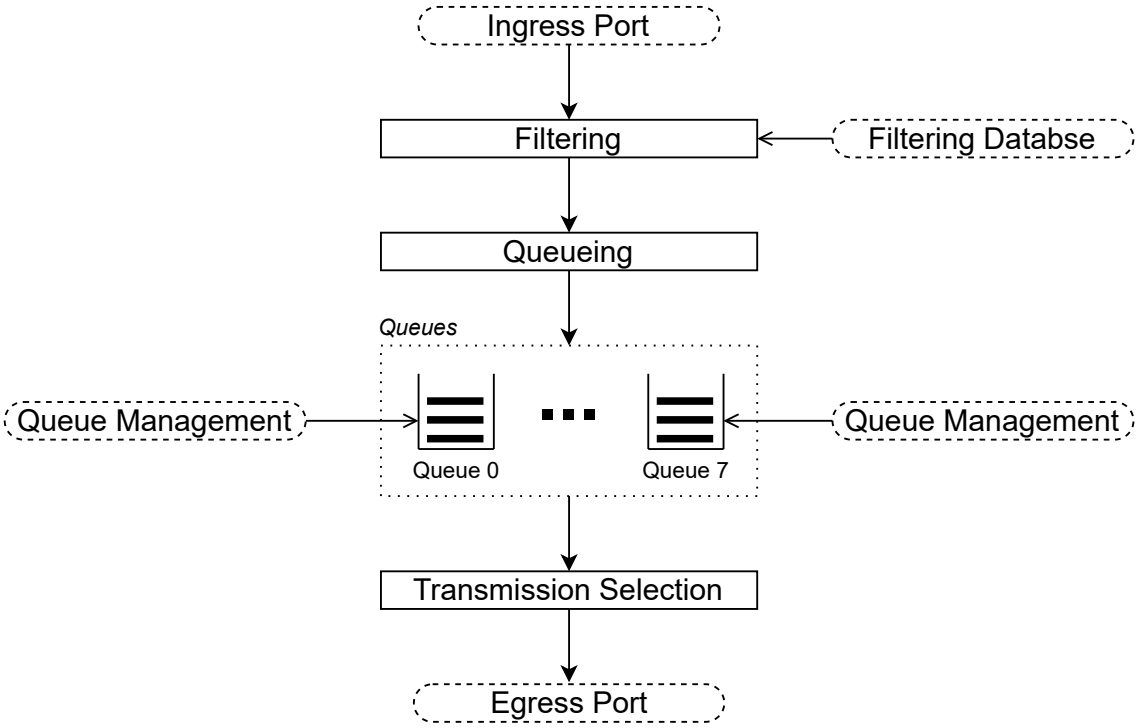


Figure 2.2: Simplified model of the forwarding process in an IEEE Ethernet compliant switch [IEE18b].

Drop Eligible Indicator (DEI) The next TCI bit is the *Drop Eligible Indicator (DEI)*. Setting this value to 1 allows a switch to drop the frame in case of congestion.

VLAN Identifier (VID) The last 12 bit field is the *VLAN Identifier (VID)*. This field defines to what VLAN the frame belongs to.

2.1.2 Forwarding Process

Based on the frame header, we can now discuss the forwarding process of IEEE compliant switches. The forwarding process (Figure 2.2) specifies the steps executed by a switch for each frame from its arrival at the ingress port until the outgoing transmission at the egress port. These steps are a simplified model of the forwarding process model specified in the IEEE 802.1Q standard [IEE18b].

Filtering

Each frame received at the ingress port of the switch first passes through the filtering step. In this filtering step, the switch needs to make a forwarding decision for each incoming frame. This decision is based on different parameters such as the frame's VLAN Identifier (VID) and destination MAC address. The filtering database contains entries mapping these parameters to a set of egress ports [IEE18b].

Some switches may support *Flow Filtering* which allows streams to be distinguished by a flow hash value in the *Flow Filtering Tag* (not further discussed in this work). Another possible option is to assign a multicast MAC address to each stream, with only the receiver(s) of this stream in the corresponding multicast group.

Queueing

Each switch may have up to eight different queues. If the switch has exactly eight queues, there is a *one-to-one* mapping with one queue for each priority specified in the previously mentioned PCP field. Otherwise, there is a *many-to-one* mapping which maps multiple PCP values to a queue [IEE18b]. After filtering, the switch queues the forwarded frame in exactly one queue for transmission at the egress port.

Each queue has its own *Queue Management*. It is mainly responsible for removing frames after a transmission requests [IEE18b]. Finally, the *Transmission Selection (TS)* selects the currently transmitting queue(s) based on a given metric. We discuss the Transmission Selection in Section 2.2.

2.1.3 Switching Types

There are two different switching types, which define, when a switch starts to forward a frame. We discuss these two methods, namely *Store-and-Forward* and *Cut-Through* Switching [Cis08], in this section.

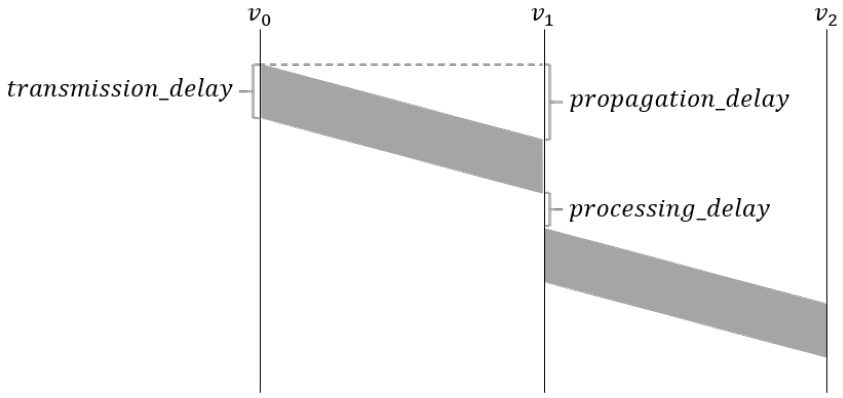
A *Store-And-Forward* switch first fully receives a frame at the ingress port. It then processes and forwards this package to the correct egress port step by step based on the pipeline described in Section 2.1.2.

In contrast, *Cut-Through* switches are able to forward an Ethernet frame before fully receiving it. They are able to start the filtering step, and thus the forwarding pipeline, immediately after receiving all relevant data for this step (e.g. the destination MAC address). When using Cut-Through switching, intermediate switches are not able to determine and discard corrupted packages before forwarding them, as the checksum is located at the end of an Ethernet frame, and thus the switch already started the internal forwarding proceess before receiving the checksum.

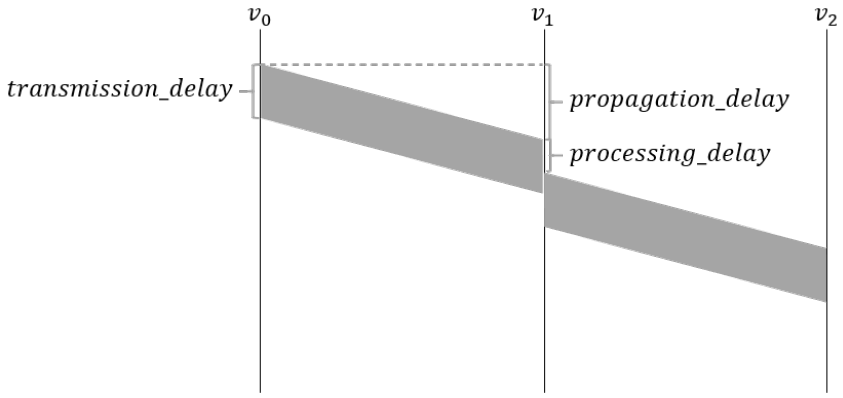
2.1.4 Delays

To calculate correct schedules for the streams in our network, we need to take different delay types into account. We define these delay types in this chapter using Figure 2.3, similar to the definition provided in the *IEEE 802.1Q* standard [IEE18b].

Propagation Delay is the time that passes until a signal change initiated by the sender reaches the receiver. This delay is independent of the frame's size and only depends on the transmission medium and the distance between the sender and receiver.



(a) Delay types for Store-and-Forward switching.



(b) Delay types for Cut-Through switching.

Figure 2.3: Visualization of different delay types.

Transmission Delay is defined as the timespan between the sender starting and finishing the transmission of a frame. This delay only depends on the frame's size s (in bit) and the speed of the connecting link b (in $\frac{\text{bit}}{\text{s}}$). Based on these two values, the transmission delay d_{trans} can be calculated using the formula $d_{\text{trans}} = \frac{s}{b}$.

Processing Delay The definition of the *Processing Delay* depends on the switching mode as described in Section 2.1.3.

For Store-And-Forward switching, our definition is similar to the *Store-and-forward delay* in the IEEE Std. 802.1Q [IEE18b]. It holds the amount of time a switch needs to process a frame through all forwarding steps as described in Section 2.1.2 under the assumption of empty queues. Figure 2.3a shows this behavior.

For Cut-Through switching, we have a similar definition. We define the processing delay as the time from receiving the first bit at the ingress port until sending the first bit at the outgress port (Figure 2.3b). Again, under die assumption of empty queues.

The processing delay highly depends on the implementation of the switch, and we assume it to be independent of the size of a frame.

Queuing Delay We define the *Queuing Delay* as the time that passes between queuing a frame and selecting this frame for transmission. Our queuing delay definition is similar to definition of the *interference delay* in the *IEEE 802.1Q* standard. This delay depends on the network's utilization and can only be calculated if the Transmission Selection (TS) as well as all interfering frames are known beforehand.

2.2 Time-sensitive Networking (TSN)

This section is about TSN specific technology and extends the Ethernet description from Section 2.1. We start with an introduction to the different TSN standards relevant for this work. Following that, we discuss the basic concepts of these standards. Finally, we explain the concept of schedules and the Time-Aware Shaping (TAS).

2.2.1 TSN Standards

The TSN standards are a set of standards developed by the *Time-sensitive Networking (TSN) Task Group (TG)* [IEEb], which is part of the *IEEE 802.1* working group [IEEa]. The purpose of the TSN standards is to extend standard Ethernet networks with time-sensitive functionality such as bounded latencies and guaranteed packet delivery. Most TSN standards were amendments to the *IEEE 802.1Q* standard [IEE18b] as discussed in Section 2.1, which have already been merged into it. The most important amendment used in this work is the *IEEE 802.1Qbv* amendment [IEE16], which introduces "Enhancements for Scheduled Traffic", such as Time-Aware Shaping (TAS) as described in Section 2.2.2.

To enable traffic scheduling in real-time networks, all devices need a synchronized global time, which normally is not part of the Ethernet specification. The *IEEE 802.1AS-2020* amendment [IEE20b] contains changes to the *IEEE 1588 [IEE20a] Precision Time Protocol (PTP)*, which allows to synchronize the clock of TSN capable devices.

There are more TSN amendments, which we do not further discuss, as they are not relevant for our specific use-case.

2.2.2 TSN concepts

One key-feature of the TSN standards is the extension of the queues in each switch as presented in Section 2.1.2. In the following, we give an overview of the main concepts, namely each queue's *Transmission Selection Algorithm (TSA)* and gate and the final *Transmission Selection (TS)*. We discuss these concepts based on Figure 2.4, starting with each queue's TSA.

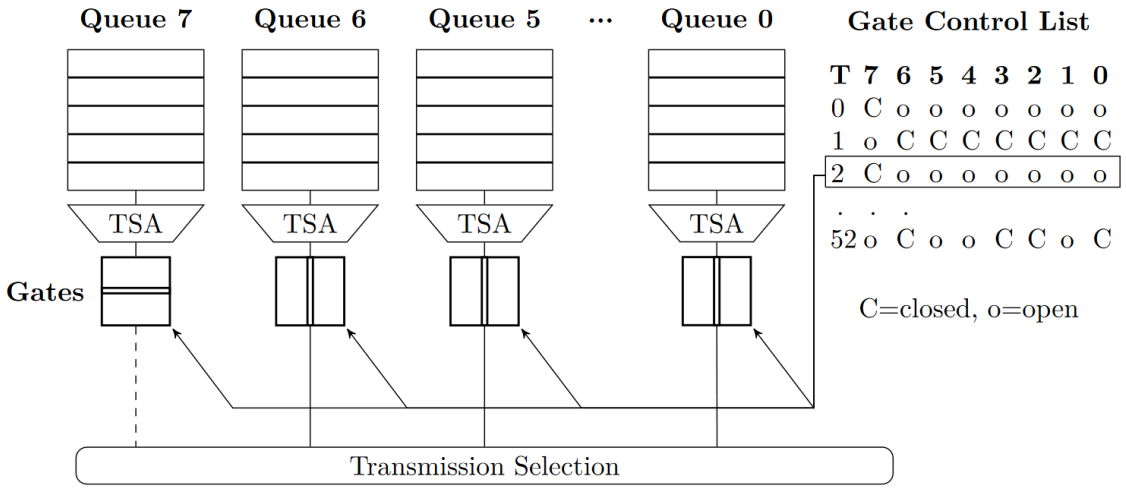


Figure 2.4: Transmission Selection Algorithm (TSA), Gates, Gate Control List (GCL) and Transmission Selection (TS) [HDHK18; IEE18b].

Transmission Selection Algorithm (TSA)

The *Transmission Selection Algorithm (TSA)* is a part of each queue in a TSN capable switch. Main purpose of each queue's TSA is to announce if a frame inside the queue is eligible for transmission at the egress port. This announcement is later used in the *Transmission Selection (TS)* step. The *IEEE 802.1Q* [IEE18b] standard already defines different TSAs. The following list gives a brief overview over these predefined TSAs, but vendor specific TSAs are also possible.

Strict Priority The most basic and trivial TSA is the *Strict Priority* TSA. It always announces a frame ready for transmission if the corresponding queue is not empty.

Credit-Based Shaper (CBS) The *Credit-Based Shaper (CBS)* is another TSA, which was added by the *IEEE 802.1Qav* amendment [IEE10]. Credit-based shaping uses credits which increase when the queue is not transmitting and decrease when the queue is transmitting frames. It only announces a frame eligible for transmission, if the amount of credits is a non-negative value.

Enhanced Transmission Selection (ETS) The third provided TSA is the *Enhanced Transmission Selection (ETS)*. It allows to select frames based on an allocation of bandwidth for different traffic classes.

Gates & Scheduling

After the TSA, each queue has a so-called gate, which holds exactly one of these states at a time [IEE18b]:

- *Open*: Queued frames are announced for transmission, in accordance with the definition of the Transmission Selection Algorithm associated with the queue.
- *Closed*: Queued frames are not announced for transmission.

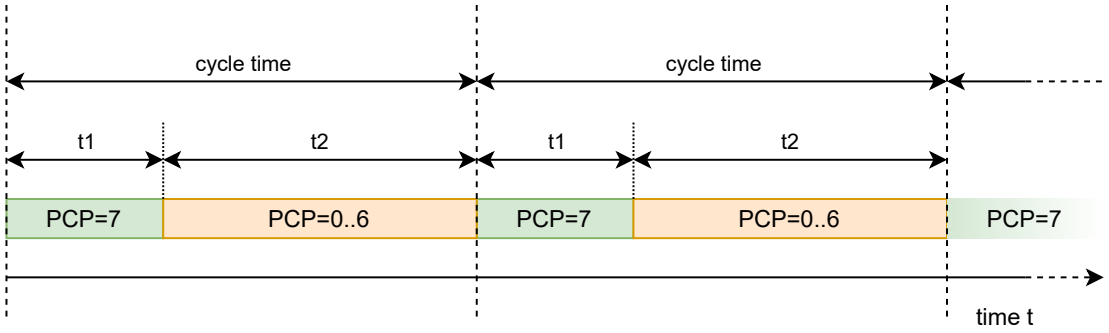


Figure 2.5: Example for Time Division Multiple Access (TDMA) [Fre13] with two timeslots per period.

A Gate Control List (GCL) controls these states of a gate on a time-based cycle. An entry in the GCL is a tuple, which specifies the current state (either Open or Closed) for each gate. Starting with the first entry, the GCL selects the next entry after a specified time interval T. When the end of the GCL is reached, it returns to the first element. The time for a full iteration of the GCL is called *cycle time* and can be calculated by multiplying T with the number of entries in the GCL.

The main purpose of the GCL is to enable *Time-Aware Shaping (TAS)*. This allows us to specify, when a queue is allowed to transmit frames based on an absolute timescale. Time-Aware Shaping is a *Time Division Multiple Access (TDMA)* based approach. TDMA [Fre13] is a technique to split a transmission channel (an Ethernet cable in our case) into multiple repeating timeslots. In each timeslot, only the corresponding queues (one or multiple) are allowed to send. Figure 2.5 shows an TDMA example with two timeslots per period. One period for frames of the highest priority (PCP=7) and one for all other priorities.

Transmission Selection (TS)

The last step in the pipeline is the *Transmission Selection (TS)*. The TS is responsible for the final decision, which frame is transmitted next at the egress port. To achieve this, the TS needs to execute the following steps [IEE18b]:

1. Check each queue's TSA if a transmission-eligible frame is announced.
2. For each of these queues only consider those with an open gate.
3. Filter out queues, if the announced frame's size exceeds the size which can be transmitted in the remaining time the gate stays open (*Guard Band*).
4. Chose the queue with the highest priority from all remaining queues.

2.3 Integer Linear Programming (ILP)

Solving the Routing and Scheduling problem for a specific network topology and streamset is NP-hard [Ste10], and thus we need an efficient algorithm to solve this problem.

One possibility, which we use in this work, is to represent the scheduling and routing conditions and optional goals as constraints and objectives of an *Integer Linear Program (ILP)*. ILPs are a specific type of linear programs, with the only difference, that all variables have to be integers.

An ILP usually consists of multiple parts [Sch86]:

Variables The variables of an ILP can take any integer value and are the part an ILP solver tries to find valid values for.

Constraints add relations between variables and fixed constants. A constraint is either a linear equation (e.g. $a + 2b = c$) or a linear inequality (e.g. $a \leq 2b$). Constraints can be split into three parts, a left-hand side (lhs) ($a + 2b$ or a in the examples above), a sense (either $<$, $\leq$, $=$, $\geq$ or $>$) and a right-hand side (rhs) (c or $2b$ above).

Bounds are an optional possibility to restrict the value range of a variable, e.g. $a \in \{x \in \mathbb{Z} \mid -100 < x < 100\}$.

Objective An objective can be used to specify an optimization goal, which is used by the solver to find the best solution regarding this objective. For example, $\max a$ can be used to find the solution with the biggest valid value for a. Some ILP-solvers offer the functionality to add multiple objectives with weights. However, this is a linear interpolation between these objectives and mathematically equivalent to one single objective.

While small ILPs are rather easy to solve, bigger ILPs need to be solved by specialized tools, so-called ILP-solvers. In this work we mainly focus on a well-known and established ILPs-solver, namely *Gurobi Optimizer* [Gur].

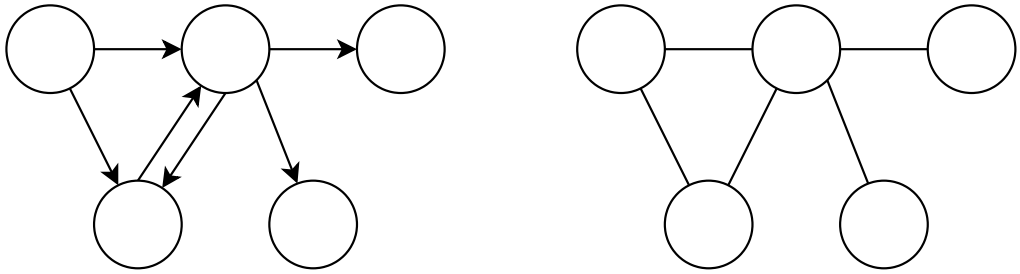
2.4 Graph Algorithms

A possible way to model a network topology is to use a graph. Based on this graph, one can execute different graph algorithms, for example to calculate the shortest path between two devices. In this section, we cover the definition of graphs and two graph algorithms, which we later use in this work. We start by defining a graph and its parts, such as vertices and edges. Then, we introduce the depth-first graph search to find vertices and corresponding paths in a graph. Finally, we explain the Dijkstra algorithm to find the shortest paths from one vertex to a single or multiple other vertices.

2.4.1 Graph

A graph always consists of a set of vertices $\mathcal{V}$. One possibility to represent a vertex is to represent it as a node. Figure 2.6 shows an example with five vertices.

We model connections between vertices by a set of edges $\mathcal{E}$. Each edge is an ordered or unordered pair of vertices (e.g. (v_1, v_2) or $\{v_1, v_2\}$). There are two different basic types of graphs, directed and undirected graphs (Figure 2.6b). In a directed graph, each edge is an ordered pair of vertices, also known as 2-tuples, whereas in an undirected graph they are modeled as unordered pairs, also known as 2-sets. For example, in a directed graph $\mathcal{E} = \{(v_1, v_2)\}$ means v_1 is connected to v_2, but v_2 is not connected to v_1. In contrast, undirected graphs use unordered pairs, so $\{v_1, v_2\}$ is equivalent to



(a) Example of a directed graph. **(b)** Example of an undirected graph.

Figure 2.6: Comparing directed and undirected graphs.

$\{v_2, v_1\}$. This means, that if v_1 is connected to v_2, v_2 is also connected to v_1. Directed edges are represented by arrows connecting nodes (Figure 2.6a), whereas undirected edges are represented as lines (Figure 2.6b).

One can also analyze a graph by different metrics. The only metric we use in this work is the diameter of a graph. The diameter of a graph defines the greatest distance between any two pairs of vertices. This means, that one needs to calculate the shortest path between any two vertices. The greatest of those values is the graph diameter.

2.4.2 Depth-First search (DFS)

The *Depth-First search (DFS)* [Tar71] is a graph search algorithm, which is used to traverse graphs. Algorithm 2.1 shows a possible iterative implementation, but recursive implementations are also possible. The algorithm works by maintaining a visited list and a stack, which initially contains a given start vertex. It then pops element by element from the stack and checks if the popped vertex is on the visited list. If not, the algorithm adds this vertex to the visited list and adds all neighbors to the stack. When there are no more elements left on the stack, the algorithm terminates. By using a stack and immediately pushing neighbors onto it, we ensure the algorithm traverses the graph in a depth-first manner.

Depending on the use-case, we make different modifications to the base algorithm, to achieve a different behavior.

- If we only need to find one target vertex, the algorithm can terminate in Line 6, if the *current* value is the searched target goal.
- To not only find vertices, but also keep track of the paths to these vertices, we introduce a dictionary. The algorithm fills this dictionary with the previous node for each node when visiting it.
- Sometimes, we need the algorithm to return all paths to a given vertex instead of only one path. It is possible to modify the DFS algorithm to support this behavior. In this work we rely on an external tool named graph-tool [Pei14], which already implements this functionality. Thus, we do not discuss this modification in detail.

Algorithm 2.1 Depth-first Search

```
 1: procedure DFS(g: Graph, current: Vertex)
 2:     stack: Stack ← emptyStack()
 3:     stack.push(source)
 4:     visited: Set ← emptySet()
 5:     while not stack.isEmpty() do
 6:         current ← stack.pop()
 7:         if current not in visited then
 8:             visited.add(current)
 9:             for each edge in g.outEdgesOf(current) do
10:                 stack.push(edge.target())
11:             end for
12:         end if
13:     end while
14: end procedure
```

2.4.3 Dijkstra

The Dijkstra algorithm [Dij59] is a graph algorithm either used to calculate the shortest path between a given source vertex and a target vertex (one-to-one Dijkstra) or to calculate the shortest path between a given source vertex and all other vertices (one-to-all Dijkstra).

Algorithm 2.2 shows a possible one-to-all implementation using a priority queue. In the beginning, the algorithm initializes the $dist$ dictionary with 0 for the source vertex and ∞ for all other vertices. It then adds the source vertex to the queue, which orders all contained elements by the currently stored respective value in the $dist$ dictionary. The algorithm now dequeues vertex by vertex, until the queue is empty. It then iterates over all neighbors of that vertex and calculates a temporary distance from the start node via the dequeued vertex to the current vertex. If this temporary distance is higher than the currently stored distance, the algorithm already knows a shorter path to that vertex and continues with the next neighbor. Otherwise, the algorithm stores the temporary distance in the $dist$ dictionary and saves the dequeued vertex in the $prev$ dictionary. Lastly, it queues the current node in the queue.

When the queue is empty, the algorithm returns two dictionaries containing the distance and the previous vertex for each vertex. One can now determine the shortest path to a specific vertex, by reverse searching based on the *prev* dictionary. We can also modify the algorithm to a one-to-one Dijkstra. In order to achieve this, the algorithm returns, when the dequeued vertex in Line 12 matches the target vertex.

Algorithm 2.2 Dijkstra (One-to-All)

```
1:  procedure DIJKSTRA(g: Graph, source: Vertex, weight: Dict[Edge, Number])
2:      dist: Dict[Vertex, Number]
3:      prev: Dict[Vertex, Vertex]
4:      queue: PriorityQueue[Vertex] (Ordered by corresponding dist value)
5:      for each vertex in g.vertices() do
6:          dist[vertex] ← ∞
7:          prev[vertex] ← ?
8:      end for
9:      dist[source] ← 0
10:     queue.enqueue(source)
11:     while not queue.isEmpty() do
12:         current ← queue.dequeue()
13:         for each next in g.neighborsOf(current) do
14:             nextDist ← dist[current] + weight[(current, next)]
15:             if nextDist < dist[next] then
16:                 dist[next] ← nextDist
17:                 prev[next] ← current
18:                 queue.enqueue(next)
19:             end if
20:         end for
21:     end while
22:     return dist, prev
23: end procedure
```

3 Related Work

Scheduling is a well-researched topic for real-time networks as well as for other use-cases. The *Job Shop Scheduling Problem (JSSP)* is one of the earliest researched scheduling problems, e.g. by Graham in 1966 [Gra66]. The main parts of a JSSP are jobs, which consist of a series of operations, and machines, which are able to execute specific operations one at a time. Solving a JSSP, means to calculate a schedule in which machines execute the operations. While the JSSP does not schedule packets in a real-time network, it is still similar to TSN and other real-time network standards, so some approaches for real-time networks are based on the JSSP.

There are also scheduling approaches for real-time networks, before the introduction of TSN. For example, there is work on the scheduling problem for TTEthernet [SAE16] or PROFINET [PRO14], which both extend Ethernet by real-time capabilities. Even though these real-time standards differ from TSN, they are still Ethernet-based and similar to TSN, and thus also relevant for this work.

This chapter gives an overview over some of these already existing approaches. We divide the different approaches into two parts. In the first part, we introduce complete approaches, which either solve the scheduling or the Joint Routing and Scheduling (JRaS) problem. As this work aims to optimize complete approaches, we mainly focus on those. The second part contains a brief overview over heuristic-based and heuristic-enhanced methods.

3.1 Complete Approaches

There are cases, where one needs to ensure the scheduler finds a solution, if there is any valid solution, and otherwise terminates without a solution. In these cases only complete scheduling algorithms are applicable.

We distinguish between two types of complete approaches. The first type contains algorithms, which calculate schedules for streams based on a given route. The second type are so-called Joint Routing and Scheduling (JRaS) approaches, which calculate schedules and routes in the same step. In the following sections, we describe approaches of both types and discuss there respective advantages and disadvantages.

3.1.1 Separate Routing and Scheduling

We start by discussing scheduling approaches, which calculate schedules based on fixed route. This route is either an input to the scheduler or calculated in a previous step. There are multiple ways, to solve the scheduling problem for a given topology and streamset. For example, one can express the scheduling problem as a Constraint Satisfaction Problem (CSP). A possible way to solve a CSP, is

to use a specific, well-defined format, such as *Satisfiability Modulo Theory (SMT)* or *Integer Linear Programming (ILP)*, as described in Section 2.3, and then use already existing solvers to solve the given constraints of the respective format.

Steiner presents such an SMT-based approach to address the scheduling problem for TTEthernet networks [Ste10]. In this work, Steiner represents the constraints of the scheduling problem as an Satisfiability Modulo Theory (SMT) formulation and solves this problem instance using the YICES SMT-solver [Dut14]. Based on the SMT formulation, he evaluates the performance of the schedules, based on different metrics, such as the runtime and maximum utilization of links. Steiner's approach is capable of scheduling a few hundred streamsets within half an hour with the out-of-the-box YICES configuration and up to multiple ten-thounsand streams for a customized YICES back-end. His approach is also capable of validating existing schedules and can be used to debug those.

Craciunas et al. also present an SMT-based scheduling approach for TTEthernet networks [CO14; CO15]. In addition to Steiner's work, they present a combined tt-task and tt-network scheduling approach to cover the whole scheduling solution space and introduce a demand-based scheduling algorithm. In contrast to the one-shot approach, which solves all constraints at once, the demand-based algorithm adds constraints on-demand in order to reduce the runtime. In [CO15], they furthermore introduce an ILP-based approach also for the combined tt-task and tt-network scheduling problem. They compare the performance of the SMT-based and ILP-based approach using YICES [Dut14] and Gurobi [Gur]. Their evaluation states, that the demand-based SMT approach performs better in most cases.

In a later work [COCS16], Craciunas and Steiner et al. adapt their SMT model for IEEE TSN networks. They evaluate this approach on relatively small network typologies with up to seven hosts connected by five switches. Instead, they use a large number of up to 100 streams. The algorithm is capable of scheduling these 100 streams on the given topology within a runtime of 4 hours.

Hanzálek et al. [HBŠ10] developed an ILP-model for another real-time capable Ethernet-based technology called *PROFINET*. In their work, they present an approach for schedule calculation in controller applications. They take real-time traffic from an IO controller to an IO device and vice-versa into account, but there is no real-time traffic between other IO devices. Hanzálek et al.'s approach is capable of calculating an optimal solution for up to one hundred streams on 20 hosts. Additionally, they introduce a heuristic approach Section 3.2 and a rescheduling mechanism, which allows adding messages and hosts to an already calculated schedule.

In [DN16], Dürr et al. propose a no-wait scheduling approach for IEEE TSN-based Ethernet networks. The proposed approach maps the no-wait packet scheduling problem to a no-wait Job Shop Scheduling Problem (JSSP). They provide an ILP for the no-wait JSSP and based on that an ILP for the no-wait packet scheduling problem. To reduce the number of gate opening events in TSN schedules, they also introduce a schedule compression algorithm, which removes gaps between two frames. They achieve this behavior by delaying the first of two frames, so the gap between both transmissions disappears. Delaying frames leads to a relaxation of the no-wait constraint, but their approach does not affect the flowspan and reduces the number of gate-opening events by 24%. Additionally, they reduce the runtime by extending their algorithm with a tabu-search heuristic approach (see Section 3.2).

The scheduling-only approaches mentioned above may be superior in networks with low utilization, due to a shorter solving time compared to JRaS approaches. However, in higher utilized networks, there are cases, where solving the scheduling problem for a given route is impossible, though solutions might exist for different routes. One possibility to circumvent this problem is to execute the scheduler multiple times with different predefined routes, until it finds a solution, assuming there is a solution for a specific set of routes. Another possibility is to use an approach, which solves the routing and scheduling problem at once, so-called *Joint Routing and Scheduling (JRaS)* approaches. We discuss those in the following section.

3.1.2 Joint Routing and Scheduling (JRaS)

In contrast to scheduling-only algorithms, Joint Routing and Scheduling (JRaS) approaches calculate routes and schedules in the same step. As the JRaS problem is an extension of the scheduling problem, similar approaches, such as ILP-based methods, can be used. In the following section, we introduce already existing work on the JRaS problem.

In [SDT+17], Schweissguth et al. present an ILP-based JRaS approach. They also introduce methods to modify this ILP to schedule streams along fixed routes using *Shortest Path Routing* or *Load Balanced Routing*. Their shortest path routing implementation always uses one of the shortest paths, and if there are multiple paths of equal length, they select them based on a round-robin scheme. The load balanced routing is suitable, if there is a low maximum link utilization. This approach leads to the lowest possible link utilization. Finally, they compare the JRaS approach against both fixed-path scheduling approaches. Regarding schedulability, they find out, that the load balanced routing and JRaS are similar, but shortest path routing is only able to schedule $\frac{1}{3}$ of the streams. In contrast, shortest path routing and JRaS are nearly equal regarding latencies, while load balanced routing leads up to 61.8% higher latencies. They also find out, that these better solutions result in longer runtimes.

In a later work, Schweissguth et al. propose another ILP-based apporach, which additionally supports multicast streams [STP+20]. They also provide possible optimizations for this ILP. One of these optimizations is to bound the timeslot on a link, by forcing the latency compared to the first link to be a positive value. They further optimize the runtime, by replacing some constraints in specific cases, e.g. when there is no possible routing decision. Furthermore, they test four different objectives, which include minimizing the cumulative flow latencies (with and without optimality gap), minimizing the path length with a second focus on latencies and having no objective, which means the solver terminates, after finding the first feasible solution. Their evaluation shows, that replacing constraints, when there is no routing decision gives the best performance boost. With this approach, they are able to schedule up to 35 streams within a runtime of 5 s, when using no optimization goal.

Hellmanns et al. aim to reduce the runtime of the Joint Routing and Scheduling (JRaS) problem without sacrificing valid solutions in [HDHK18]. Their approach is also ILP-based, thus, most of the optimizations focus on reducing the ILP's variables and constraints. The ILP by Hellmanns et al. contains two main parts, namely the routing and the scheduling part. They propose an approach for optimizing the routing and scheduling part. First, they optimize the routing part, by using an algorithm called *topology reduction*. Based on its outcome, they also reduce the scheduling part. The topology reduction algorithm calculates all loopless paths from the source host to the destination host and creates a reduced topology only containing those paths. This reduction algorithm removes

all edges in the opposite direction. In some cases, it also removes nodes and their corresponding edges, when a node is not accessible without entering a loop. Based on the topology reduction, they are also able to remove all scheduling variables and constraints for the removed edges, which results in a decimated ILP-model. They find out, that the topology reduction reduces the runtime by a factor of up to 100.

There is not only work on creating and optimizing JRaS approaches, but also work, which explores the impact of different factors regarding the runtime. One example is Falk et al., who evaluate an ILP-based approach on different parameters, to explore the practical limitations of the JRaS problem [FDR18]. They vary parameters, such as the number of flows and the topology size and type (e.g. line, ring and random topologies). Their main finding are, that the runtime is heavily dependent on the number of streams and less influenced by the topology size. They also find out, that the runtime increases, if there are links with a high utilization. Lastly, they state, that the topology type also strongly influences the runtime. Especially topologies with a large amount of links between nodes, result in longer runtimes.

While Joint Routing and Scheduling approaches are able to find feasible solutions more often than separated approaches, they suffer from a higher runtime. As previously mentioned, Falk et al. [FDR18] already analyzed, which factors have a huge impact on the runtime of ILP-solvers. In this work we use these results to reduce the runtime of our ILP-based Joint Routing and Scheduling approach.

3.2 Heuristic approaches

Solving the scheduling problem using a complete or optimal algorithm is reasonable in many cases. But especially in large network environments or networks with a large number of streams, complete approaches may be too slow due to the previously mentioned NP-hardness of scheduling problem (Section 2.3). One possibility to circumvent this problem, is to forego optimality, in favor of a shorter runtime, by using heuristic approaches. Heuristics can also be used to enhance complete approaches. In the following section, we discuss different heuristic-based and heuristic-enhanced scheduling approaches.

In Section 3.1.1, we already discussed Dürr et al.'s no-wait JSSP-based scheduling approach for TSN networks [DN16]. To further enhance this approach, they adapt a tabu-search heuristic for the no-wait JSSP [MMR99] to the no-wait packet scheduling problem. The tabu-search works by finding an initial solution and improves by exploring its neighborhood taking specific tabu criteria into account. The quality difference of the calculated schedules between these two approaches is negligible, but they are able to schedule up to 1500 streams instead of 30 to 50 streams, with this heuristic-enhanced approach.

For Profinet networks, Hanzálek et al. propose two different heuristic-based scheduling approach, namely *Filtered Beam Search (FBS)* and *Iterative Resource Scheduling (IRS)* [HBŠ10]. They find out, that the FBS heuristic is only able to solve small problem statements, and thus does not provide a noteworthy improvement over their optimal ILP approach (Section 3.1.1). The IRS heuristic is able to find a near optimal solution within a much shorter runtime and for larger topologies. In scenarios with a high link utilization however, the IRS heuristic is not able to always find a feasible solution.

Glavackij presents different scheduling-only tracing-based approaches for TSN networks in [Gla20]. Tracing-based approaches use a lightweight network simulator and calculate schedules for streams from the simulation results. However, trivial tracing-based approaches fail, if a single stream violates its deadline requirements. In his work, Glavackij explores different possibilities to handle these violations. The base of all approaches is the same: They first identify all streams that violate their deadline (late streams), and then delay conflicting streams in order to allow late streams to be scheduled before. His first *Naive* approach delays all streams that have a common edge with a late stream. The *Link Time Remaining Time* and *Link Time Late Streams* approach advances this method by redefining what a "conflicting stream" is. More precisely, they do not consider a stream as conflicting, if it is already fully transmitted on arrival of the late stream or if it is queued after the late stream. Additionally, the *Link Time* approaches calculate an overlap-time for each conflicting stream and use this overlap-time to delay streams. The difference between these approaches is the heuristic, which defines how to explore the search tree. Glavackij improves the *Link Time* approaches further by using a Monte Carlo Tree Search (MCTS). He refers to this approach as the *Link Time MCTS* approach. The evaluation shows, that this *Link Time MCTS* has the best scheduling capabilities.

In [PRGS18], Pop et al. present a heuristic scheduling approach for TSN-based fog computing environments. In contrast to other scheduling approaches, Pop et al.'s approach is capable of reconfiguring the schedule at runtime. To reduce the rescheduling runtime, their heuristic algorithm first tries to fit the new streams into the old schedule. If this fails, they recalculate the complete schedule. As a second fallback option, they rely on different design-time approaches, meanwhile, they continue using the old schedule until these approaches finish their calculation. They compare their approach to optimal approaches on low- and medium-utilized networks. While their heuristic algorithm is runtime-superior in these scenarios, they note, that their heuristic scheduler does not guarantee to find a feasible solution in all cases.

As earlier mentioned, the advantage of heuristic-based approaches is their reduced runtime compared to complete scheduling approaches. However, they are not necessarily complete, and thus there is no guarantee, they always find a feasible solution.

4 System Model

Before we are able to explain our optimization approach, we first need to formalize different parts of our problem, such as the network topology and streams. We also need to make assumptions regarding the configuration of our network and its devices. We start this chapter by introducing our network model, which formalizes network topologies and streams. Furthermore, we define helper functions to access attributes of these network parts. In the next section, we explain our assumptions and configuration of network devices. Finally, we introduce a basic ILP-model based on the work by Hellmanns et al. [HDHK18].

4.1 Network Model

For our ILP-model, we need a formal description of our network. In this section, we first give an overview over our network topology model, which contains the hosts, switches and links. Thereafter, we introduce our stream model. Finally, we define some helper functions for frequently used calculations.

4.1.1 Topology

This section contains a description of our network topology model. We start by introducing our representation of the different parts of the network. After that, we explain, how we represent attributes of those network parts.

Our network consists of three main components: hosts, switches and links. As hosts and switches are connected by links, we can model both of them as a set of vertices $\mathcal{V}$. Combined with the representation of links as a set of edges $\mathcal{E}$, they make a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Even though links in an Ethernet network are full-duplex, we use a directed graph and represent a full-duplex link as two opposing edges connecting the participating vertices. This allows us to remove edges in one direction if needed and to schedule streams in both directions independently of each other.

This graph defined above models the structure of our network, but it does not include details such as the device type of vertices or the link speed of edges. Thus, we use global functions, which return those values. This is a list of all global functions regarding a vertex $v \in \mathcal{V}$ or an edge $e \in \mathcal{E}$:

- is_switch : $v \mapsto \begin{cases} 1 & \text{if vertex is a switch} \\ 0 & \text{otherwise (if vertex is a host)} \end{cases}$
- in_edges : $\mathcal{V} \to \mathcal{P}(\mathcal{E})$ gives a set of all incoming links of a vertex v.
- out_edges : $\mathcal{V} \to \mathcal{P}(\mathcal{E})$ give a set of all outgoing links of a vertex v.

- processing_delay : $\mathcal{V} \rightarrow \mathbb{N}$ gives the processing delay, if v is a switch in ns.
- propagation_delay : $\mathcal{E} \rightarrow \mathbb{N}$ gives the propagation delay of a link in ns.
- link_speed : $\mathcal{E} \rightarrow \mathbb{N}$ gives the link speed of a link in $\frac{\text{bit}}{\text{s}}$.

4.1.2 Streams

We model all streams, which are relevant for scheduling, as a set $\mathcal{S} \subset (\mathcal{V} \times \mathcal{V} \times \mathbb{N})$ of streams. Each stream object $(v_{\text{src}}, v_{\text{dst}}, \text{id}) \in \mathcal{S}$ has three properties, which describe the source vertex, target vertex, and the id of a stream. Each stream id is unique and may only belong to exactly one element in $\mathcal{S}$. In some cases, we only need the source and target vertex of a stream. In these cases, we leave out the id and use $(v_{\text{src}}, v_{\text{dst}}, \_) \in \mathcal{S}$ as a simplified notation.

Similar to our network topology, we access attributes of the streams by using global functions. The following list contains functions which map a stream $s = (v_{\text{src}}, v_{\text{dst}}, \text{id}) \in \mathcal{S}$ to a given attribute of the stream:

- stream_size : $\mathcal{S} \rightarrow \mathbb{N}$ gives the size of a stream in Bytes.
- e2e_delay : $\mathcal{S} \rightarrow \mathbb{N}$ gives the maximum end-to-end delay in ns.
- stream_vertices : $\mathcal{S} \rightarrow \mathcal{P}(\mathcal{V})$ gives all vertices in the reduced topology of a stream (see Section 6.2.1).
- stream_edges : $\mathcal{S} \rightarrow \mathcal{P}(\mathcal{E})$ gives all edges in the reduced topology of a stream (see Section 6.2.1).

4.1.3 Helper Functions

There are repeating calculations using streams $s \in \mathcal{S}$, vertices $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$ in this work. To allow an easy usage of these values, we define functions to express those values:

- transmission_delay$(s, e) = \frac{8 \cdot \text{stream_size}(s)}{\text{link_speed}(e)} \cdot 10^9$ calculates the transmission delay of a stream s on edge e in ns.
- lower_bound$(s, e) : \mathcal{S} \times \mathcal{V} \rightarrow \mathbb{N}$ gives the lower bound of a stream s on edge e in ns (see Section 6.3.2) .
- upper_bound$(s, e) : \mathcal{S} \times \mathcal{V} \rightarrow \mathbb{N}$ gives the upper bound of a stream s on edge e in ns (see Section 6.3.2).

4.2 Ethernet Model

In order to correctly calculate schedules and convert them to a Gate Control List (GCL), we need a model of our Ethernet network. The Ethernet model in this section contains assumptions and configurations on different parts of the network and its devices.

We distinguish between time-critcial and best-effort traffic. We only calculate schedules for time-critical streams, and assume that they always use the highest priority, which means, they have a Priority Code Point (PCP) value of 7. As we do not take best-effort traffic into account, all other PCP values are available for this traffic class. We only distinguish between time-critical and best-effort frames using this PCP value. Thus, we do not use other values of the VLAN tag, such as the VLAN Identifier (VID). As time-critical frames may never be discarded in our case, the Drop Eligible Indicator (DEI) is always 0 for time-critical frames.

In our work, we do not take multicast streams into account. This means, a stream always has exactly one sender and exactly one receiver. Still, a host may be the sender and/or receiver of multiple streams. That means, we are not able to distinguish streams based on the MAC address of its sender or receiver. As mentioned in Section 2.1.2, a switch may forward a frame based on the destination MAC address or a *Flow Filtering* tag. For the sake of simplicity and generality, we do not rely on the flow-based approach in this work. Instead, we assign a multicast MAC address to each stream, which only contains the single receiver of this stream in the corresponding multicast group.

While the TSN standards define the general behavior of a TSN-capable switch, the implementation and supported functionality of the forwarding process differs between various devices. Nevertheless, we have to ensure our schedules are applicable to these switches. Therefore, we demand that all switches have at least two different queues, while one queue only contains our time-critical frames with PCP = 7. Furthermore, we assume that the processing delay of a switch is constant and especially independent of a frame's size. To fulfill our demand of a No-wait schedule, we need to ensure all switches are able to immediately forward a frame without a queuing delay. Thus, we configure all switches to use the *Strict Priority* Transmission Selection Algorithm. Lastly, we require all switches in our network to use the same switching type. This can be either Store-and-Forward or Cut-Through, as explained in Section 2.1.3.

4.3 ILP-model

As mentioned in Section 3.1.2, different approaches to model the *Joint Routing and Scheduling (JRaS)* as an ILP already exist. Therefore, we decided to improve upon an already existing ILP formulation. We use the ILP-model by Hellmanns et al. [HDHK18] in this work, as it is directly based on the networks topology and streamset, and hence allows us to directly project changes of the network topology and streamset to the ILP formulation.

In this section, we describe Hellmanns et al.'s basic ILP formulation in detail. We separate the constraints of the ILP-model into three different groups, namely the *Routing*, *Scheduling* and *Conflict* group. The following sections define the variables and constraints of each group.

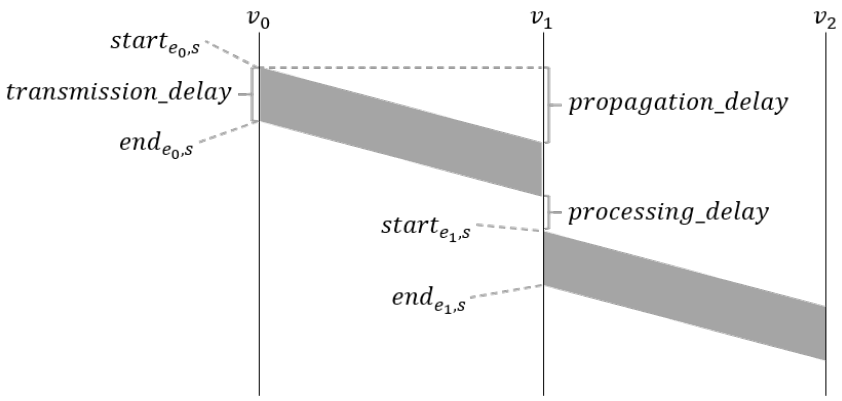


Figure 4.1: Definition of all delays, start variables and end variables for Store-and-Forward switching.

4.3.1 Routing Constraints

To represent a routing decision, which defines, whether a stream $s \in \mathcal{S}$ uses an edge $e \in \mathcal{E}$, Hellmanns et al. introduce a binary decision variable $x_{s,e}$ which is 1 if the stream s uses the edge e and 0 otherwise.

Using these variables, they are able to define all constraints relevant for the routing decision. Constraint (4.1) and Constraint (4.2) limit the number of outgoing used edges of the source node and incoming used edges of the destination node to one. These constraints enforce, that each stream has to start at its given start vertex and stop at its given end vertex. To ensure, that streams cannot have any other start or end point, Constraint (4.3) enforces, that the number of used incoming and used outgoing edges of each vertex has to be equal. In combination, all three routing constraints provide a continuous flow from the start vertex to the end vertex.

$$\forall s = (v_{\text{src}}, v_{\text{dst}}, \_) \in \mathcal{S} :$$

$$\sum_{e \in \text{out\_edges}(v_{\text{src}})} x_{s,e} = 1 \tag{4.1}$$

$$\sum_{e \in \text{in\_edges}(v_{\text{dst}})} x_{s,e} = 1 \tag{4.2}$$

$$\forall v \in \mathcal{V} \setminus \{v_{\text{src}}, v_{\text{dst}}\} : \sum_{e \in \text{in\_edges}(v)} x_{s,e} = \sum_{e \in \text{out\_edges}(v)} x_{s,e} \tag{4.3}$$

4.3.2 Scheduling Constraints

The main task of the scheduling constraints is to provide a schedule for each stream. Hellmanns et al. use two variables per Edge e and Stream s to define the start and end time. The $start_{s,e}$ variable, defines the start time of Stream s on Edge e and $end_{s,e}$ holds the end time of s on e, as Figure 4.1 shows.

To ensure, that a stream does not exceed the cycle time, Hellmanns et al. provide a constraint for each stream, which limits all end variables to the cycle time. By multiplying the cycle time with the decision variable $x_{s,e}$, they also achieve that the time for each unused edge is set to zero.

$$\forall s \in \mathcal{S}, \forall e \in \mathcal{E} : end_{s,e} \leq x_{s,e} \cdot \text{cycle\_time} \tag{4.4}$$

The end variable $end_{s,e}$ only depends on the start variable $start_{s,e}$ and the transmission delay of the same edge. Constraint (4.5) represents this dependency between the start and end time of each stream and edge. When Edge e is used, Constraint (4.5) calculates the end time $end_{s,e}$ by adding the transmission delay to the corresponding variable $start_{s,e}$. If e is unused, this constraint in combination with Constraint (4.4) ensures, that both variables are set to zero.

$$\forall s \in \mathcal{S}, \forall e \in \mathcal{E} : end_{s,e} = start_{s,e} + x_{s,e} \cdot \text{transmission\_delay}(s, e) \tag{4.5}$$

To schedule streams continuously in a No-wait manner, Hellmanns et al. provide another scheduling constraint. On the rhs, Constraint (4.6) sums the end times of all incoming edges and adds the propagation and processing delay based on the routing variable $x_{s,e}$. On the lhs, it sums the start time of all outgoing edges. By requiring both sides to be equal, this constraint fulfills the needs to generate a continuous No-wait schedule.

$$\begin{aligned} &\forall s = (v_{\text{src}}, v_{\text{dst}}, \_) \in \mathcal{S}, \forall v \in \mathcal{V} \setminus \{v_{\text{src}}, v_{\text{dst}}\} : \\ &\sum_{e \in \text{out\_edges}(v)} start_{s,e} \\ &= \sum_{e \in \text{in\_edges}(v)} end_{s,e} + x_{s,e} \cdot (\text{propagation\_delay}(e) + \text{processing\_delay}(v)) \end{aligned} \tag{4.6}$$

4.3.3 Conflict Constraints

To prohibit the collision of two streams s and s' on an edge e, a constraint to disallow streams to overlap on an edge is needed. Therefore, Hellmanns et al. introduce a new Variable $b_{s,s',e}$ for each pair of distinct streams (s, s') on each edge. $b_{s,s',e}$ intuitively reads as "*stream s is scheduled before stream s′ on edge e*".

To be able to model this behavior in an ILP, we need to use *big M* constraints. A big M variable holds a value bigger than the maximum value any other variable in this constraint can normally take. By multiplying the binary decision variable $b_{s,s',e}$ with M, Hellmanns et al. are able to define a set of two constraints, which model either the time for s before s' or s' before s based on the decision variable b:

$$\begin{aligned} &\forall((s,s'),e) \in (\mathcal{S}\times\mathcal{S})\times\mathcal{E} \text{ (with } s \neq s') : \\ &end_{s,e} \leq start_{s',e} \quad + (1 - b_{s,s',e}) \cdot M \\ &end_{s',e} \leq start_{s,e} \quad + b_{s,s',e} \cdot M \end{aligned} \tag{4.7}$$

Choosing too big values for M may lead to numerical issues [Gur], but we can simply choose M as the cycle time, as the end time of a stream can never exceed the cycle time.

5 Problem Statement

To ensure time-sensitive frames arrive within their specified time limit, they need to be separated from non-time-critical (best-effort) traffic. We can achieve this, by distributing the available bandwidth between both traffic classes in such a way, that all time-critical frames arrive within their time limit. To do this, we need to generate a so-called *Schedule*, which contains an exact time-allocation for each link on a frame's route. This is also known as the *Scheduling* process. Based on a schedule, one can calculate entries for the *Gate Control List (GCL)*, as explained in Section 2.2.2. Calculating minimum-sized timeslots for real-time traffic requires knowledge about all delays in the network, including the transmission delay, and thus we need to know the size of all time-sensitive frames. The calculation of timeslots with knowledge of each frame's size is proven to be NP-hard [Ull75].

Scheduling-only algorithms require a given route, which is either an input to the scheduler or calculated in an earlier preprocessing step. Using a fixed route normally reduces the algorithm's complexity and runtime, but also excludes possible solutions in advance. An extension of the scheduling problem, which circumvents this downside, is the so-called *Joint Routing and Scheduling (JRaS)* problem. JRaS approaches calculate routes and schedules in the same step, and thus cover the whole solution space. Another modification of scheduling and JRaS approaches are so-called *No-Wait Schedules*. These type of schedules prohibit intermediate switches to keep frames in the queue. This means, each switch immediately redirects a frame, after finishing the internal processing pipeline (see Section 2.1.2). Furthermore, this implies, that frames may never pass other time-critical frames inside a switch.

We already introduced Hellmanns et al.'s base ILP-model for the *No-wait Joint Routing and Scheduling (JRaS)* problem in Section 4.3. Figure 5.1 visualizes the scheduling capability of this approach for test cases with up to 150 streams. We can see, it is not able to provide a valid schedule for all test cases within a runtime limit of 900 s per test case, and due to the earlier mentioned NP-hardness it does not scale for a larger numbers of streams. Thus, our goals are to improve the scheduling capabilities and to reduce the runtime of this approach. In Figure 5.1, these goals corresponds to a larger percentage of solved streamsets and to a steeper increase of the cumulative curve.

The complexity of the problem lies in the number of variables and constraints of our ILP-model. Thus, optimizing the ILP-model means to generate less variables and constraints. As the ILP-model is directly based on the topology model and streams, we can either do this by modifying the underlying topology model or by altering our ILP generation process. Additionally, we are able to make use of the ILP-solver specific functionalities. In the next Chapter 6 we present our approaches with which we aim to reach our goals.

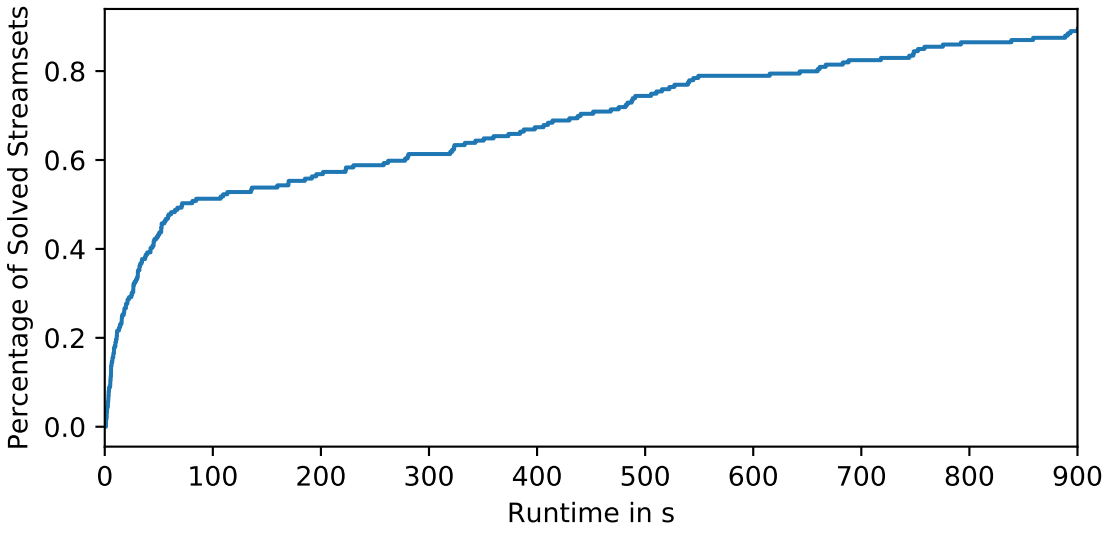


Figure 5.1: Scheduling capability and runtimes of the base ILP-model.

6 Design

As we already discussed in Chapter 3, there are many approaches which address the *Scheduling Problem* or the *Joint Routing and Scheduling (JRaS) Problem*. In this work, we aim to improve and optimize an already existing ILP-based JRaS approach and explore the influence of those optimizations. As a first step in this chapter, we extend the base ILP-model with additional functionality. This includes support for Cut-Through switching and stream-specific end-to-end delays.

In Chapter 5, we already mentioned, that the scalability of our approach depends on the ILP-model size, and thus on the number of variables and constraints in the ILP. Furthermore, the three major components of the base ILP (routing, scheduling and conflict constraints) are directly based on the model of the underlying topology. Thus, one possibility to optimize the size of the ILP is to assess each of these components by reducing our topology model. In the first optimization part of this chapter, we analyze different possibilities to reduce our topology model and then introduce optimization approaches based on our findings.

Instead of modifying the underlying topology model, we are also able to change the way we generate the ILP from our topology model. In the second optimization part, we analyze our ILP model generation process. We find out, that we are able to remove some unnecessary auxiliary variables and that we are also able to limit the solution space by providing additional variable bounds.

Finally, we aim to reduce the runtime of the ILP-solver without modifying the structure of the ILP itself. For this, we explore specific functionalities of the Gurobi ILP-solver. For example, we make use of different constraint types, but also provide solution hints for the variables in our ILP. We also introduce an approach to reduce the runtime by modifying parameters, which influence the solving process itself.

6.1 ILP Enhancements

In this first section, we provide multiple ILP-enhancements. We found out, that under certain conditions, the base ILP can lead to incorrect results. In a first step, we fix this by adding a new constraint, which prohibits loops. After that, we present two extensions for our ILP. In our first addition, we modify scheduling constraints in order to support Cut-Through switching. Our second contribution is a new constraint, which allows stream-specific End-to-End deadlines.

6.1.1 Prohibit Loops

While verifying the results of our scheduler, we noticed a scenario similar to the scenario shown in Figure 6.1. In this scenario our scheduler generated the following schedule on the green path ($1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow 8$):

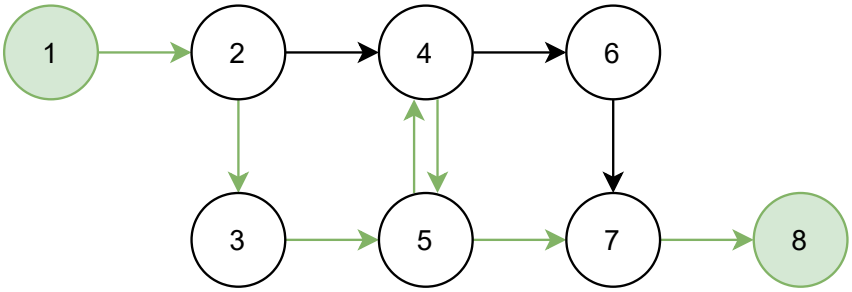


Figure 6.1: Problematic topology, which might lead to invalid schedules.

$$
\begin{aligned}
start_{s,(1,2)} &= 0; & end_{s,(1,2)} &= 1{,}000 \\
start_{s,(2,3)} &= 3{,}000; & end_{s,(2,3)} &= 4{,}000 \\
start_{s,(3,5)} &= 6{,}000; & end_{s,(3,5)} &= 7{,}000 \\
start_{s,(5,4)} &= 0; & end_{s,(5,4)} &= 1{,}000 \\
start_{s,(4,5)} &= 3{,}000; & end_{s,(4,5)} &= 4{,}000 \\
start_{s,(5,7)} &= 15{,}000; & end_{s,(5,7)} &= 16{,}000 \\
start_{s,(7,8)} &= 18{,}000; & end_{s,(7,8)} &= 19{,}000
\end{aligned}
$$

In this scenario, all delays had a constant value of 1,000 ns (processing_delay = propagation_delay = transmission_delay = 1,000 ns). We found out, that this schedule is incorrect. This trivial to show, as the values for $start_{s,(5,4)}$ and $start_{s,(4,5)}$ are invalid in a real network, as the stream's source is Vertex 1, thus the stream cannot start from Vertex 4 at time 0. But based on the scheduling Constraint (4.6), which defines the start time on consecutive edges, the calculated schedule is valid. We show this for the critical Vertices 4 and 5:

For Vertex 4:

$$
\begin{aligned}
\sum_{e \in \text{out\_edges}(4)} start_{s,e} &= start_{s,(4,5)} \\
&= 3{,}000 \\
&= 1{,}000 + 1 \cdot (1{,}000 + 1{,}000) \\
&= end_{s,(5,4)} + 1 \cdot (\text{propagation\_delay}((5,4)) + \text{processing\_delay}(4)) \\
&= \sum_{e \in \text{in\_edges}(4)} end_{s,e} + x_{s,e} \cdot (\text{propagation\_delay}(e) + \text{processing\_delay}(4))
\end{aligned}
$$

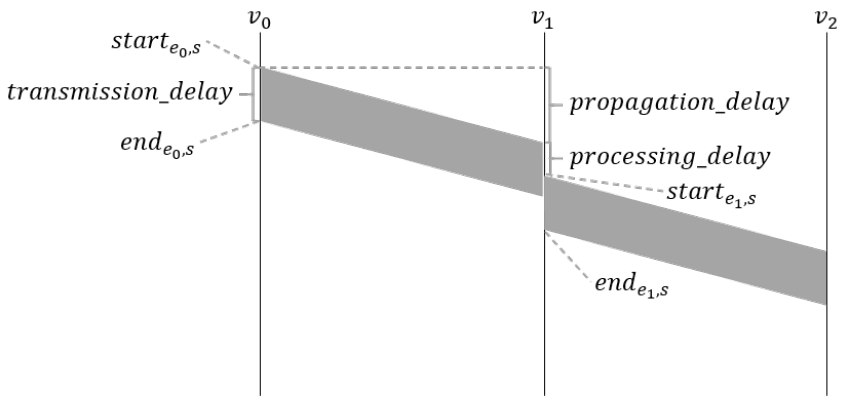


Figure 6.2: Definition of all delays, start variables and end variables for Cut-Through switching.

For Vertex 5:

$$\begin{aligned}
\sum_{e \in \text{out_edges}(5)} start_{s,e} &= start_{s,(5,4)} + start_{s,(5,7)} \\
&= 0 + 15{,}000 \\
&= 15{,}000 \\
&= 7{,}000 + 1 \cdot (1{,}000 + 1{,}000) + 4000 + 1 \cdot (1{,}000 + 1{,}000) \\
&= end_{s,(3,5)} + 1 \cdot (\text{propagation_delay}((3,5)) + \text{processing_delay}(5)) \\
&+ end_{s,(4,5)} + 1 \cdot (\text{propagation_delay}((4,5)) + \text{processing_delay}(5)) \\
&= \sum_{e \in \text{in_edges}(5)} end_{s,e} + x_{s,e} \cdot (\text{propagation_delay}(e) + \text{processing_delay}(5))
\end{aligned}$$

Figure 6.1 shows, that the main problem is, that the stream passes Vertex 5 two times. To prevent this, we introduce a new constraint which disallows to route streams two times over the same vertex. In other words, a stream may only use a maximum of one incoming edge for each vertex:

$$\forall s \in \mathcal{S}, \forall v \in \mathcal{V} : \sum_{e \in \text{in\_edges}(v)} x_{s,e} \leq 1 \tag{6.1}$$

6.1.2 Support of Cut-Through Switching

Hellmanns et al.'s ILP formulation solely supports *Store-and-Forward* switching. As a first contribution, we present a modification to this ILP, which allows us to schedule streams in a network with Cut-Through switches. Similar to the scheduling definition in Section 4.3.2, we first define a start variable $start_{s,e}$ and an end variable $end_{s,e}$ for each Stream s on Edge e. Figure 6.2 shows a visualization of our definition. For Store-and-Forward switching, the start time on the next edge depends on the end time of the previous edge. In Cut-Through switching however, start variables of outgoing edges only depend on the start variables of incoming edges. To this end, we need to

replace the end variable $end_{s,e}$ on the rhs of Constraint (4.6) with the start variable $start_{s,e}$. As Constraint (6.2) shows, neither the end variable, nor the transmission delay is needed to schedule streams.

$$\forall s = (v_{\text{src}}, v_{\text{dst}}, \_) \in \mathcal{S}, \forall v \in \mathcal{V} \setminus \{v_{\text{src}}, v_{\text{dst}}\} :$$

$$\sum_{e \in \text{out\_edges}(v)} start_{s,e} \tag{6.2}$$
$$= \sum_{e \in \text{in\_edges}(v)} start_{s,e} + x_{s,e} \cdot (\text{propagation\_delay}(e) + \text{processing\_delay}(v))$$

6.1.3 Stream-specific End-to-end Limit

In some cases, one needs different end-to-end limitations for different streams. We provide a new Constraint (6.3), which optionally provides this functionality. This constraint subtracts the departure time $start_{s,e}$ at the source vertex from the arrival time $end_{s,e}$ at the target vertex and requires the difference to be lower than the provided end-to-end limit. Constraint (4.1) and Constraint (4.2) require, that there is exactly one outgoing edge at the source and one incoming edge at the target. Thus, we define a new constraint, which subtracts the start time on the first edge from the end time on the last edge and requires this difference to be smaller than the given end-to-end limit:

$$\forall s = (v_{src}, v_{dst}, \_) \in \mathcal{S}$$
$$\sum_{e \in \text{out\_edges}(v_{src})} start_{s,e} - \sum_{e \in \text{in\_edges}(v_{dst})} end_{s,e} \leq \text{e2e\_delay}(s) \tag{6.3}$$

We only provide this constraint as an optional addition. It is not part of our default ILP-model.

6.2 Topology Model Optimizations

As mentioned before, one way to reduce the size of the ILP-model is to reduce the underlying topology model. In this work, we mainly consider hierarchical factory automation networks (e.g. Figure 6.3). These topologies consist of interconnected line and ring subtopologies. The base ILP would generate routing, scheduling and conflict constraints for each vertex, edge and stream in this model. When analyzing these factory automation networks however, we noticed, that we do not need each of these constraints for each stream. As an example, there are edges and vertices, which a stream may never pass without entering a loop. We can also see, that there are no routing decisions in line topologies and only two routing decisions for ring topologies per stream. This leads us to the finding, that we are able to reduce our model by discarding constraints if we omit or replace vertices and edges in a stream-specific network topology model.

In this section we present two different types of optimization approaches, which both reduce the toplogy model on a per-stream basis. First, we focus on reducing the ILP model by discarding edges and vertices which are with a high probability not part of a solution. Therefore, we analyze

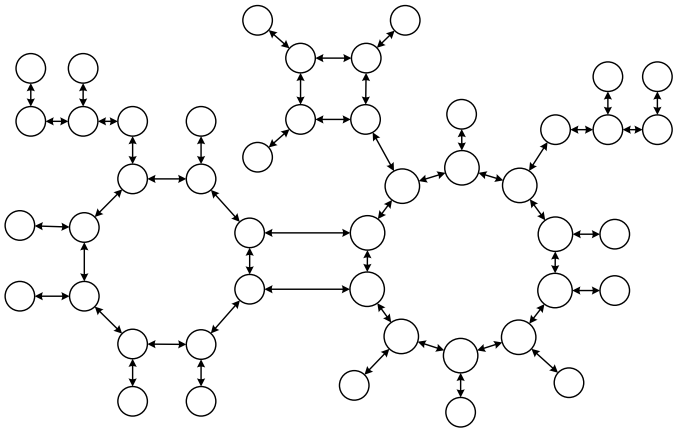
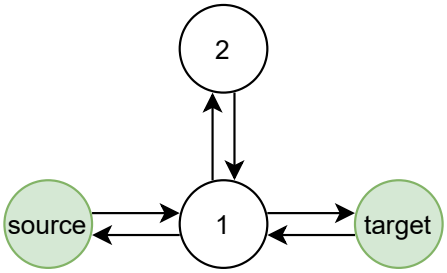
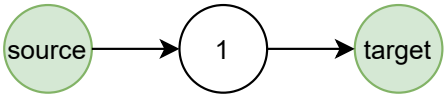


Figure 6.3: Example of a factory automation network [HDHK18].



(a) Graph before executing the topology reduction algorithm.



(b) Graph after executing the topology reduction algorithm.

Figure 6.4: This figure shows, how the topology reduction algorithm removes inaccessible vertices and edges.

the structure of typical factory automation networks and explain why discarding certain edges and vertices does not limit the solvability. After that, we further analyze routing possibilities in these networks, and based on that we provide further optimizations by merging vertices and edges on paths without routing decisions.

6.2.1 Remove Vertices on Invalid Paths

In Figure 6.4a, we can see an example topology and a given source and target vertex for a stream. As earlier mentioned, only loopless paths from the source to the target vertex are allowed. We refer to these paths as *valid paths*. This means, the only valid path in this example is source $\rightarrow 1 \rightarrow$ target, and thus all other paths are invalid. That implies, the given stream may never pass vertex 2 and edges including vertex 2. Also, it may never use edges in the opposite direction ($(1, \text{source})$ and $(\text{target}, 1)$). These findings allow us to remove all variables and constraints of vertices and edges, which are only part of invalid paths.

Hellmanns et al. already introduced a *Topology Reduction* algorithm, which identifies vertices and corresponding edges of invalid paths. This algorithm calculates all loopless paths for each stream's source-target combination using the depth-first search algorithm (Section 2.4.2). It then generates a new, stream-specific topology model, which contains all vertices and edges used in any of those paths, and makes them accessible via our global functions stream_vertices and stream_edges (Section 4.1). Figure 6.4a shows an example graph and Figure 6.4b shows the same graph processed by the topology reduction algorithm.

All of our previously introduced constraints are either generated for each edge or generate a sum using the in- or out-edges of a vertex. For the sake of simplicity we now assume, that we only use edges $e \in \text{stream\_edges}(s)$ to generate these constraints.

6.2.2 Cutoff in Path Calculation

In Section 2.4.1, we already introduced the diameter metric of a graph, which describes the longest distance between any two vertices. This especially means, that there exists at least one path between any two vertices, which does not exceed the size of the diameter. As the calculation of the diameter is inefficient, graph tool uses an approximation to calculate the so-called pseudo-diameter. Using this pseudo-diameter as a cutoff in the DFS reduces the runtime of the graph search, but still gurantees to find a path between the source and the target vertex. In some cases, using the cutoff may also discard paths, but as they are longer than the graph diameter, they are most likely not part of a valid solution. This means, we now have a smaller amount of edges in stream_edges. It is possible to prohibit discarding longer paths by adding a margin to the diameter.

Again, for the sake of simplicity, we assume constraints are only generated, if stream_edges contains the respective edge.

6.2.3 Merge Edges

Our previous optimizations already reduced stream-specific topologies. We already mentioned, that there is only one routing possibility in line subtopologies and two routing possibilities for ring subtopologies in our example factory automation network (Figure 6.3). We now introduce a smaller example topology in Figure 6.5a. In this example, Vertex 2, Vertex 3 and Vertex 4 do not contain routing choices. As an example, the Routing Constraint (4.3) of Vertex 4 looks like this: $x_{s,(\text{source},4)} = x_{s,(4,\text{target})}$. This means, if a stream uses the only in edge of Vertex 4, it also has to use its only out edge. Thus, we are able to discard this vertex from the stream-specific topology model and merge its in and out edge for our routing decision. In a first step, we use these merged edges in order to reduce the number of routing constraints. Based on this, we then analyze if and how we are able to utilize merged edges for the reduction of scheduling and conflict constraints.

In Algorithm 6.1, we present our *Edge Merger* algorithm, which merges edges on paths without routing choices. Our algorithm starts with the source vertex of a stream and then recursively continues with all following edges. For each edge, it checks if the source vertex of this edge is redundant. We define a vertex as redundant, if it has exactly one outgoing and exactly one incoming edge, and thus is not part of any routing decision. In this case, we remove the redundant vertex and merge the incoming and the outgoing edge. This means, that two edges, e.g. (a, b) and (b, c) with the redundant vertex b become a new edge (a, c). We then remove the redundant vertex and both

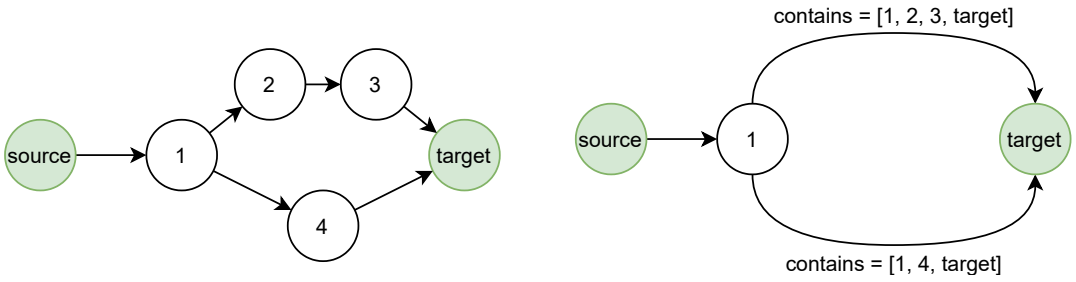
Algorithm 6.1 Edge Merger

```
1:  global connects: Dict[Edge, List[Vertex]]
2:  global g: Graph
3:  global visitied: Set[Edge] ← emptySet()
4:
5:  function INIT(stream: Stream)
6:      g ← new Graph(stream.vertices, stream.edges)          // See Sections 6.2.1 and 6.2.2
7:      for each edge in g.edges do
8:          connects[edge] ← [edge.src, edge.tgt]              // Initialize connects list
9:      end for
10:     MERGERECURSIVE(stream.src)          // Start recursive algorithm with source of stream
11: end function
12:
13: function MERGERECURSIVE(curr: Edge)
14:     if curr not in visited then
15:         return                                             // Edge already processed
16:     end if
17:     visited.add(curr)
18:     if curr.src.inDegree = 1 and curr.src.outDegree = 1 then      // We are able to merge
19:         prevEdge ← curr.src.inEdge                         // There is only one inEdge
20:         newEdge ← g.addEdge(prevEdge.src, curr.tgt)        // newEdge "skips" curr.src vertex
21:         contains[newEdge] ← contains[prevEdge] + curr.tgt  // Update mapping
22:         g.removeEdge(curr)                                 // Remove redundant edge from curr.src
23:         g.removeEdge(prevEdge)                             // Remove redundant edge to curr.src
24:         g.removeVertex(curr.src)                           // Remove redundant vertex
25:     end if
26:     for each edge in curr.tgt.outEdges do
27:         MERGERECURSIVE(edge)                               // Recursive call for all following edges
28:     end for
29: end function
```

edges and replace them by the new merged edge. Figure 6.5 visualizes our *Edge Merger*. For our following optimizations, we also keep track of which edge "contains" which vertices. We always update this mapping, when we merge two edges. Figure 6.5b shows this for an example topology. It is important to note, that our *Edge Merger* works on a temporary topology per stream and also generates the *contains*-mapping for each single stream, as we need to access both of them in our further optimizations.

6.2.4 Routing Optimization

We already mentioned above, that we can use the merged edges to reduce the of routing constraints. Instead of generating a routing constraint for each edge $e \in$ stream_edges, we now only generate a routing variable for each edge in our temporary topology generated by our *Edge Merger*. This means, we now only have routing variables, when there is a real routing decision. To differentiate between two merged edges connecting the same two nodes, we introduce a new notation: $a \rightarrow b \rightarrow c$ is



(a) Graph before merging edges.

(b) Graph after merging edges including the contains mapping.

Figure 6.5: Visualization of our Edge Mapper algorithm for an example topology.

the edge (a, c), which was created by merging edge (a, b) and (b, c). This notation can hold an arbitrary amount of in-between vertices, e.g. $a \rightarrow b \rightarrow c \rightarrow d$ but also $a \rightarrow b$. We generate this notation based on the *contains*-mapping. For our example topology in Figure 6.5, we would create these three routing variables: $x_{s,source \rightarrow 1}$, $x_{s,1 \rightarrow 2 \rightarrow 3 \rightarrow \text{target}}$ and $x_{s,1 \rightarrow 4 \rightarrow \text{target}}$. We also only create routing constraints for these variables. If we need to access the routing variable for any in-between removed edge, we always fall back to the routing variable of our merged edge. For example, this means accessing a routing variable for $(1, 2)$, $(2, 3)$ or $(3, target)$ always uses the routing variable $x_{s,1 \rightarrow 2 \rightarrow 3 \rightarrow \text{target}}$. This is an optional optimization, which the user may enable or disable.

6.2.5 Scheduling Optimization

We already showed, that our *Edge Merger* allows us to reduce the number of routing constraints. Previously, we generated scheduling variables for each edge in the remaining network topology. If we now take a look at Figure 6.5a, we can see the scheduling variable $start_{s,1 \rightarrow 2}$ and the routing variable $x_{s,1 \rightarrow 2 \rightarrow 3 \rightarrow \text{target}}$ directly define the start variable for all following edges, such as $start_{s,2 \rightarrow 3}$ and $start_{s,3 \rightarrow \text{target}}$, as there is no intermediate routing decision. This means, schedules only depend on the first edge of this path, and thus we only need to create a scheduling variable for the first edge on a mergeable path. For all other edges, we create a pseudo-var, which is a linear expression of the first start var and the corresponding delays. In every constraint, where we access a scheduling variable, we check, whether it is a real or a pseudo-variable and insert it respectively. If a constraint only contains pseudo-vars, we are able to omit it. This is the case for all intermediate nodes on a merged path. For our example in Figure 6.5a, we would create the following variables and linear expressions on the upper path. We only show this exemplary for the start variables, as it works similar for the end variables:

1. Variable: $start_{s,1 \rightarrow 2}$
2. PseudoVariable: $start_{s,2 \rightarrow 3}$

   For Store-and-Forward Switching:

$$\begin{aligned} start_{s,2 \rightarrow 3} = start_{s,1 \rightarrow 2} + x_{s,1 \rightarrow 2 \rightarrow 3 \rightarrow \text{target}} \cdot (&\text{transmission\_delay}(s, 1 \rightarrow 2) \\ &+ \text{propagation\_delay}(1 \rightarrow 2) \\ &+ \text{processing\_delay}(2)) \end{aligned}$$

For Cut-Through Switching:

$$start_{s,2\to3} = start_{s,1\to2} + x_{s,1\to2\to3\to\text{target}} \cdot (\text{propagation\_delay}(1\to2) + \text{processing\_delay}(2))$$

3. PseudoVariable: $start_{s,3\to\text{target}}$

   For Store-and-Forward Switching:

$$\begin{aligned} start_{s,3\to\text{target}} &= start_{s,2\to3} + x_{s,1\to2\to3\to\text{target}} \cdot (\text{transmission\_delay}(s, 2\to3) \\ &\quad + \text{propagation\_delay}(2\to3) \\ &\quad + \text{processing\_delay}(3)) \\ &= start_{s,1\to2} + x_{s,1\to2\to3\to\text{target}} \cdot (\text{transmission\_delay}(s, 1\to2) \\ &\quad + \text{propagation\_delay}(1\to2) \\ &\quad + \text{processing\_delay}(2)) \\ &\quad + x_{s,1\to2\to3\to\text{target}} \cdot (\text{transmission\_delay}(s, 2\to3) \\ &\quad + \text{propagation\_delay}(2\to3) \\ &\quad + \text{processing\_delay}(3)) \end{aligned}$$

   For Cut-Through Switching:

$$\begin{aligned} start_{s,3\to\text{target}} &= start_{s,2\to3} + x_{s,1\to2\to3\to\text{target}} \cdot (\text{propagation\_delay}(2\to3) \\ &\quad + \text{processing\_delay}(3)) \\ &= start_{s,1\to2} + x_{s,1\to2\to3\to\text{target}} \cdot (\text{propagation\_delay}(1\to2) \\ &\quad + \text{processing\_delay}(2)) \\ &\quad + x_{s,1\to2\to3\to\text{target}} \cdot (\text{propagation\_delay}(2\to3) \\ &\quad + \text{processing\_delay}(3)) \end{aligned}$$

This optional optimization can only be used, if the routing optimization is also enabled.

6.2.6 Conflict Optimization

Our previous *Edge-Merger*-based approaches optimize variables and constraints for each stream individually. For our conflict constraints however, we always need to take two streams into account. Figure 6.6 shows, that two streams may share a common mergeable path (red edges). Our goal is to reduce the number of conflict variables and constraints in these cases. If we take a look at Figure 6.7, we can see, that there are exactly two cases on this path. In the first one, we schedule the smaller stream (green) first, and in the second one, we schedule the large stream (blue) first. This means, we only need one conflict variable for these streams on the red path, and thus two constraints. The base ILP, however, generates a variable and constraints for each edge on the red path, and thus 2 variables and 4 constraints. We aim to change this behavior, so that we only generate the minimum number of variables and constraints for a common merged path.

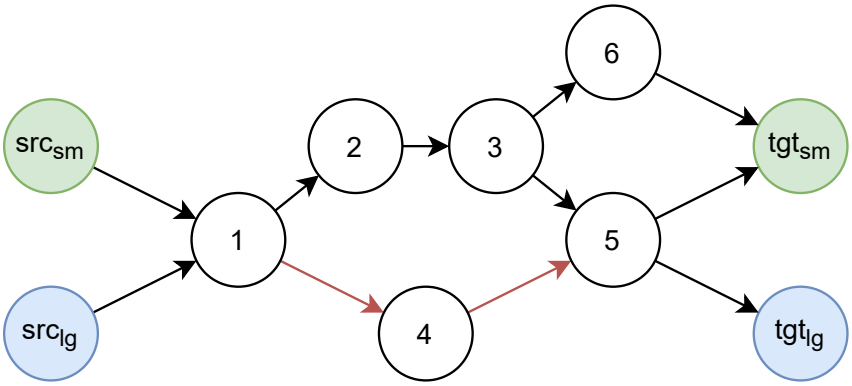
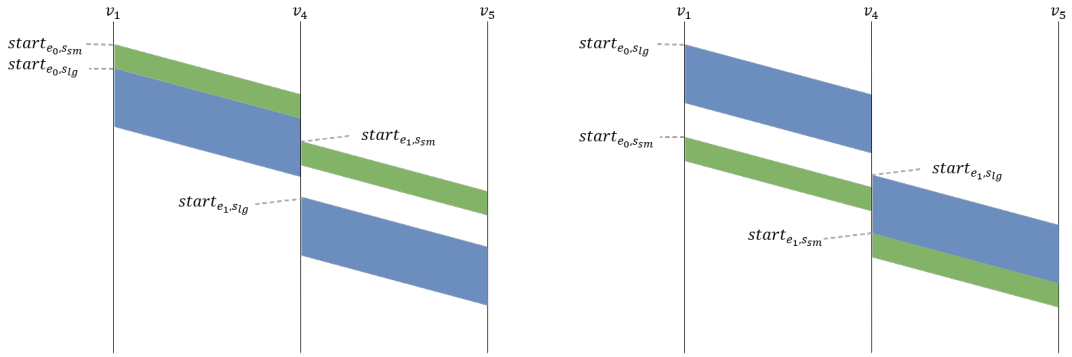


Figure 6.6: An example topology, where we are able to reduce the number of conflict constraint based on a common merged path, e.g. the red path.



(a) Scenario, which schedules the small stream first. **(b)** Scenario, which schedules the large stream first.

Figure 6.7: Two different possibilities of a no-wait schedule for streams of unequal size.

In the following, we present two different approaches for this optimization. But first, we need to take a closer look at the two different scenarios for streams of unequal size. We visualize our explanation using the red path of the example topology in Figure 6.6 and the two schedules in Figure 6.7. First, we differentiate the streams based on their stream size and refer to them as the small stream s_{sm} and the large stream s_{lg}. Figure 6.7a shows a scenario, where the small stream is scheduled first. The small stream always has a shorter transmission delay than the large stream. This means, that the gap between the small and the large stream increases after each switch. On the other hand this means, that if we schedule the large stream first (Figure 6.7b), the gap between those stream decreases. Both scenarios only apply for Store-and-Forward switching, as Cut-Through switching is independent of the transmission delay in our model. Using this knowledge, we are now able to present our two different conflict optimization approaches.

Conflict Optimization for Common Merged Edges

Our first optimization approach only works, if the merged edges of a stream are exactly the same. For our example topology in Figure 6.6, we are only able to optimize our conflict constraints for the merged edge 1→4→5, but not for 1→2→3, as s_{lg} has the merged edge 1→2→3→5.

To reduce the number of conflict constraints, we do not create them for each edge. Instead, we only create the conflict constraints once per common merged edge. For all other edges (those who are not part of a common merged edge), we create conflict constraints as before. Based on the previously described fact, that the gap between the schedule of two streams depends on the size of two streams, we generate our conflict constraints for common merged edges as follows: First, we identify, which is the small stream s_{sm} and which is the large stream s_{lg}. Secondly, we identify, which is the first edge e_{first} and which is the last edge e_{last} of the common merged edge e_{comm}. For our example (Figure 6.6) this means: $e_{\text{first}} = (1, 4)$ and $e_{\text{last}} = (4, 5)$. We now create the conflict variable $b_{s_{\text{sm}}, s_{\text{lg}}, e_{\text{comm}}}$. Based on this, we are able to create the conflict constraints similar to the Constraint (4.7). In contrast, we only create the first one (s_{sm} before s_{lg}) for e_{first} and vice versa for the second constraint (s_{lg} before s_{sm} on e_{last}) .

$$\begin{aligned} end_{s_{\text{sm}}, e_{\text{first}}} &\leq start_{s_{\text{lg}}, e_{\text{first}}} + (1 - b_{s_{\text{sm}}, s_{\text{lg}}, e_{\text{comm}}}) \cdot M \\ end_{s_{\text{lg}}, e_{\text{last}}} &\leq start_{s_{\text{sm}}, e_{\text{last}}} + b_{s_{\text{sm}}, s_{\text{lg}}, e_{\text{comm}}} \cdot M \end{aligned} \tag{6.4}$$

This optional optimization is only available, if the routing optimization is also enabled.

Conflict Optimization for Partly-Common Merged Edges

In some cases, merged edges are only partly-common, thus our previous optimization approach does not take them into account. For example, this is the case in Figure 6.6 for the path 1→2→3. This path is common to the merged edge 1→2→3 of s_{sm} but only partly-common to the merged edge 1→2→3→5 of s_{lg}.

We optimize these cases using the following steps: We first get all edges, on which s_{sm} and s_{lg} might conflict. For each edge e_{conf} of these edges, we get the merged edge for both streams, which contain this edge. In our example, e_{conf} might be $(1, 2)$, so we get 1→2→3 for s_{sm} and 1→2→3→5 for s_{lg}. We then search for the *Longest Common Subsequence (LCS)* of both edges. The LCS is 1→2→3 in our example. For all edges with a LCS, we then perform the same steps as in our previous approach for common merged edges. We do this only once per LCS by remembering for which LCSs we have already created the constraints. For all other edges, we perform the basic steps as in Constraint (4.7). Again, this optimization is only available, if the routing optimization is also enabled.

6.3 ILP Generation Optimizations

In the previous section, we already introduced different optimizations, which reduce the number of variables and constraints by reducing the underlying topology model. As already mentioned in Chapter 5, another possibility to reach our goals is to modify the ILP generation procedure itself. In this section, we first analyze our generation procedure and based on that, we provide two different optimization approaches. Our first finding is, that our ILP generates unnecessary auxiliary variables, which we may discard. Based on this finding, our first ILP generation optimization is the removal of end-variables from our ILP. The second finding is, that we are able to bound the variables of our ILP more tightly. Thus, we provide an optimization approach, which additionally bounds some variables of our ILP.

6.3.1 Remove End Variables

If we take a look at Constraint (4.5), we see, that all end variables of a stream solely depend on the start variable of the respective edge. In our current Store-And-Forward ILP, we still need them to correctly schedule streams on following edges (Constraint (4.6)) and to resolve conflicts between streams (Constraint (4.7)). However, we do not need the end variables to schedule streams in our Cut-Through ILP (Constraint (6.2)) at all. This finding allows us to eliminate end variables from our ILP, by replacing them with their respective definition of Constraint (4.5). Discarding end variables also allows us to discard the corresponding Constraint (4.5). We show the process of replacing the end variable exemplary for Constraint (4.6), but it works similar for all other constraints:

$$\begin{aligned}\sum_{e \in \text{out\_edges}(v)} start_{s,e} = &\sum_{e \in \text{in\_edges}(v)} end_{s,e} \\ &+ x_{s,e} \cdot (\,\text{propagation\_delay}(e) + \text{processing\_delay}(v)) \\ &+ x_{s,e} \cdot (\,\text{transmission\_delay}(s,e)) \\ = &\sum_{e \in \text{in\_edges}(v)} end_{s,e} \\ &+ x_{s,e} \cdot (\,\text{propagation\_delay}(e) + \text{processing\_delay}(v) \\ &\qquad + \text{transmission\_delay}(s,e))\end{aligned}$$

This optimization is independent of all other optimizations we introduced yet, and can be switched on and off as desired.

6.3.2 Variable Bounds

Our current ILP-model initially allows every scheduling variable to take any integer value. Only Constraint (4.4) and optionally Constraint (6.3) limit the maximum value to the cycle time or end-to-end limit respectively. In a valid schedule, however, the effective integer range is much smaller. As an example, the earliest departure of a stream is at $t_0 = 0\,\text{ns}$. Thus, a stream may never depart at the second edge at 0 ns. The same applies for the latest departure of a stream, as it may never exceed the cycle time. Thus, a stream must depart at the second last switch before the cycle time ends. This applies similar to all scheduling variables in our network. We use this knowledge to restrict our ILP-solver to a number range for each variable, and thus reduce our ILP's complexity by reducing the search-space.

In order to create these number ranges, we need to calculate a lower and an upper bound for each edge and stream in our network. Algorithm 6.2 shows our Dijkstra-based *Bound Calculator*. In this algorithm, we use the delays of edges and vertices as weights. This means, the "distance" between two vertices matches the delays between them, including the processing delay of the target vertex. We now execute the Dijkstra algorithm to calculate the length of the shortest path to each vertex using the weights. The length of the shortest path from a stream's source to a vertex now equals the earliest departure time on that vertex. This value directly corresponds to the earliest departure on each outgoing edge of that vertex, and thus to the lower bound of its start variable.

Algorithm 6.2 Bound Calculator

```
1:  function boundCalculator(stream: Stream)
2:      g ← new Graph(stream.vertices, stream.edges)
3:      weights: Dict[Edge, Number]
4:      for each edge in g.edges do
5:          weights[edge] ← generateWeigth(stream, edge)
6:      end for
7:      lowerDist, _ ← dijkstra(g, stream.source, weights)
8:      g ← g.reverse()
9:      upperDist, _ ← dijkstra(g, stream.target, weights)
10:
11:     lowerBounds: Dict[Edge, Number]
12:     upperBounds: Dict[Edge, Number]
13:     for each edge in g.edges do
14:         lowerBounds[edge] ← lowerDist[edge.source]
15:         tempUpper ← stream.cycleTime
16:         tempUpper ← tempUpper − upperDist[edge.source]
17:         tempUpper ← tempUpper + processingDelay(edge.source)
18:         upperBounds[edge] ← tempUpper
19:     end forreturn lowerBounds, upperBounds
20: end function
21:
22: function generateWeight(stream: Stream, edge: Edge)
23:     delaySum: Number ← 0
24:     delaySum ← delaySum + transmissionDelay(stream, edge)
25:     delaySum ← delaySum + propagationDelay(edge)
26:     delaySum ← delaySum + processingDelay(edge.target)
27:     return delaySum
28: end function
```

We use a similar approach to calculate the latest departure on edges. First, we invert all edges in the graph, but keep the weights of them. As in the first approach, we now execute the Dijkstra algorithm, but starting from the target vertex of the stream. We cannot directly use this value as an upper bound for the edges. Instead, we need to subtract it from the cycle time in order to gain the upper bound. But as we did not change our weights, we now have to re-add the processing delay for the source node, as we have also subtracted it from the cycle time.

For each start variable, we can directly set the lower and upper bounds based on the calculation of our *Bound Calculator*. We do not increase the lower bound of end variables, as e.g. Constraint (4.5) requires, that they are equal, if an edge is unused. Still, we have to increase the upper bound of end variables by the transmission delay.

Creating a lower bound for scheduling variables creates another issue. Our basic ILP-model relies on scheduling variables being zero if they are unused. For example, Constraint (4.4) sets the end variable of an Edge e to zero if it is unused. This applies for nearly all scheduling and conflict constraints, e.g. Constraint (4.6) sums up all start variables with the assumption, that only one of these variables holds a non-zero value. Thus, we need to introduce some changes to our ILP.

First, we need to ensure, variables of unused edges always contain the lower bound of this variable. Thus, we replace Constraint (4.4):

$$\forall s \in \mathcal{S}, \forall e \in \mathcal{E} : end_{s,e} \leq start_{s,e} + x_{s,e} \cdot \text{transmission\_delay}(s,e) + \text{lower\_bound}(s,e) \quad (6.5)$$

Second, we need to ensure we correctly calculate the start time for subsequent edges. Thus, we have to subtract the lower bound for each unused edge on Constraint (4.6) with Constraint (6.3.2) This works similar for the end to end Constraint (6.3)

$$\forall s = (v_{\text{src}}, v_{\text{dst}}, \_) \in \mathcal{S}, \forall v \in \mathcal{V} \setminus \{v_{\text{src}}, v_{\text{dst}}\} :$$

$$\begin{aligned} &\sum_{e \in \text{out\_edges}(v)} start_{s,e} - \sum_{e \in \text{out\_edges}(v)} (1 - x_{s,e}) \cdot \text{lower\_bound}(s,e) \\ = &\sum_{e \in \text{in\_edges}(v)} end_{s,e} + x_{s,e} \cdot (\text{propagation\_delay}(e) + \text{processing\_delay}(v)) \\ &- \sum_{e \in \text{in\_edges}(v)} (1 - x_{s,e}) \cdot \text{lower\_bound}(s,e) \end{aligned} \quad (6.6)$$

Lastly, we have to fix the conflict constraints. We show this exemplary for Constraint (4.7), but it works equivalent for Constraint (6.4):

$$\begin{aligned} &end_{s,e} - (1 - x_{s,e}) \cdot \text{lower\_bound}(s,e) \\ \leq &start_{s',e} - (1 - x_{s',e}) \cdot \text{lower\_bound}(s',e) + (1 - b_{s,s',e}) \cdot M \\ &end_{s',e} - (1 - x_{s',e}) \cdot \text{lower\_bound}(s',e) \\ \leq &start_{s,e} - (1 - x_{s,e}) \cdot \text{lower\_bound}(s,e) + (b_{s,s',e}) \cdot M \end{aligned} \quad (6.7)$$

This optimization is fully compatible to all other optimizations we explained above. It can also be used independently to all other optimizations.

6.4 Gurobi Optimizations

While all previous optimizations aim to improve the runtime by modifying the ILP itself, it may also be possible to improve the runtime by using ILP-solver specific functionality. As we only use the Gurobi ILP-solver, we make use of Gurobi-specific modifications and additions in this section. Our first optimization approach makes use of Gurobi's indicator constraints. After that, we analyze the influence of our different ILP components (routing, scheduling and conflict). We use our findings to provide some constraints with Gurobi's *lazy*-attribute. Gurobi also enables us to specify hints for a possible solution to reduce the runtime of the solver. We use this opportunity to introduce an approach which creates hints for routing variables and based on that also for scheduling and conflict variables. Lastly, we notice, that it is possible to modify Gurobi's solving process by tweaking different parameters of Gurobi.

6.4.1 Indicator Constraints

As an alternative to big M constraints, Gurobi provides so-called *Indicator Constraints*. They allow us to provide a binary indicator variable. Only if this variable holds the value 1, Gurobi checks the following constraint. We aim to reduce the runtime of our solver by replacing our current conflict constraints with new indicator constraints.

We notate indicator constraints as follows: i $\rightarrow$ c. This notation reads as: "If indicator i is 1, the following constraint c needs to be satisfied." Based on this notation, we provide our conflict constraints as indicator constraints:

$$\begin{aligned} b_{s,s',e} &\rightarrow end_{s,e} \leq start_{s',e} \\ (1 - b_{s,s',e}) &\rightarrow end_{s',e} \leq start_{s,e} \end{aligned} \tag{6.8}$$

This optimization is optional and fully compatible to all other previous optimizations. In order to use it without end variables or with bounds, we perform the same steps as provided in the respective sections.

6.4.2 Lazy Constraints

There are scenarios, in which conflicts on edges are unlikely to occur. For example, this is the case in networks with a low link utilization or when edges are part of an unlikely path of a stream. In these cases, the ILP-solver often has to unnecessarily verify, whether the conflict constraints are satisfied. To prevent this, Gurobi allows us to use so-called *Lazy Constraints*. When our model contains lazy constraints, the ILP-solver first tries to find a feasible solution without considering any lazy constraints. If the solver finds a solution, it checks if the solution also satisfies all lazy constraints. If not, it converts all unsatisfied lazy constraints to normal constraints and repeats this process.

As earlier mentioned, there are cases, in which conflict constraints are likely to be satisfied by default. Thus, we aim to gain a performance boost by converting all of our conflict constraints to lazy constraints. This works by setting a flag on each conflict constraint which marks it as lazy.

This optimization is not compatible with our indicator constraint optimization. Apart from that, it works with all other previous optimizations and it is optional.

6.4.3 Variable Hints

As mentioned earlier, the scheduling problem is an NP-hard problem. However, it is easier to find an "almost working" schedule. If we have such an almost working solution, Gurobi enables us to provide them as hints for our variables, and thus reduce the runtime. We provide two optimizations build upon each other, which make use of variable hints. Our first optimization provides hints for routing constraints. The second optimization uses them for scheduling and conflict constraints based on the route hints.

Route Hints

All streamsets we use already contain a possible route from the stream's source to the stream's target. Otherwise, calculating a path or the shortest path between the stream's source and target is also efficiently possible by using the DFS or Dijkstra algorithm (Sections 2.4.2 and 2.4.3). We can simply use the given or calculated path and convert it to a hint for our routing variables. First, we set the hint of all routing variables $x_{s,e}$ to 0. We then walk along the path from the source to the target and set the hint for all routing variables on this path to 1.

This optimization is compatible with all previous optimizations. It may be be used independently of all other optimizations and is optional.

Schedule & Conflict Hints

In our previous optimization, we create hints for the routing variables of our ILP. Similarly, we can also create hints for scheduling variables, and hence also for conflict variables. We create those schedules for the given or calculated route of our previous approach. As we aim to keep the runtime of all preprocessing steps low, our *Trivial Scheduler* (Algorithm 6.3) does not optimize schedules in any form, but schedules them stream after stream. Especially, our *Trivial Scheduler* does not guarantee, that all streams are scheduled within the cycle time.

Our *Trivial Scheduler* works as follows: First, we sort our streamset by the size of the streams in ascending order. We do this in order to schedule as many streams as possible within the cycle time. After that, we iterate over all streams and schedule them one after the other. Per stream, we first generate a schedule which starts at time 0 on the first edge. This is trivial, as we know all delays and can simply sum those up along the path. In the next step, we push this schedule to the back until it is behind all previously scheduled streams on each utilized edge. For this, we need to get the latest utilization on each edge along the stream's route and then move our schedule, so that it directly starts after this utilization.

With the created schedule, we are able to set the variable hints for scheduling variables. For all streams within the provided cycle time, we directly use the generated schedule as a hint value for the scheduling variables. In order to generate hints for the conflict variable, we check which conflicting stream is scheduled first. As previously mentioned, there may be streams, which utilize edges after the end of a cycle. We aim to minimize this number by sorting the streams before scheduling them. For all streams with an invalid schedule, we do not set any scheduling hints. As our *Trivial Scheduler* mostly generates a lot of spacing between streams, we expect Gurobi to schedule the remaining streams between those streams with a correct scheduling hint.

This optimization depends on routing hints, and thus is only available if routing hints are also enabled. Furthermore, this optimization is incompatible with the End-to-End constraint of Section 6.1.3, Apart from that, it is optional and compatible with all other optimizations.

Algorithm 6.3 Trivial Scheduler

```
1:  type Schedule = Dict[Edge, Tuple(Number, Number)]
2:
3:  function trivialScheduler(streams: List[Stream])
4:      schedules: Dict[Stream, Schedule]
5:      streams ← streams.sortBySize()
6:      for each stream in streams do
7:          schedule ← createScheduleAtZero(stream)
8:          for each edge, (start, end) in schedule do
9:              last ← getLastUtilizationOnEdge(schedules, edge)
10:             if last < start then
11:                 schedule ← moveWholeSchedule(schedule, last − end)
12:             end if
13:         end for
14:         schedules[stream] ← schedule
15:         return schedules
16:     end for
17: end function
18:
19: function createScheduleAtZero(stream: Stream)
20:     // This function creates a schedule starting at time 0 on the edges of stream.route.
21:     // This is trivial by summing up the delays edge by edge.
22:     return schedule
23: end function
```

6.4.4 Parameter Tuning

All previous optimizations aim to reduce the number of variables and constraints or providing them with additional attributes. However, Gurobi allows to modify the solving process by adjusting various parameters. In our last optimization approach, we aim to tweak these Gurobi parameters in order to achieve a shorter runtime.

Gurobi provides about 50 different parameters for MIP and ILP, which we may change in order to influence its behavior and thus its runtime. As an example, we are able to define the number of *Presolve Passes* but also more sophisticated parameters, such as the *BranchDir* parameter, which allows us to specify the direction of the internal *branch-and-cut* search. Due to the large amount of available parameters, we are not able to test all different parameters, especially not all possible combinations, on a large number of topologies and streamsets. Instead, we use another approach based on Gurobi's *Parameter Tuning Tool (PTT)*. The PTT takes an ILP-model and aims to find sets of parameters, which result in a lower runtime than the default parameter set. It then outputs those parameter sets. As the structure of our ILP-model is the same for all topologies and streamsets, we assume there may be parameters which result in a lower runtime in the majority of our models. Thus, we run the PTT for different topologies and streamsets in order to find these parameters, if they exist. We present these results in Chapter 7.

7 Evaluation

In Chapter 6, we introduced different optimization approaches for our ILP-model. So far we do not know which of these ILP adjustments actually provide a runtime boost. In this chapter, we first evaluate our different optimization approaches individually and combined for Store-and-Forward and Cut-Through switching. Based on the results, we select the best optimization combination for both switching types. We then use our selected optimization-set to benchmark our optimized ILP-approach against other approaches of Chapter 3. In both sections, we first present our evaluation setup followed by the evaluation results.

7.1 Optimization Evaluation

Before we are able to compare our ILP-based JRaS approach to other schedulers, we first need to compare our own optimization approaches and combinations of them. In this section, we first portray our evaluation setup. This includes the evaluation hardware as well as the topology and streamset generation. After that, we first evaluate individual optimization approaches, which we then combine to select the best combination. We also make use of Gurobi's Parameter Tuning Tool (PTT) to find a parameter combination, that reduces the runtime for the majority of our test cases.

7.1.1 Evaluation Setup

This section contains the setup we use for evaluating our different optimization approaches. We first introduce the hardware on which we run our test cases. After that, we explain our evaluation procedure, which contains our approach to generate topologies and streamsets and the Gurobi execution itself. Lastly, we present abbreviations for our different optimizations to distinguish between them in our evaluation.

Hardware

For our optimization evaluation we use a cluster of identical nodes. Table 7.1 shows the hardware of each node in our cluster. We only run one Gurobi instance per node at a time. This instance may utilize the all server resources, but to reduce the impact of OS-specific processes, we limit the number threads to 11 instead of 12.

OS	CentOS Linux 7 (Core)
Kernel	Linux 5.7.0-1.el7.elrepo.x86_64
CPU	Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz (6 Cores, 12 Threads)
Memory	4×4 GB RDIMM DDR4 Synchronous 2,133 MHz

Table 7.1: Hardware for parameter evaluation.

Evaluation Procedure

To evaluate our different optimization approaches, we first need to generate topologies and streamsets. As we developed some of our optimizations with a factory-backbone topology in mind, we generate random factory-backbone topologies with a random number of vertices between 100 and 400. All switches have a constant and equal processing delay of 2,000 ns and all links have a constant and equal propagation delay of 200 ns and link speed of 1 $\frac{\text{Gbit}}{\text{s}}$. For each topology, we create a streamset with a random number of streams between 40 and 150. Each stream has a random start and end vertex, while both of them are hosts. Further, all streams use a cycle time of 1,000,000 ns and a random payload size between 64 B and 300 B. As both, topologies and streamsets are random, we cannot guarantee there is a feasible solution, even though this is unlikely due to the high cycle time.

After generating topologies and streamsets, we are able to execute our scheduler on the created test cases. To get comparable results within a reasonable amount of time, we always limit the runtime of Gurobi to 900 s per test case. If Gurobi hits this time limit, we mark the execution as *unsuccessful*, which means, the tested approach did not find a solution withing the given time limit. Depending on the presentation of our results, we either show unsuccessful runs separately or treat them like a 900 s execution.

Naming and Abbreviations of Optimizations

In Chapter 6 we introduced different optimizations, which we may use in any combination fulfilling the given dependencies. The following Section 7.1.2 contains results using different combinations of these results. In order to distinguish between different optimization sets without using their full names, we introduce an optimization abbreviation dictionary in Table 7.2. For example, *ilp-cut-red+conf+adv-bounds* means we use a Cut-Through ILP-model with extra bounds and with merged edges for routing and conflict constraints (advanced method).

In some cases, we may use even shorter abbreviations. *-red+all* instead of *+red+sched+conf+adv* and *-allhint* instead of *-routehint+schedulehint*

7.1.2 Evaluation Results

The following sections contain the results of our optimization evaluation. We first present results for the evaluation of single optimizations and combinations for optimizations, which cannot be used individually. After that, we analyze the influence of different Gurobi parameters based on

Abbreviation		Description
ilp		We always use this prefix to indicate we use our ILP-model.
-cut		We use our Cut-Through model. If this parameter is missing, we use a Store-and-Forward model.
-e2e		We limit streams to their maximum end-to-end delay.
-red		We use reduce the topology size, by merging edges and use this reduction for the routing part of the ILP.
	+sched	We also use merged-edges for the scheduling part.
	+conf	We also use merged-edges for the conflict part (basic method).
	+adv	We use the advanced method for the conflict part for partly-common edges. (Requires +conf).
-noend		We remove end variables from our ILP.
-bounds		We add extra bounds to our scheduling variables.
-ind		We use indicator constraints instead of big-M constraints.
-lazy		We use lazy conflict constraints.
-routehint		We add hints for routing variables.
	+ schedulehint	We also add hints for scheduling and conflict variables.

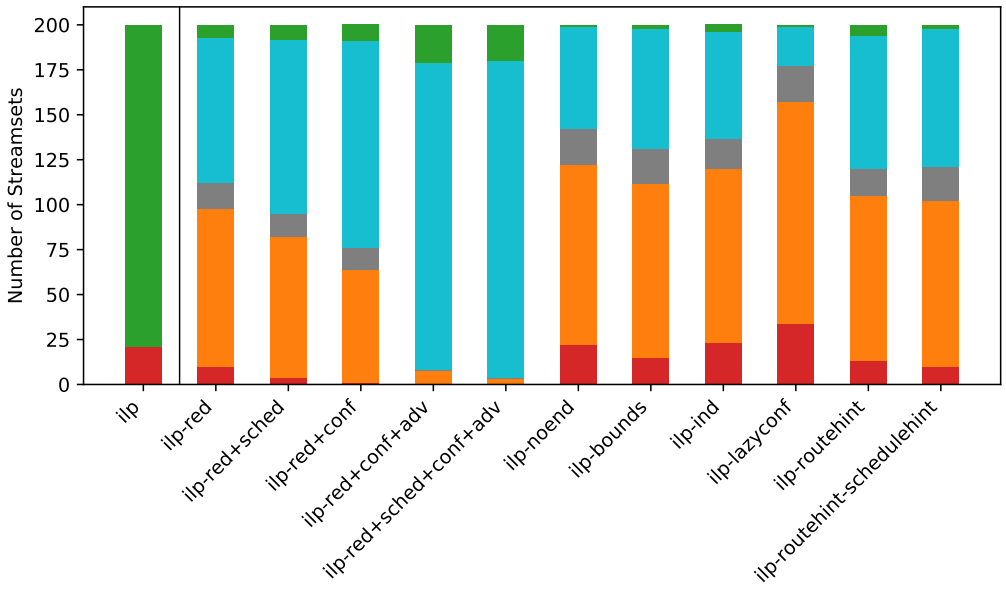
Table 7.2: Abbreviation for optimization combinations.

Gurobi's Parameter Tuning Tool (PTT). Based on these results, we evaluate combinations of different optimizations. Finally, we select the best optimization combination for Store-and-Forward and Cut-Through.

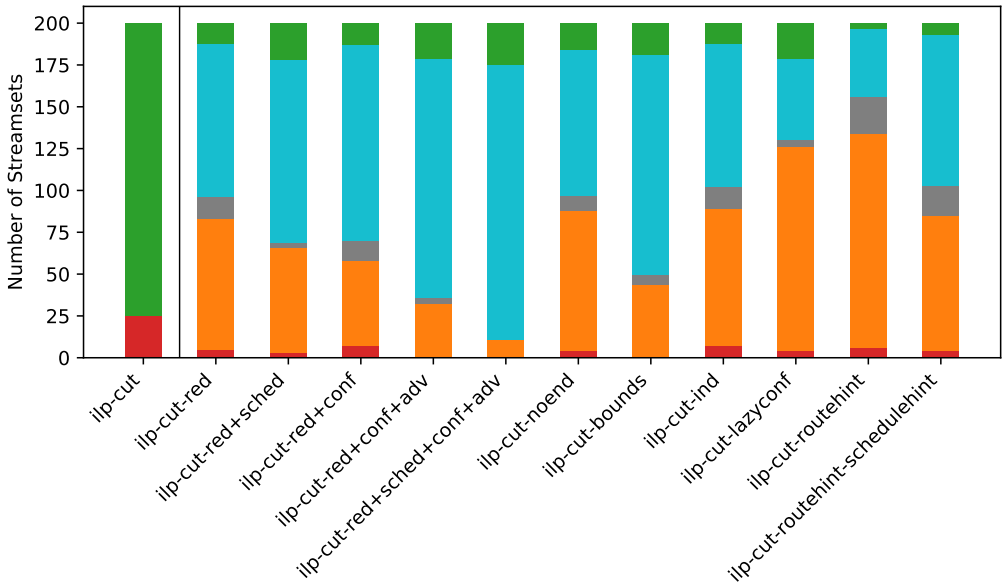
Single Optimization Evaluation

Our first evaluation results contain a comparison between the base model and single optimizations for Store-and-Forward and Cut-Through respectively. This evaluation also contains optimizations, which depend on other optimizations. We first compare the runtime of optimizations for each single stream to compare them to our base model. After that, we present the runtime distribution of different approaches.

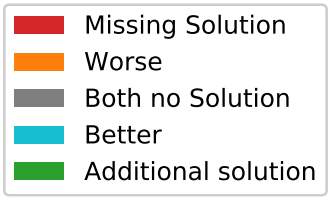
Figure 7.1 shows the runtime comparison between the optimization approaches and the base model for Store-and-Forward respectively. As we can see, our topology optimizations (*ilp-red*+*) perform better in nearly all cases for both, Store-and-Forward and Cut-Through switching. Some optimizations, when using them individually, do not decrease the runtime of the ILP-solver in general. For example, the removal of end variables and our additional bounds performs slightly worse for our Store-and-Forward model when compared to the base ILP. However, this does not apply for Cut-Through switching. In our Cut-Through model, we can see a slight performance boost when removing end variables and adding bounds. We assume the different scheduling constraints



(a) Comparison for Store-and-Forward.

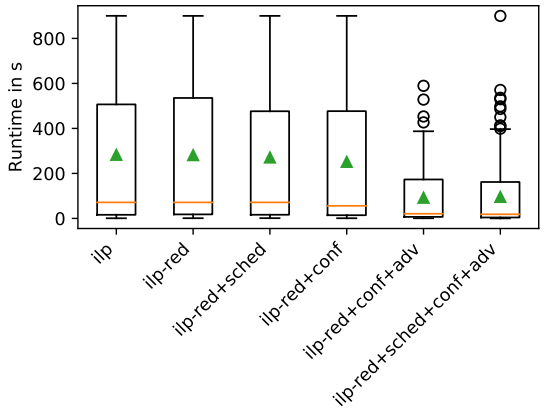


(b) Comparison for Cut-Through.

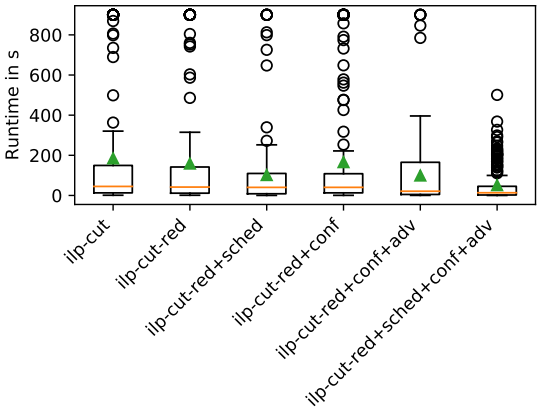


(c) Graph legend. Optimization approaches always compares to the respective base model (first bar).

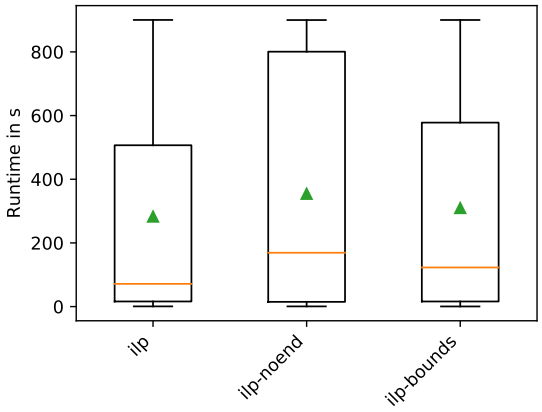
Figure 7.1: Solution count and runtime comparison between optimization approaches and the base model for Store-and-Forward and Cut-Through respectively. For the base model, we show the number of solved and unsolved streamsets.



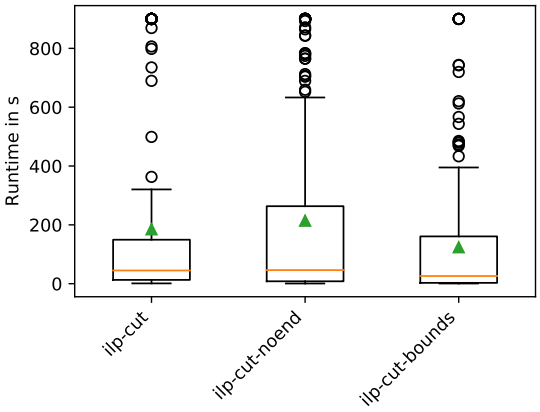
(a) Runtime comparison for Store-and-Forward topology optimizations.



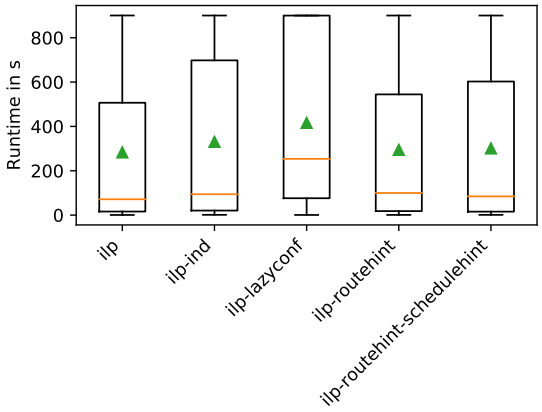
(b) Runtime comparison for Cut-Through topology optimizations.



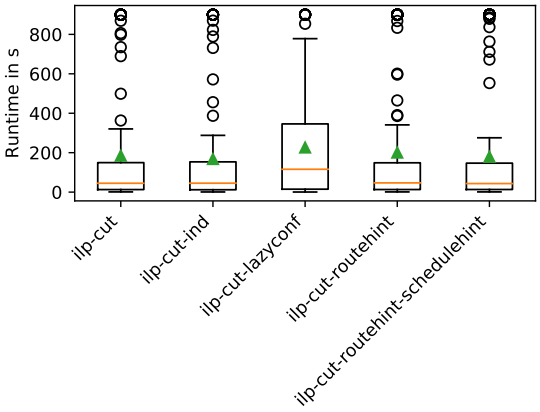
(c) Runtime comparison for Store-and-Forward ILP optimizations.



(d) Runtime comparison for Cut-Through ILP optimizations.



(e) Runtime comparison for Store-and-Forward Gurobi optimizations.



(f) Runtime comparison for Cut-Through Gurobi optimizations.

Figure 7.2: Runtime comparison of different optimization approaches for Store-and-Forward (left) and Cut-Through (right).

cause this difference. In Store-and-Forward switching, we need the end variables to schedule streams on the next edges in Constraint (4.6). This is different in our Cut-Through model. We do not need the end variables to schedule our streams in Constraint (6.2). So, in Store-and-Forward switching, removing end variables increases the complexity of the scheduling constraints. In our Cut-Through model however, the complexity does not increase. We also find out, that using lazy constraints increases the runtime in most of the cases. This most likely comes from the fact, that the ILP-solver needs to regenerate solutions if the initial solution violates some conflict constraints.

While Figure 7.1 might suggest, our routehint optimization approach performs worse than the original ILP, Figure 7.2 shows, that the runtime is comparable between both approaches. This is explainable by the fact, that in some cases our route hint is correct and leads to a remarkably improved runtime, but in many other cases our hint might be wrong and the runtime is slightly higher. On average, indicator constraints perform almost the same as our default ILP-model. This most likely comes from the fact, that Gurobi maps indicator constraints to normal constraints [Gur] and thus, they perform similar.

Overall we can say, that our topology optimizations result in the best improvements. For Store-and-Forward, most other improvements result in a similar or slightly worse runtime when used individually. For Cut-Through, removing end variables and adding bounds improves the runtime a bit further.

Parameter Tuning

Our next step is to analyze the influence of different Gurobi parameters on the runtime of different test cases. We aim to find parameters, which decrease the runtime in nearly all cases. In order to do this, we run Gurobi's PTT on another testset with 200 random streamsets and topologies using our best evaluated Store-and-Forward optimization combination. We then count how many times Gurobi sets specific values for these parameters. Figure 7.3 shows these results.

We can see, there is only one parameter which tunes more than half of our test cases. All other parameters do not qualify as a general improvement for our ILP. Still, the most used parameter (*MIPFocus*) uses three different values, thus we cannot simply set a specific value for this parameter to optimize the runtime for the majority of our streamsets. Thus, we aim to find a correlation between the value of the *MIPFocus* parameter and the topology size and number of streams. For this, we generate another testset with 500 different topologies and streamsets. We calculate the runtime for each possible $\mathit{MIPFocus} \in \{\text{None}, 1, 2, 3\}$.

Figure 7.4 shows the results of our *MIPFocus* parameter evaluation. A colored dot represents a specific test case and the color corresponds to the *MIPFocus* value with the shortest runtime for this test case. Using this approach, we are not able to cluster specific colors using the topology size and/or the number of streams. Thus, we could not find a correlation between the number of streams and nodes and a specific *MIPFocus* value. In general, we are not able to find a specific parameter set that improves the runtime for a majority of the test cases.

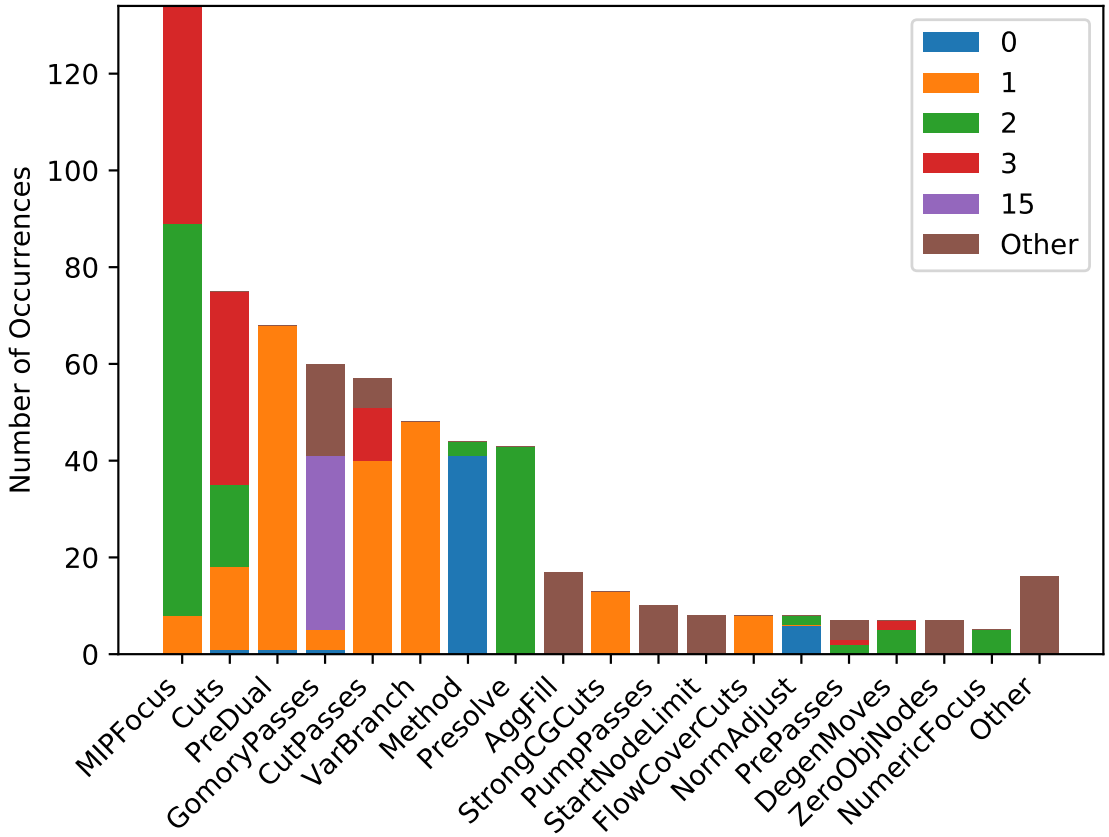


Figure 7.3: Number of usages per parameter and value by the PTT.

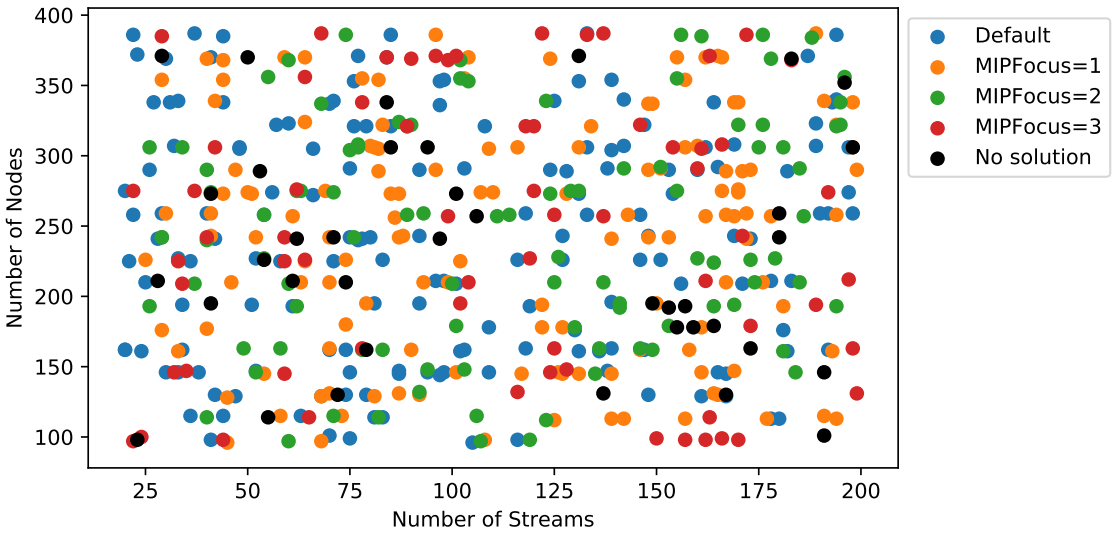


Figure 7.4: *MIPFocus* runtime evaluation. Each dot represents a testcase and its color the MIPFocus with the lowest runtime.

Combined Optimization Eval

In the last sections, we already presented results for single optimization approaches and the influence of different Gurobi parameters. However, we did not yet combine different kind of optimizations to further reduce the runtime. Due to the large amount of possible optimization combinations, we are unable to evaluate all of them on a large testset. Thus, create promising combinations to find the best-possible optimization set. In order to do this, we use the results from the last sections. We generate a lager number of combinations with optimizations, which perform good individually (e.g. our topology optimizations) and a low number of combinations with less promising ones (e.g. lazy constraints). As we were not able to find Gurobi parameters with a general improvement, we only evaluate combinations using our own optimization approaches. We run these combinations on the same testset as our single optimization evaluation. This allows us to compare the results to our original ILP and to single optimizations.

Our total evaluation contains about 35 different combinations, thus we are unable to show results for all of them. Instead, we only present the best combinations. This includes combinations, which provided solutions in all test cases and combinations with the lowest average runtime. As our best combinations differ between Store-and-Forward and Cut-Through, we do not present all combinations for both approaches.

In Figure 7.5, we can see, that nearly all of our best combinations perform better than our best single optimization approach (including those, who depend on other optimizations). Figure 7.6 confirms this, as the average runtime as well as upper quartile is lower in most cases. To find the best approach out of these combinations, we need to take a more detailed look at the average runtimes in the next section.

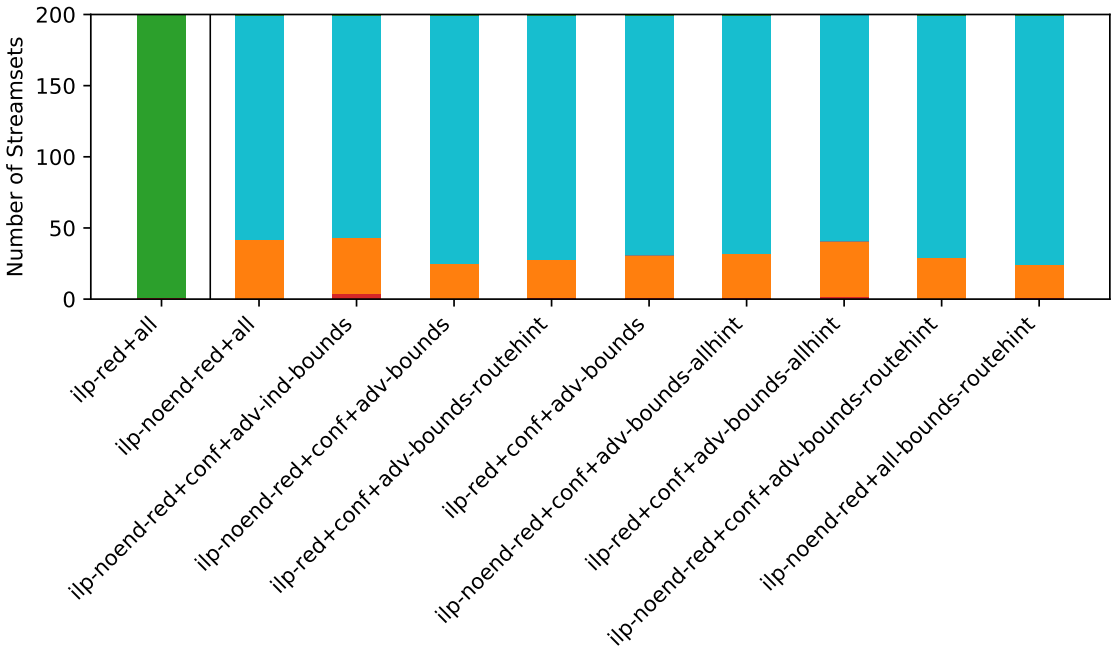
Best Optimization Combination

Based on our previous results, we are now able to select the best parameter combination for Store-and-Forward and Cut-Through switching respectively. For this, we use two different metrics:

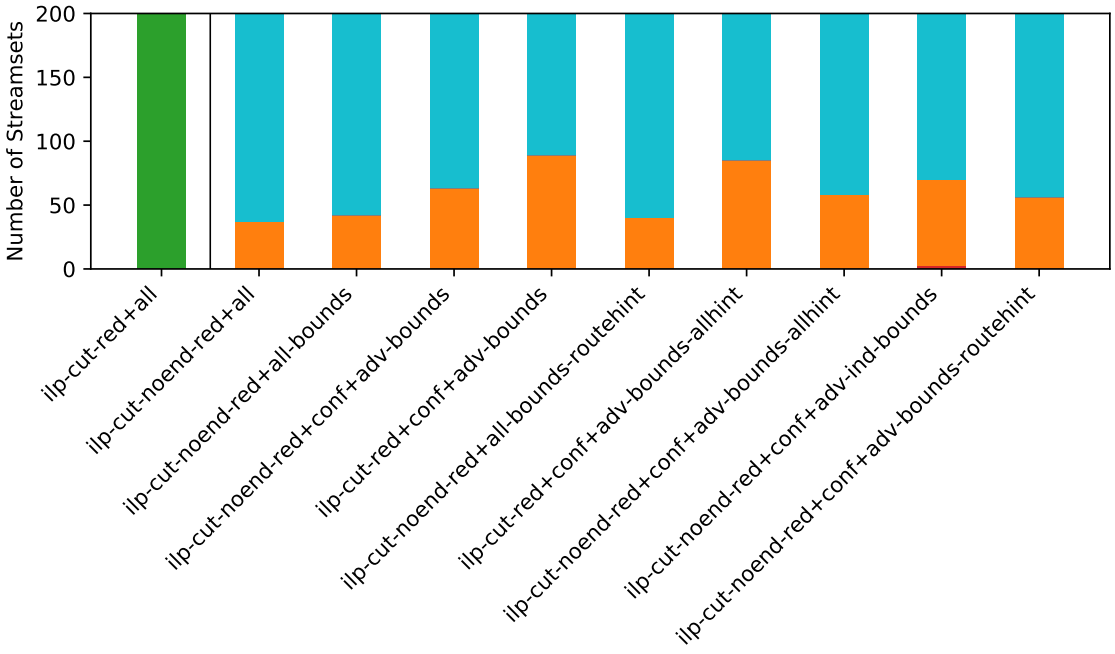
1. Number of solved ILPs: For each optimization combination analyzed in the sections above, we calculate the percentage of solved ILPs out of our total number of 200.
2. Average runtime: We calculate the average runtime for each optimization combination. This only includes test cases, which were solved within the runtime limit.

In Table 7.3 we present an overview of the best 15 optimization combinations sorted by their runtime for Store-and-Forward and Cut-Through respectively. For Cut-Through switching, the best combination is clearly *ilp-cut-noend-red+conf+adv-bounds-routehint*. However, for Store-and-Forward switching, we have to choose between *ilp-red+conf+adv-bounds-routehint*, which has the lowest average runtime and *ilp-noend-red+conf+adv-bounds*, which has a slightly higher runtime but solves 100 % of our testset. In most cases, their performance is similar, so we could use both of them. For all following evaluations we use the first one (*ilp-red+conf+adv-bounds-routehint*).

Now, that we have selected our best optimization combinations for both, Store-and-Forward and Cut-Through switching respectively, we check whether these approaches meet our goals from Chapter 5. Figure 7.7 shows, that our optimized approach is able to schedule more test cases than the base

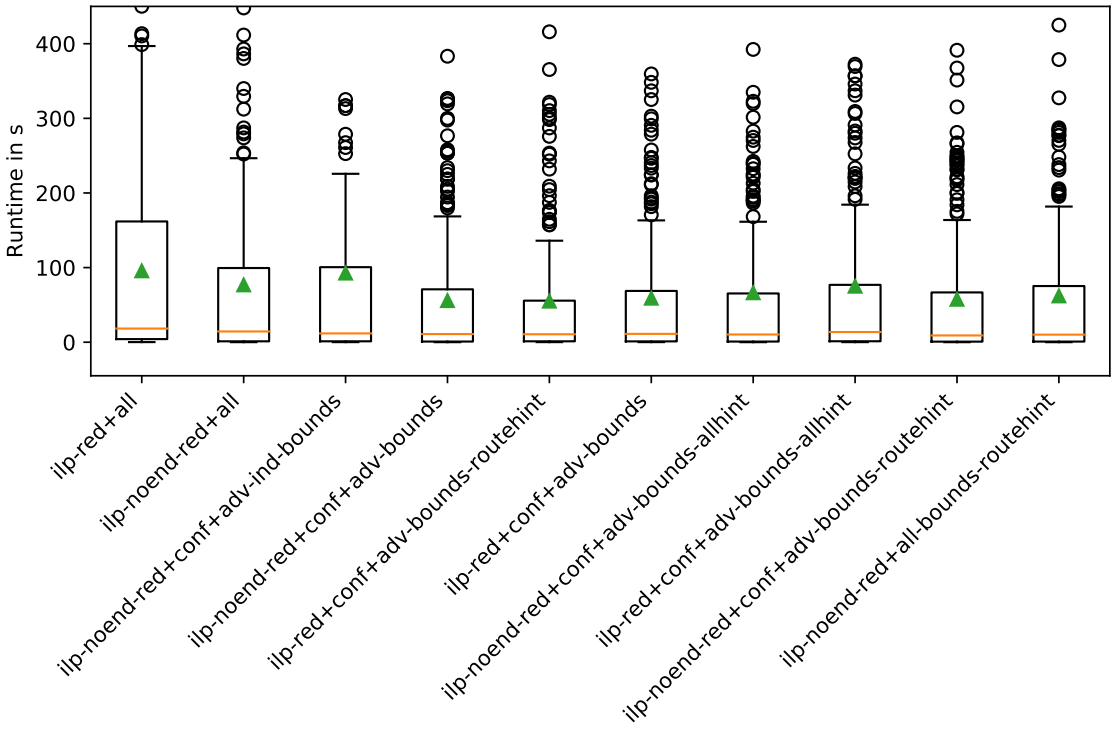


(a) Comparison for Store-and-Forward.

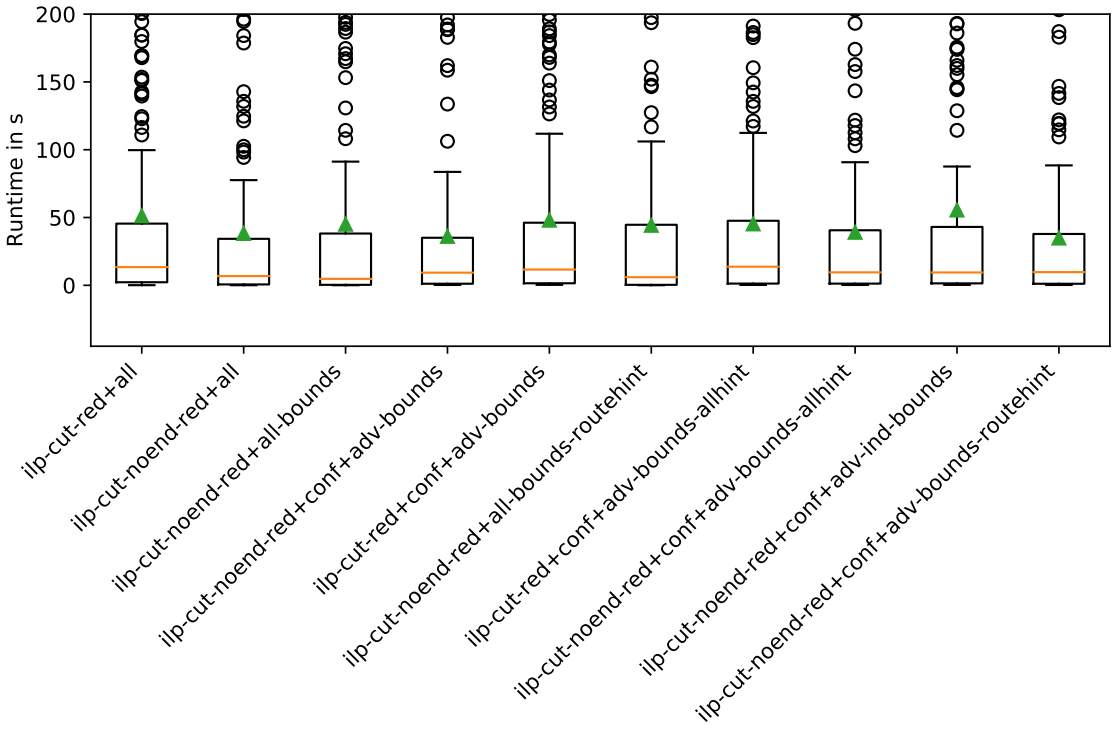


(b) Comparison for Cut-Through.

Figure 7.5: Direct comparison between the best result of the single optimization evaluation and the best combinations (Legend in Figure 7.1c).



(a) Comparison for Store-and-Forward.



(b) Comparison for Cut-Through.

Figure 7.6: Runtime comparison between the best combined optimization approaches. There are more outliers above the respective runtime border, we cut them off to allow a more detailed comparison between the average runtimes.

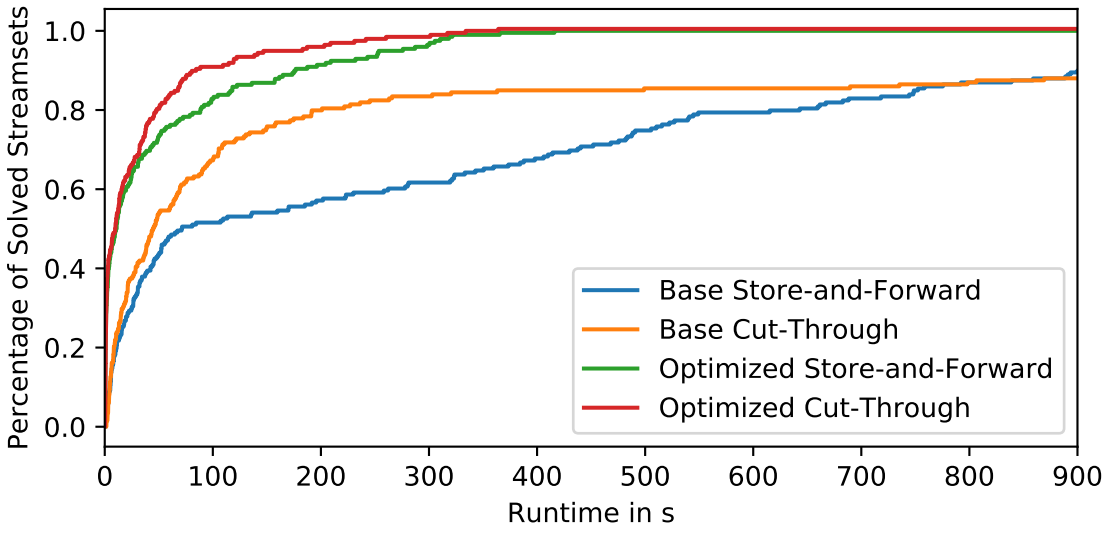


Figure 7.7: Scheduling Capability and runtimes of the base ILP-models and our optimized ILP-models using the best combinations.

ILP-model within the time limit of 900 s per test case. We can also see that it is able to solve these test cases within a shorter runtime. Thus, our best optimization combination for Store-and-Forward as well as for Cut-Through is able to improve the base ILP-model.

If we count executions reaching the runtime limit as executions with a runtime of 900 s, our base models for Store-and-Forward and Cut-Through have an average runtime of about 283 s and 185 s. For our best optimization combinations, the average runtimes are 55 s and 35 s respectively. This means, our optimized approaches improve the runtime by about 80.6 % for Store-and-Forward and 81.1 % for Cut-Through switching.

Selected Optimizations	Solved	Avg. Runtime
ilp-red+conf+adv-bounds-routehint	99.50 %	50.75 s
ilp-red+conf+adv-bounds	99.50 %	54.74 s
ilp-noend-red+conf+adv-bounds	100.00 %	55.72 s
ilp-noend-red+conf+adv-bounds-routehint	100.00 %	57.50 s
ilp-noend-red+sched+conf+adv-bounds-routehint	99.50 %	57.79 s
ilp-noend-red+conf+adv-bounds-routehint-schedulehint	99.50 %	62.02 s
ilp-red+conf+adv-bounds-routehint-schedulehint	98.50 %	62.46 s
ilp-noend-red+conf+adv-ind-bounds	98.00 %	76.10 s
ilp-noend-red+sched+conf+adv	100.00 %	76.78 s
ilp-noend-red+conf+adv	99.50 %	81.50 s
ilp-noend-red+sched+conf+adv-bounds	100.00 %	84.70 s
ilp-red+sched+conf+adv	99.50 %	91.55 s
ilp-red+conf+adv	100.00 %	92.00 s
ilp-red+conf+adv-bounds-lazyconf	100.00 %	99.71 s
ilp-red+sched+conf+adv-bounds	98.50 %	103.02 s

(a) Results for Store-and-Forward.

Selected Optimizations	Solved	Avg. Runtime
ilp-cut-noend-red+conf+adv-bounds-routehint	100.00 %	34.79 s
ilp-cut-noend-red+conf+adv-bounds	100.00 %	36.03 s
ilp-cut-noend-red+sched+conf+adv	100.00 %	38.18 s
ilp-cut-noend-red+conf+adv-bounds-routehint-schedulehint	100.00 %	39.04 s
ilp-cut-noend-red+sched+conf+adv-bounds-routehint	100.00 %	44.20 s
ilp-cut-noend-red+sched+conf+adv-bounds	100.00 %	44.99 s
ilp-cut-red+conf+adv-bounds-routehint-schedulehint	100.00 %	45.23 s
ilp-cut-noend-red+conf+adv-ind-bounds	99.00 %	46.95 s
ilp-cut-red+conf+adv-bounds	100.00 %	47.97 s
ilp-cut-red+conf+adv-bounds-routehint	100.00 %	48.67 s
ilp-cut-red+sched+conf+adv	100.00 %	51.21 s
ilp-cut-red+conf+adv-bounds-lazyconf	100.00 %	69.42 s
ilp-cut-red+sched+conf+adv-bounds	100.00 %	69.44 s
ilp-cut-noend-red+conf+adv	100.00 %	75.94 s
ilp-cut-noend-bounds	99.50 %	76.12 s

(b) Results for Cut-Through.

Table 7.3: Average runtime comparison of the best evaluated combinations. Only contains the best 15 combinations.

7.2 Benchmarking

In order to identify the capabilities of our optimized ILP-based scheduler, we compare it to other schedulers from Chapter 3. For this, we use a benchmarking framework by Schneefuss et al. [SWHD20]. This benchmarking framework automates the topology and streamset generation and splits the scheduler execution between different nodes. In this section, we first present the benchmarking setup, including the benchmarking hardware and a detailed description of the benchmarking framework. Thereafter, we present the benchmarking results in the second section.

7.2.1 Benchmarking Setup

In a first step of the benchmarking section, we explain our evaluation setup. We start with an explanation of Schneefuss et al.'s benchmarking framework including their topology and streamset generation. After that, we give a brief introduction to the other schedulers, to which we compare our optimized ILP-based approach. Lastly, we present our benchmarking hardware.

Benchmarking Framework

The benchmarking framework by Schneefuss et al. [SWHD20] allows an easy comparison between different TSN schedulers. It automates the whole evaluation process. Figure 7.8 shows the activity diagram of this process. The process contains two main parts, namely the testset generation and the scheduler execution.

In the first part, the benchmarking framework generates a testset with multiple testcases. For this, the benchmarking framework first generates multiple random topologies. Those are not necessarily factory automation networks. All generated topologies in our testset have the following number of switches (first tuple value) and nodes (second tuple value): $\{(10, 20); (20, 40); (30, 60); (1, 10); (1, 20); (1, 30); (50, 100); (100, 200)\}$. All other topology details are equal to our previous testset. This means, all switches have a constant and equal processing delay of 2,000 ns and all links have a constant and equal propagation delay of 200 ns and link speed of 1 $\frac{\text{Gbit}}{\text{s}}$. The framework then generates multiple streamsets for each topology with the number of streams being $10, 50, 100, 250$ or 500. While the cycle time is still 1,000,000 ns, streams in this testset also contain a maximum end-to-end delay.

The second part is the scheduler execution itself. Each scheduler instance runs in a docker container on one of the machines in the benchmarking cluster. The benchmarking framework automatically distributes all test cases to the scheduler instances of a scheduler. It then saves the schedulers output and runtime combined with the id of the test case in a database. In order to gain the benchmarking results, we access the data in this database.

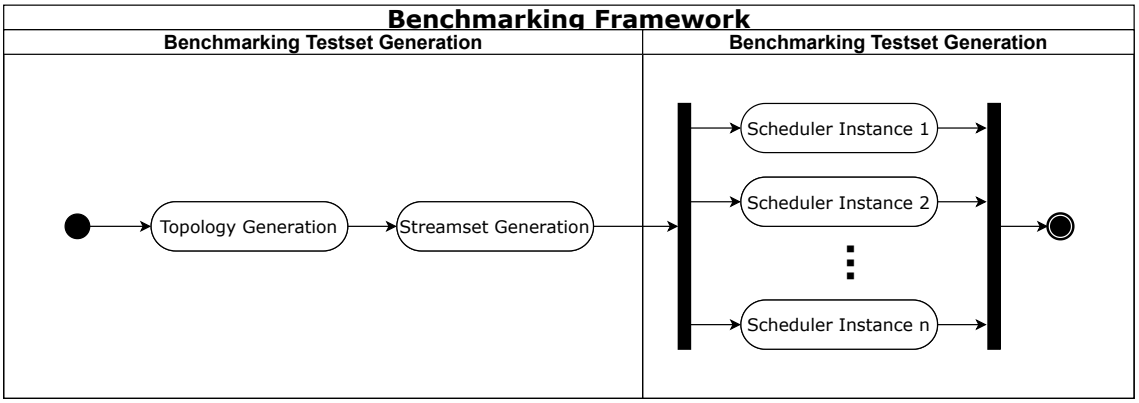


Figure 7.8: Process of Schneefuss et al.'s benchmarking framework [SWHD20].

Schedulers

In the benchmarking part, we compare our ILP-based scheduler to two other schedulers. These are Dürr et al.'s JSSP-based scheduler and Glavackij's tracing-based scheduler. We already introduced their functionality in Chapter 3. In the following, we provide a more detailed view on their functionality.

As already described, Dürr et al. map the no-wait JSSP problem to the no-wait packet scheduling problem. To further improve the runtime, they adapt a tabu-search heurisitc to the packet scheduling problem. In our benchmark, we only use the heuristic-enhanced approach. Dürr et al.'s approach is a scheduling-only approach, thus it needs a fixed route to calculate schedules for. It is capable of calculating the shortest route on its own or to use a given route. We use the possibility to provide a precalculated route. This is the same route that we also use for our routing variable hints. The JSSP scheduler does not take stream-specific end-to-end limitations into account and thus qualifies a schedule as valid if all streams are fully scheduled within the cycle time. As the JSSP scheduler is a single-threaded approach, it is only able to utilize one core per instance. We only use this scheduler to benchmark Store-and-Forward test cases, as it does not support Cut-Through.

The tracing-based scheduler by Glavackij also is a scheduling-only approach, thus we again provide the precalculated path. We use Glavackij's Monte Carlo Tree Search (MCTS)-based approach (*Link Time MCTS*), as it has the best scheduling capabilities. As the tracing-based scheduler allows to limit the runtime, we limit it to 30 min = 1,800 s, as this is a reasonable amount of time for schedule calculation in our opinion. Glavackij's tracing-based scheduler is also single-threaded, and thus also only utilizes one core.

Our ILP-based scheduler is a JRaS approach, thus we do not necessarily need precalculated routes. But as described in Section 6.4.3, we use them to calculate schedule hints. We use our best optimization combination for Store-and-Forward and Cut-Through respectively. These are *ilp-cut-noend-red+conf+adv-bounds-routehint* and *ilp-red+conf+adv-bounds-routehint*, as described in Section 7.1.2. We also limit our runtime to 1,800 s. Our optimization calculations are also single-threaded, but we allow Gurobi to utilize all cores of its node.

OS	Ubuntu 18.04.5 LTS
Kernel	Linux 4.15.0-118-generic
CPU	Intel(R) Xeon(R) W-2145 CPU @ 3.70GHz (8 Cores, 16 Threads)
Memory	4×8 GB DIMM DDR4 Synchronous 2666 MHz

Table 7.4: Hardware for benchmarking.

Hardware

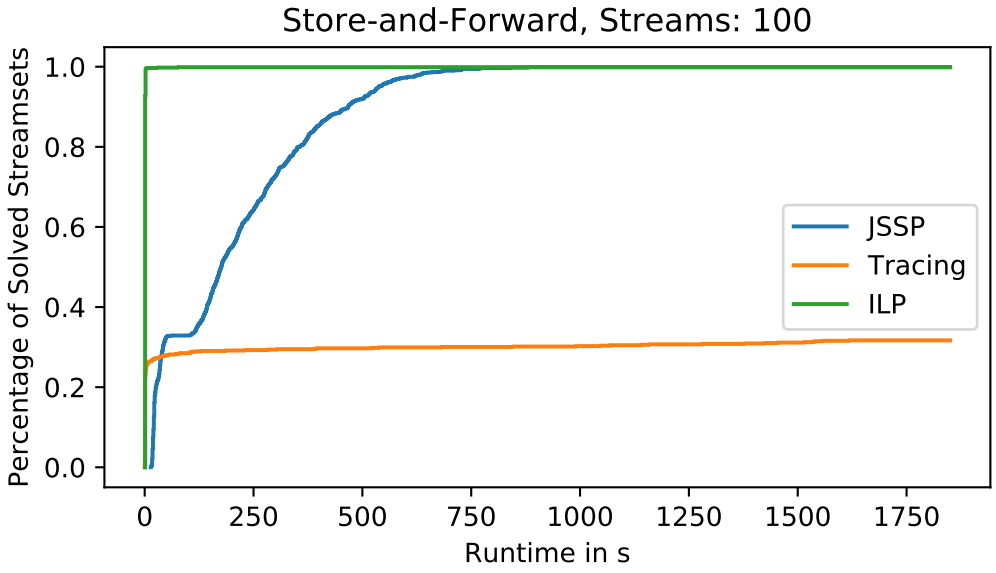
We run the scheduler instances of the benchmarking framework on a cluster of identical nodes. Each node only executes exactly one instance of a scheduler at a time. This especially means, that each scheduler may utilize all resources of its node at any time. Table 7.4 shows the specification of each identical benchmarking node in the cluster.

7.2.2 Benchmarking Results

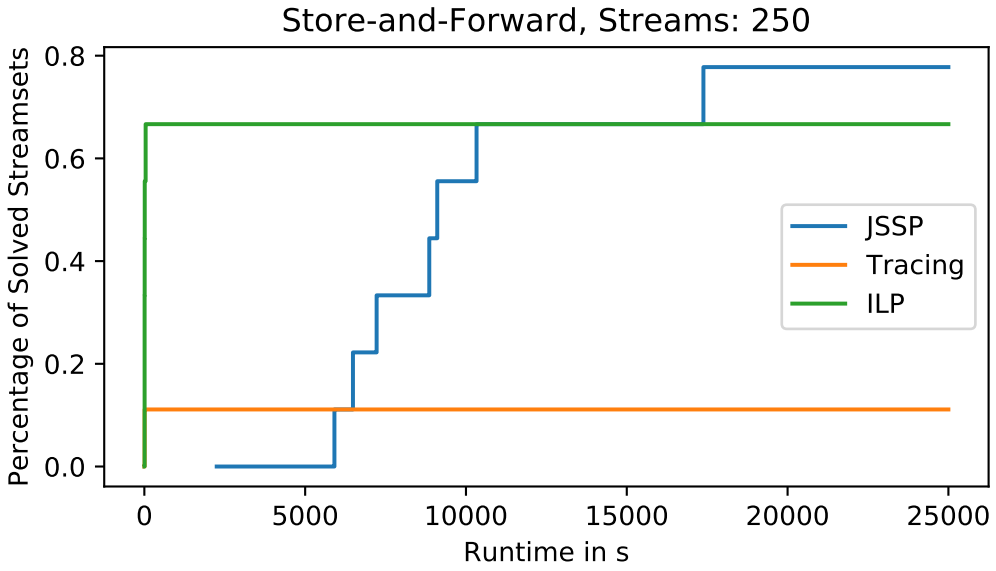
In this section, we now present the benchmarking results for the three different schedulers. Figure 7.9 contains two examplary benchmarking results. In Figure 7.9a we show the results for all test cases with 100 streams. We can see, that our scheduler is able to solve these test cases in only a few seconds. Dürr et al.'s JSSP-based solver solves about 35 % of the testcases within the first 100 seconds and then approaches 100 % between 200 s and 750 s. The tracing-based scheduler by Glavackij reaches about 30 % within the first few seconds, but then only increases slowly and reaches about 35 % at the runtime limit. For test-cases with a smaller number of streams, the graph looks similar. In these cases, the JSSP-based scheduler reaches 100 % faster and the tracing-based scheduler is able to solve a greater number of test cases.

Even though we limited the runtime of our approach and Glavackij's tracing-based scheduler to 1,800 s, we provide the evaluation for 250 streams up to a timescale of 20,000 s, as it allows us to include the JSSP-based scheduler. The tracing-based scheduler is again able to solve some test cases within the first few seconds, but is unable to improve the number of solved test cases within the remaining runtime. Our approach reaches a threshold of about 70 % within the first 100 s, but we are unable to improve this value within the remaining runtime. The JSSP-based scheduler has a hugely higher runtime, however it is able to steadily increase the number of solved test-cases. For greater numbers of streams, the behavior is again similar: The percentage of solved test cases decreases for our as well as for the tracing-based scheduler and the runtime for our solved test cases increases further. The runtime of the JSSP-based scheduler increases further, but it is still able to solve these test-cases after a large amount of time.

These evaluation results show, that our optimized approach is capable of scheduling a large number of problem instances within a reasonable amount of time and can compete with different other scheduling approaches. Further, we assume it is a benefit to use Joint Routing and Scheduling (JRaS) approaches, as they are able to chose between different possible routes, and thus are more flexible in resolving conflict streams.



(a) Benchmarking results for 100 streams.



(b) Benchmarking results for 250 streams.

Figure 7.9: Store-and-Forward benchmarking results for our optimized ILP-based approach, Dürr et al.'s JSSP scheduler and Glavackij's tracing-based scheduler.

8 Conclusion

In this work, we optimized the ILP-based *Joint Routing and Scheduling (JRaS)* problem for *Time-sensitive Networking (TSN)*. We mainly focused on improving an already existing *No-wait JRaS* ILP-model by Hellmanns et al. Therefore, we defined two objectives to measure the improvement of our approaches. These two objectives are the scheduling capability and the runtime. We provided optimization approaches of different categories to improve upon these objectives, and enhanced the model with additional functionality. Lastly, we evaluated our optimization approaches and benchmarked our best combination against other schedulers.

The main contribution of our work is an ILP-model with additional features, an increased scheduling capability and on average a reduced runtime. The additional features of the ILP include the support for Cut-Through switching and stream-specific end-to-end limitations. We reached our goal of an increased scheduling capability and reduced runtimes by providing optimization approaches in three different categories. The first category includes the reduction of our ILP-model by reducing the underlying topology model, e.g. by removing vertices on paths, which are only accessible with loops and by merging edges without routing decisions on intermediate vertices. In the second category, we presented two different approaches, which modify our ILP generation process. Our first approach in this category removes unnecessary auxiliary variables and the second approach provides additional bounds for variables to limit the search space of the ILP-solver. Optimization approaches of the last category make use of Gurobi-specific functionalities, such as different constraint types, solution hints and solving-process specific parameters.

The evaluation of our approaches shows, that there are huge differences in the results between the three optimization categories, when using them individually. Optimizations of the first category result in a shorter runtime and an increased scheduling capability in nearly all cases. In contrast, approaches of the last category often result in higher runtimes and a reduced scheduling capability. However, when combining multiple approaches of different categories, we are able to find combinations for Store-and-Forward and Cut-Through switching respectively, which reduce the runtime by about 80 % and also result in an increased scheduling capability. The benchmarking confirms, that our optimized ILP-based approach is able to compete with other schedulers.

Outlook

While we already developed and presented optimizations for ILP-based scheduling approaches, there are multiple possibilities to improve upon them in a future work. In this last section of our work, we present possible improvements.

In our evaluation, we already analyzed the scheduling capabilities and runtimes of different optimizations individually and in combination. Future work might take a deeper look, how the performance of these optimizations depends on different metrics. Examples for such metrics are the topology type and size, the degree of intermeshing in those topologies and the number and size of streams among others. It might be possible, that the best optimization combination depends on these metrics.

We already investigated the influence of some of these metrics (topology size and number of streams) on the MIPFocus parameter, but we were not able to find correlations. However, with more sophisticated metrics, it might be possible to find a correlation between one or multiple of these metrics and to find the optimal value for the MIPFocus parameter or other parameters.

In its current state, our ILP-model always calculates routes and schedules from scratch. However, changes in a network are common, and thus one might need to add a new stream to an already existing schedule. Future work might extend our ILP-based approach to support a rescheduling mechanism, which adapts to changes in the network topology and/or streamset without the need of calculating a new schedule from scratch.

Bibliography

[Cis08] Cisco. *Cut-Through and Store-and-Forward Ethernet Switching for Low-Latency Environments*. 2008. URL: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-5020-switch/white_paper_c11-465436.html (visited on 09/26/2020) (cit. on p. 20).

[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 p. 30).

[CO15] S. S. Craciunas, R. S. Oliver. "Combined task- and network-level scheduling for distributed time-triggered systems". In: *Real-Time Systems* 52.2 (Oct. 2015), pp. 161–200. DOI: 10.1007/s11241-015-9244-x (cit. on p. 30).

[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 p. 30).

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on p. 27).

[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 '16. Brest, France: Association for Computing Machinery, 2016, pp. 203–212. ISBN: 9781450347877. DOI: 10.1145/2997465.2997494. URL: https://doi.org/10.1145/2997465.2997494 (cit. on pp. 30, 32).

[Dut14] B. Dutertre. "Yices 2.2". In: *Computer-Aided Verification (CAV'2014)*. Ed. by A. Biere, R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, July 2014, pp. 737–744 (cit. on p. 30).

[FDR18] J. Falk, F. Dürr, K. Rothermel. "Exploring Practical Limitations of Joint Routing and Scheduling for TSN with ILP". In: *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2018, pp. 136–146 (cit. on p. 32).

[Fre13] L. Frenzel. *Fundamentals of Communications Access Technologies: FDMA, TDMA, CDMA, OFDMA, AND SDMA*. Jan. 22, 2013. URL: https://www.electronicdesign.com/technologies/communications/article/21802209/fundamentals-of-communications-access-technologies-fdma-tdma-cdma-ofdma-and-sdma (visited on 08/25/2020) (cit. on p. 24).

[Gla20] A. Glavackij. "Tracing-based Scheduling of Isochronous Traffic in Time-Sensitive Networks". Bachelor's Thesis. Aug. 19, 2020 (cit. on p. 33).

[Gra66] R. L. Graham. "Bounds for certain multiprocessing anomalies". In: *The Bell System Technical Journal* 45.9 (June 11, 1966), pp. 1563–1581 (cit. on p. 29).

[Gur] Gurobi. *Gurobi Optimizer*. Gurobi Optimization. URL: https://www.gurobi.com/ (visited on 06/09/2020) (cit. on pp. 25, 30, 40, 66).

[HBŠ10] Z. Hanzálek, P. Burget, P. Šůcha. "Profinet IO IRT Message Scheduling With Temporal Constraints". In: *IEEE Transactions on Industrial Informatics* 6.3 (2010), pp. 369–380 (cit. on pp. 30, 32).

[HDHK18] D. Hellmanns, F. Dürr, R. Hummen, S. Kehrer. In: *Reducing Runtime of Schedule and Route Synthesis in TSN without Sacrificing Valid Solutions*. 2018. DOI: 10.4230/LIPIcs.ECRTS.2018.YY (cit. on pp. 16, 23, 31, 35, 37, 47).

[IEEa] IEEE. *IEEE 802.1 Working Group*. URL: https://1.ieee802.org/ (visited on 08/25/2020) (cit. on p. 22).

[IEEb] IEEE. *Time-Sensitive Networking (TSN) Task Group*. URL: https://1.ieee802.org/tsn/ (visited on 08/25/2020) (cit. on p. 22).

[IEE10] IEEE. "IEEE Standard for Local and metropolitan area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams". In: *IEEE Std 802.1Qav-2009 (Amendment to IEEE Std 802.1Q-2005)* (2010), pp. 1–72 (cit. on p. 23).

[IEE16] 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. 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. 18–24).

[IEE20a] IEEE. "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008)* (2020), pp. 1–499 (cit. on p. 22).

[IEE20b] IEEE. "IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications". In: *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)* (2020), pp. 1–421 (cit. on p. 22).

[MMR99] R. Macchiaroli, S. Mole, S. Riemma. “Modelling and optimization of industrial manufacturing processes subject to no-wait constraints”. In: *International Journal of Production Research* 37.11 (July 1999), pp. 2585–2607. DOI: 10.1080/002075499190671 (cit. on p. 32).

[Pei14] T. P. Peixoto. “The graph-tool python library”. In: *figshare* (2014). DOI: 10.6084/m9.figshare.1164194. URL: http://figshare.com/articles/graph_tool/1164194 (visited on 09/10/2014) (cit. on p. 26).

[PRGS18] P. Pop, M. L. Raagaard, M. Gutierrez, W. Steiner. “Enabling Fog Computing for Industrial Automation Through Time-Sensitive Networking (TSN)”. In: *IEEE Communications Standards Magazine* 2.2 (2018), pp. 55–61 (cit. on p. 33).

[PRO14] PROFIBUS Nutzerorganisation e. V. “PROFINET System Description”. In: (2014) (cit. on p. 29).

[SAE16] SAE International. *Time-Triggered Ethernet*. Nov. 9, 2016. DOI: 10.4271/as6802 (cit. on pp. 15, 29).

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. USA: John Wiley Sons, Inc., 1986. ISBN: 0471908541 (cit. on p. 25).

[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. 31).

[Ste10] W. Steiner. “An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks”. In: *2010 31st IEEE Real-Time Systems Symposium*. 2010, pp. 375–384 (cit. on pp. 15, 24, 30).

[STP+20] E. Schweissguth, D. Timmermann, H. Parzyjegla, P. Danielis, G. Mühl. “ILP-Based Routing and Scheduling of Multicast Realtime Traffic in Time-Sensitive Networks”. In: *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. 2020, pp. 1–11 (cit. on p. 31).

[SWHD20] P. Schneefuss, M. Weitbrecht, D. Hellmanns, F. Dürr. “Benchmarking TSN Schedulers”. In: (2020) (cit. on pp. 73, 74).

[Tar71] R. Tarjan. “Depth-first search and linear graph algorithms”. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 1971, pp. 114–121 (cit. on p. 26).

[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 p. 41).

[Wil93] R. N. Williams. “A Painless Guide to CRC Error Detection Algorithms V3.00”. In: (Aug. 19, 1993) (cit. on p. 18).

All links were last followed on October 28, 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