Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

ILP-based Schedule Planning for Dynamic IEEE Time-Sensitive Networks

Michel Weitbrecht

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	Dr. rer. nat. Frank Dürr
Commenced:	August 15, 2021
Completed:	February 15, 2022

Abstract

Time-Sensitive Networking (TSN) denotes a set of IEEE standards that amend Ethernet by realtime capabilities. Specifically, the *Time-Aware Shaper* (TAS) implements a *time division multiple access* (TDMA) scheme allowing to temporally isolate time-sensitive traffic. In order to deploy a TSN network configuration providing hard bounds on delay and jitter, synchronized clocks and a global transmission schedule are required. However, the calculation of such a global schedule is not part of the TSN standard. Most scheduling approaches proposed in the literature target static scenarios where the network topology and the flow set are known a priori. Novel use cases encountered in Industry 4.0 have more dynamic requirements, such as connecting further machines or enabling new features that result in additional flows that need to be admitted into the schedule.

Dynamic TSN scheduling poses challenges not only in adding new flows to an existing schedule, but also for initial offline scheduling, as each placement of flows in the original schedule may decrease future schedulability. In our work, we define an *Integer Linear Programming* (ILP) model that supports incrementally adding new flows to an existing schedule, taking original flow placements into account. It supports both *defensive scheduling*, where existing flows are not temporally moved, as well as *offensive scheduling*, in which existing flows may be rescheduled by a limited amount configurable per flow. We propose two optimization objectives aiming to improve schedulability of future flows. The first objective `max-freeblock` schedules flows in a way that increases the unoccupied time at the end of each schedule cycle time, leaving space to place future flows in defensive scheduling scenarios. The second objective `link-desync` maximizes the unoccupied gap between every two consecutive flows of a link, creating flexibility for future offensive scheduling. Further, we propose a heuristic-based algorithm that strategically selects a reduced set of initial flows for rescheduling based on link occupancy, reducing the model size and decreasing the execution time necessary to place additional flows.

Our evaluations show that our `max-freeblock` strategy outperforms the baseline `min-flowspan` objective in defensive scheduling. Utilizing our rescheduling selection algorithm, 97 % of additional flows were admitted in offensive scheduling scenarios, compared to only 72 % achieved without restricting the rescheduling flow set.

Kurzfassung

Time-Sensitive Networking (TSN) umfasst mehrere IEEE Standards, die den Ethernet-Standard um Echtzeitfähigkeiten erweitern. Der darin beschriebene *Time-Aware Shaper* (TAS) implementiert ein *Time Division Multiple Access* (TDMA) Verfahren um zeitkritischen Datenverkehr zeitlich zu isolieren. TSN-basierte Netzwerke können nur dann Echtzeiteigenschaften wie Delay und Jitter garantieren, wenn alle beteiligten Geräte zeitsynchronisiert sind und einem globalen *Schedule* (Zeitplan) folgen. Die Berechnung eines solchen Schedule bleibt im Standard explizit offen. Die meisten in der Forschung diskutierten Berechnungsverfahren beziehen sich auf statische Szenarien, in denen die Netzwerktopologie und alle Flowspezifikationen im Voraus bekannt sind. Moderne Anwendungsfälle z.B. in der Industrie 4.0 haben wechselnde Anforderungen an zeitkritische Netzwerke, beispielsweise das Hinzufügen weiterer Geräte oder aktivieren zusätzlicher zeitkritischer Funktionen im laufenden Betrieb.

Dynamisches TSN Scheduling stellt nicht nur in Bezug auf das Hinzufügen von Flows im laufenden Betrieb eine Herausforderung dar, sondern betrifft auch das initiale Scheduling, da die Platzierung von Flows die Platzierbarkeit von künftigen Flows beeinträchtigt.

In dieser Arbeit wird ein *Integer-Linear-Programming-Modell* (ILP) formuliert, mit dem neue Flows in einen bestehenden TSN-Schedule unter Beachtung der existierenden Restriktionen eingefügt werden können. Das Modell unterstützt sowohl *defensives Scheduling*, bei dem bestehende Flows nicht verschoben werden dürfen, als auch *offensives Scheduling*, welches das Verschieben bestehender Flows bis zu einem pro Flow definierten Limit ermöglicht. Weiter enthält das Modell zwei Strategien in Form von Optimierungszielen, um die Platzierbarkeit von künftigen Flows zu verbessern. Das erste Optimierungsziel, `max-freeblock`, platziert Flows so innerhalb des Schedules, dass bei jedem Link ein großer freier Bereich zeitlich am Ende des Schedules entsteht, um Platz für die freie Platzierung weiterer Flows freizuhalten. Das zweite Optimierungsziel, `link-desync`, maximiert paarweise den Abstand zwischen allen Flows, die über einen Link geroutet werden und schafft somit Lücken zwischen aufeinanderfolgenden Flows. Die Lücken bieten Flexibilität um sowohl neue Flows direkt darin zu platzieren, als auch um durch geringe Verschiebungen an geschickten Stellen Platz für neue Flows zu schaffen. Weiterhin wurde ein heuristikbasierter Algorithmus implementiert, der basierend auf den Linkbelegungen strategisch eine reduzierte Menge bestehender Flows auswählt, bei denen eine zeitliche Verschiebung erlaubt wird. Die Einschränkung der Verschiebbarkeit auf diese Teilmenge reduziert die Größe der Probleminstanz und damit die benötigte Laufzeit um zusätzliche Flows im Schedule zu platzieren.

Evaluationen der Ansätze zeigen, dass die `max-freeblock`-Strategie bessere Ergebnisse im defensiven Scheduling erzielt als die Vergleichsstrategie `min-flowspan` aus der Literatur. Durch den Verschiebbarkeitsalgorithmus können bei offensivem Scheduling durchschnittlich 97 % der zusätzlichen Flows platziert werden, während ohne Einschränkung der Verschiebbarkeit nur 72 % der Flows aufgenommen werden können.

Contents

List of Figures

1 Introduction

A *cyber-physical system* (CPS) is a set of distributed devices that communicate via a network and interact with the physical world with sensors and actuators. CPS are employed in many fields including the (Industrial) Internet of Things [XYGG18], medical care systems [LS10] or autonomous driving [CYH+17]. Typically, CPS consist of software-based controller devices, sensors and actuators that interact in so-called closed control loops. This means that the system regulates a physical target by constantly measuring using sensors, calculating necessary actions with the controller and possibly adapting the target using actuators. A simple example would be a medical system that measures the oxygen level of a medical patient, calculating a moving average of it and adapting the saturation of the oxygen mask as soon as a threshold is crossed.

The underlying network of a CPS must provide reliable real-time communication, otherwise the quality of the physical process decreases or much worse, the system will not work reliably and safely anymore. Therefore, the network must provide hard bounds on end-to-end delays and jitter (variance of packet arrival times). Traditionally, field buses have been employed for reliable real-time communication, e.g. using a CAN bus within a car. With ever-expanding CPS like smart factories or the industrial Internet of Things, field buses are not suitable anymore, as the network link is shared between all devices, leading to reduced bandwidth per device, inflexible placement restrictions and error-prone behavior.

The prevalence, reliability and performance of Ethernet have previously motivated industrial groups to bring hard real-time guarantees into Ethernet and even combine such time-critical traffic with conventional low priority traffic in a converged network. Initially, industrial groups created technologies like PROFINET [Fel04] and TTEthernet [Eth16] which provide such guarantees based on Ethernet. The Institute of Electrical and Electronics Engineers (IEEE), the standard body of Ethernet, also saw the relevance of aforementioned properties and founded a task group for Time-Sensitive Networking (TSN). This task group created a set of standards to bring support for time-critical communication into standard Ethernet networks while concurrently supporting regular best-effort traffic. In particular, they specified the Time-Aware Shaper (TAS), a time division multiple access (TDMA) scheme to exclusively reserve Ethernet link access for certain traffic. Based on traffic classification, per-traffic-class queues and a gating mechanism, low priority traffic can be buffered in order to forward high-priority Ethernet frames with minimal delay.

Combined with PTP time synchronization [20a], this allows for deterministic time-triggered Ethernet communication. For this, all switches along the path of a time-triggered traffic flow need to be synchronized and their gate control lists (GCLs) configured to exclusively reserve the respective links for this flow. Calculation of a global schedule that reserves the respective link times of all time-critical flows across all switches is an $\mathcal{NP}$-hard problem [DN16; SL86] and not part of the TSN standards. Therefore, different approaches have been proposed in the literature or adapted from

existing solutions for similar planning problems [COCS16; DN16; FDR18; FGD+21; GRK+21; NDR17; PHL+20; PSRH15; SDT+17]. Most approaches assume a static network with a constant set of traffic flows that are known a priori and calculate a valid schedule under these criteria.

The TSN Profile for Industrial Automation considers many use cases [BDE+18] that require a more dynamic approach able to adapt the global schedule with changing requirements. For example, live connection of subnetworks, additional hardware or software and feature upgrades require dynamic addition of traffic flows to the global schedule. A straightforward approach to schedule additional flows would be to re-execute the original offline computation with the extended flow set. Aside from its typically long runtime, this approach is unsuitable as it may move and disrupt already active flows, violating their jitter constraints. Therefore, dynamic adaptation of TSN devices and flows requires a scheduling process that both takes into account the existing flows and computes a new schedule within a short time period.

Dynamic TSN scheduling poses challenges both for the online scheduling process of adding flows and for the initial scheduling process. Of course, the dynamic scheduler needs to accommodate new flows and hosts into an existing schedule in a relatively short time frame. But the initial scheduling strategy also affects the schedulability of later-added flows, as positioning of the initial flows may be suboptimal in combination with them. The challenge is to find a strategy that achieves a high schedulability without knowing and preparing for the actual added flows. Some dynamic scheduling approaches were proposed in the literature, with most of them relying only on heuristics to schedule flows.

In this work, we make the following contributions:

1. We define an ILP model based on [DN16] that schedules TSN flows while supporting different strategies to better accommodate additional flows later-on without knowing their requirements a priori.
2. We propose an adapted version of this model that allows to incrementally schedule additional flows into an existing schedule. The model optionally allows for rescheduling existing flows adhering to a per-flow defined jitter tolerance.
3. We introduce heuristic-based algorithms to select the set of flows to be rescheduled.
4. We evaluate all proposed models and algorithms with respect to their runtime and schedulability of additional flows based on a factory network topology and random flows.

We structure our work as follows: In Chapter 2, we introduce some technical background, in particular Time Sensitive Networking (TSN) and Integer Linear Programming (ILP). Further, we look into related work such as a static no-queuing ILP-based TSN scheduling approach [DN16] as well as dynamic scheduling approaches based on flexibility curves [GRK+21], conflict graphs [FGD+21] and solution-space-reduced ILP [NDR17]. In Chapter 3, we describe the system model with all assumptions we make as well as the definition of the problem to be solved. In Chapter 4, we explain and define our main contributions, namely the ILP model adaptations for consideration and rescheduling of previous schedules and for preparation of initial schedules for higher schedulability of amended flows. We also introduce an algorithm for selecting and weighting the set of initial flows to be rescheduled. Next, we evaluate all proposed models and algorithms with server- and desktop-grade computers in Chapter 5. In particular, we examine the runtime, complexity and

solution quality of the proposed scheduling strategies as well as the effect of tuning parameters of our dynamic rescheduling algorithm. Lastly, we summarize our work and discuss possible future work.

2 Background and related work

2.1 Background

2.1.1 Time-Sensitive Networking

Time-Sensitive Networking (TSN) describes a set of IEEE standards that bring real-time communication properties to Ethernet-based network infrastructures. In its original form, Ethernet lacks mechanisms to guarantee real-time properties such as timeliness, end-to-end latency and jitter bounds. This is because all frames are treated with the same priority in a *best effort* manner, possibly dropping or delaying frames unpredictably when network congestion occurs. Applications heavily depending on real time guarantees previously had to use field buses such as CAN [HMFH+00] or FlexRay [FHHW03], Ethernet-based third-party solutions like TTEthernet [Eth16], PROFINET [Fel04], or even use point-to-point connections. Those approaches have many shortcomings such as increased costs, feature and performance limitations, missing interoperability or even vendor lock-ins.

The IEEE TSN Task Group [Ins20] shared these concerns and published multiple amendments to their IEEE 802.1Q-2014 [14] standard (VLAN Tagging) that enable TSN in 2015 and 2016. The group previously concerned itself with Audio and Video Bridging under a different name, publishing standards relating to reliable low-latency transmission of audio and video streams in 2010. The TSN standards bring a variety of mechanisms into Ethernet that aim at facilitating reliable low-latency communication, such as traffic shaping, improved stream reservation, explicit routing and multi-path replication.

In our work, the most relevant TSN standard is 802.1Qbv [Ins16b], which introduces scheduled traffic, i.e. forwarding Ethernet frames according to a global timetable or *schedule* using a time division multiple access (TDMA) scheme. Figure 2.1 shows the queuing network located at every egress port of every TSN-enabled switch. Frames to be forwarded arrive from the top and get inserted into one of the eight queues according to the 3-bit priority code point value within their VLAN tag. At the bottom, the *transmission selection* component selects the next frame to be transmitted over the medium according to the priorities of the seven queues. Frames enqueued in a lower-prioritized queue will only be transmitted once no frame of any higher-prioritized queue is available. This is not only the case when a queue is empty, but more importantly when the transmission gate in front of the queue is closed. The transmission gates can either be opened, allowing access to the queue, or closed, denying it. The so-called *Time Aware Shaper* (TAS) controls the gate states according to the *port schedule*, a configurable timetable. For discrete points in time, the schedule dictates the binary state of all eight transmission gates with a bit vector. The schedule as a whole is cyclic, i.e. it will be repeated after each full execution, requiring the time parameter in each schedule entry to be relative to the cycle time.

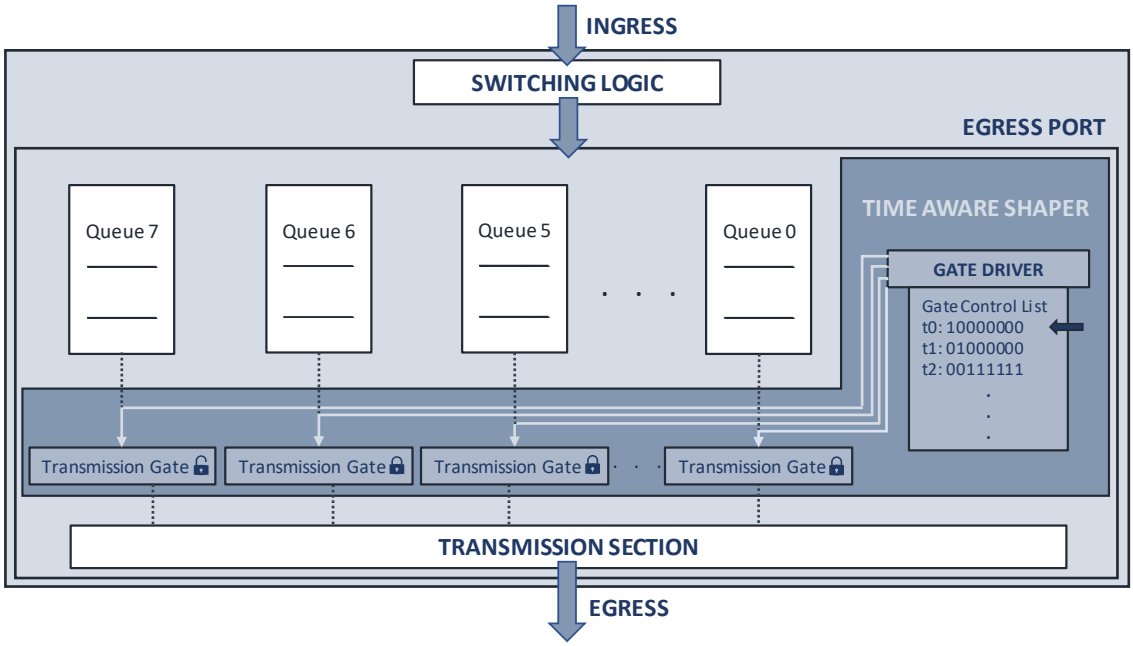


Figure 2.1: The queuing network positioned logically before each egress port of a TSN-capable Ethernet switch

The TAS mechanism in combination with prioritized queues allows for granting exclusive access to the physical medium to a single traffic class for certain time periods, thereby guaranteeing hard real-time properties such as bounded end-to-end latency and jitter. Note that each switch port has its own schedule that only concerns itself with frames and priorities on that exact port. In order to guarantee real-time properties of a time-critical flow transferred via a network consisting of multiple switches, the schedules of all switch ports on the flow's path as well as the schedule of the source host must be configured to exclusively grant access to the medium at the correct time. Of course, following the distributed schedule requires very precise synchronized clocks on every participating device, which is facilitated by another TSN standard, IEEE 802.1AS [20a]. The schedule calculation is very specific to the application requirements and not part of the TSN standards, i.e. it is up to the network administrator to calculate a coordinated schedule that meets their requirements.

As the calculation of such coordinated schedules is an $\mathcal{NP}$-hard problem [DN16; SL86], Section 2.2 will introduce different approaches to calculate such a global schedule. Typically, the input of such a calculation is a set of flows defined by source and destination hosts, a payload size, end-to-end latency requirements and a flow cycle, i.e. the timespan until the periodic flow is sent again. In order to schedule flows with differing cycle periods, typically a so-called *hyper-cycle* is employed network-wide, i.e. the least common multiple of all available cycle periods. Some algorithms assume static flow paths while others include route selection as a further dimension in their scheduling model, allowing to perform e.g. balancing the traffic load across multiple alternative routes, increasing overall capacity. Many scheduling goals are imaginable above fulfilling the individual flow requirements, such as load-balancing across links and switches, creating blocks consisting only of scheduled traffic or desynchronizing link occupations. While TSN enables real-time guarantees for some flows or traffic classes, low-priority, so-called *best-effort traffic* may still be transferred in the network without interference by using a different priority and queue as

described above. Such traffic will be queued up at switch egress ports while high-priority traffic is transferred, but will still be delivered on a best-effort basis. It may therefore also be a criterion to optimize a schedule for a high remaining block of link capacity for best-effort traffic or even many small blocks, depending on the expected traffic patterns exhibited by the low-prioritized traffic.

2.1.2 Network Delays

Four different types of delay may occur when transferring an Ethernet frame through a switched network and are relevant when scheduling traffic flows. Figure 2.2 illustrates the four types of delay based on an example frame that is forwarded through multiple switches.

Transmission Delay The *transmission delay* occurs while actually modulating the full frame as bits onto the transmission medium, e.g. by switching voltages on a copper link or sending light through a glass fiber link. The length of the transmission delay depends on the number of bytes to be transmitted as well as the link speed i.e. the delay on a 1 Gbps link is much shorter than on a 10 Gbps link.

Propagation Delay The time that elapses between modulation at the sender and demodulation at the recipient, i.e. the time the information *propagates* through the transmission medium, is called *propagation delay* and depends on the type of medium and the length of the actual link.

Processing Delay When the full frame is received, it needs to be interpreted and processed within the switch, e.g. checksums are verified, headers are interpreted and the frame is passed on to the port it needs to be sent out next. This timespan is called *processing delay* and, small variations aside, can be viewed as a static delay in modern switches.

Queuing Delay At the egress port, the frame will be put into a queue that is normally processed in FIFO order (first in first out). When the queue is not empty, the frame can only be forwarded as soon as all frames in front of it in the queue are processed. The delay imposed by this queueing is called *queuing delay* and of course depends on the particular number of frames waiting to be forwarded on the same link. Without TSN traffic classes and gating mechanisms, high queueing delays and a high jitter, i.e. the variance of end-to-end delays, may occur when the network is congested.

2.1.3 Integer Linear Programming

Linear Programming (LP) is a standard way in computer science to model an optimization problem via mathematical inequalities. Note that the term programming has nothing to do with the traditional computer programming of algorithms, but is rather a way to express an optimization problem by way of mathematical inequalities. An LP model consists of variables, constraints and an objective function, where the latter two must be linear. Linear programs can be interpreted geometrically in the Euclidian space with all constraints represented as halfplanes where all allowed variable assignments lie on one side of the halfplane [SF20]. The intersection of all such halfplanes forms a convex hull in which all *feasible solutions*, i.e. all variable assignments not violating any constraints lie. The optimal solution to the linear program now has to lie in one of the corners of the convex hull. An example equation and its respective geometric interpretation can be observed in Figure 2.3. The example equation is in standard form, meaning that all constraints are in form of *less than or*

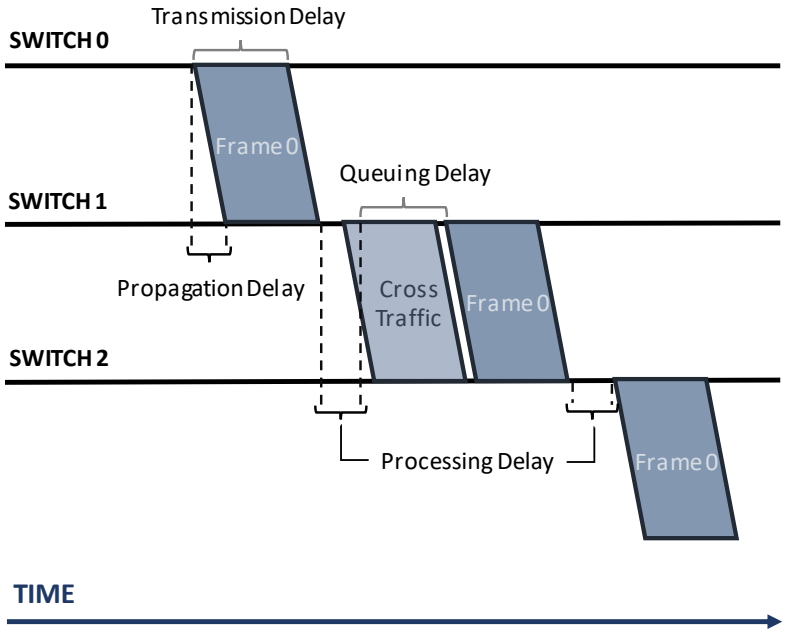


Figure 2.2: Temporal behavior and imposed delays on an Ethernet frame when forwarding it via multiple hops (exemplary without scale and units)

equal inequalities and the objective maximizes a linear term containing all variables. The standard form is necessary as it is the starting point for the simplex algorithm prevalently used for solving linear programs.

In standard linear programming, the variables may only be constrained by lower or upper bounds, but no restrictions on contiguous values are imposed, i.e. all variables belong to $\mathbb{R}$. In *Integer* Linear Programming (ILP) or *Mixed* Integer Linear Programming (MILP), all or some variables may be further constrained to taking only integral values, or even only binary values. It may not be obvious at first that integer restrictions heavily complicate finding a feasible solution to a given ILP. In contrast to Integer Programming, not all points in the aforementioned convex hull form a valid solution and while finding the optimal value with respect to the objective function by combining all constraints was comparably easy before, most corners of the convex hull are not feasible and the optimal solution may not lie in such a corner at all. Note that the term ILP is often used synonymously for MILP in the literature.

LPs are typically solved by commercial solver programs like CPLEX[1] or Gurobi[2], which are both readily available for free use in academic applications. Those solvers use many algorithms, heuristics and optimizations to improve performance, most of which may be tuned and configured individually. A great benefit of those solvers is that they not only find the optimal solution to the given problem formulation, but they also prove optimality. Even for early-spotted non-optimal solutions, they can quantify how good the solution is in terms of the objective function value, and give the so-called *MIP gap*, i.e. to what percentage the solution is optimal, based on the currently

[1] https://www.ibm.com/products/ilog-cplex-optimization-studio
[2] https://www.gurobi.com/products/gurobi-optimizer/

$$
\begin{array}{lrcrcrl}
\max & & x & + & y & & \\
\text{s.t.} & - & x & - & y & \leq -3 & (c1) \\
 & 0.5 & x & + & y & \leq 7 & (c2) \\
 & 0.5 & x & - & y & \leq 0 & (c3) \\
 & -0.5 & x & + & y & \leq 3 & (c4)
\end{array}
$$

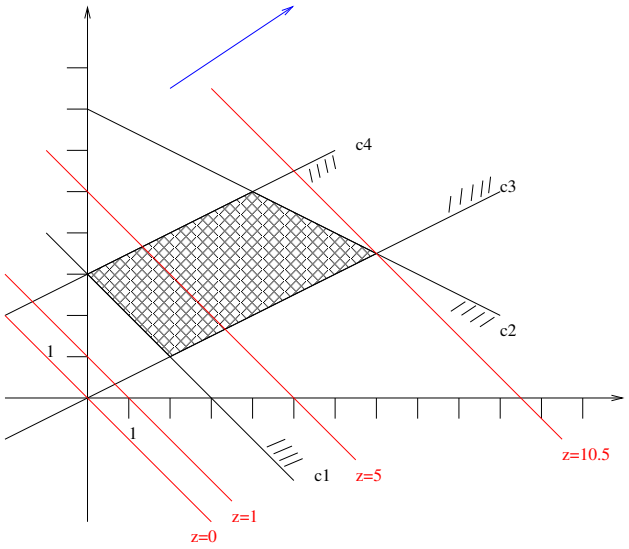


Figure 2.3: Example equation of a linear program and the corresponding geometric interpretation showing halfplanes (black lines), the feasible region (hatched) and different objective values (red lines). Source: [SF20]

known bounds of the problem space. While solving a problem to optimality may take a very long period of time, most solvers also allow stopping the computation after a given time has elapsed or as soon as a certain MIP gap is reached, returning the best solution encountered until then [Gur21]. Because of the aforementioned characteristics, linear programming is often used when solving problems to which no (optimal) solving algorithm exists.

2.2 Related Work

In this section, we first introduce the No-Wait Packet Scheduling problem our work is based on. Then, we discuss three approaches to dynamic flow scheduling found in the literature.

2.2.1 No-Wait Packet Scheduling

No-wait Packet Scheduling for IEEE Time-Sensitive Networks [DN16] is an often cited approach for TSN scheduling that models the problem using the well known *Job-Shop-Scheduling Problem*, an $\mathcal{NP}$-hard [SL86] optimization problem. Given a set of machines, a set of operations and a set of jobs to be executed, a timetable must be calculated that fulfills all constraints and optimizes an objective.

The Job-Shop-Scheduling problem is defined as follows: Every job consists of a set of operations that must be performed in an order specific to the job. Every machine has a set of operations it is capable of, and may only work on at most one job at a time. The processing of a job at a machine takes a defined timespan depending on the machine, the operation and the job and may not be paused or interrupted within a machine. Given these constraints, a timetable must be

computed that schedules every job at a set of machines capable of processing it. Typically, the optimization objective is to minimize the *makespan*, i.e. the timespan between starting the first job and completing the last job.

There are many variants of the problem that introduce other constraints, possibilities or objectives, e.g. dependencies between jobs, duplicate machines or constraining wait times between machines. [DN16] makes use of the latter adaptation, enforcing that once a job has been started, it must pass all subsequent machines without pause until it is finished. With this extension, the problem is called *No-Wait*-Job-Shop-Scheduling problem.

Dürr and Nayak now model the *No-Wait Packet Scheduling* problem (NW-PSP) as follows: Switch ports and host network interfaces correspond to machines, frame transmissions are mapped to operations. Jobs are defined by the transmission operations needed to forward the frame along the precomputed path of the respective time-sensitive traffic flow. The mapping of transmission operations to machines is defined by all flows that need to pass the respective interface in order to reach their destination. The *no-wait* property forces all queuing delays to be zero, ensuring all flows arrive at their destination with the shortest possible end-to-end delay. This property combined with static delays also allow generating a full TSN schedule based on just the start times of each flow. Although the three remaining delays as described in Subsection 2.1.2 do not all occur at the egress port in practice, their sum can be mapped to the job processing time without violating real world feasibility. The model assumes constant processing, transmission and propagation delays, translating to deterministic processing times and links of the same speed and length as well as same-sized Ethernet frames. These restrictions could easily be rescinded by extending the model with a calculation incorporating differing switch processing times, link speeds and lengths and frame sizes.

Dürr and Nayak define an ILP model for NW-PSP, as well as a heuristic-based algorithm based on *Tabu search* as proposed in [MMR99], using the ILP model as a baseline reference to compare their algorithm against. The approach splits NW-PSP into a time-tabling problem, generating a schedule based on a totally ordered set of flows, and a sequencing problem that computes this total order. The time-tabling problem is approached via a greedy algorithm that assigns the earliest possible start time to every flow in the given order, starting with zero and selecting the first start time that does not violate any constraints, specifically that no switch port can transmit two frames at the same time. The sequencing algorithm is based on *Tabu search* [Glo90], which works by finding some initial solution that is subsequently adapted by a neighborhood function, avoiding loops in the search space via the so-called *Tabu list*, a circular list of the solutions explored last. Neighborhoods are created by selecting the last-finishing flow and inserting or swapping it with every preceding flow, yielding a list of possible configurations, of which the best in terms of the makespan calculated by the timetabling algorithm is selected. After n iterations that do not yield a better result while avoiding neighborhoods in the Tabu list of configurable length, the algorithm terminates.

Afterwards, an optional compression algorithm moves flows by small margins in order to avoid short gaps that lead to a high number of gate events and therefore wasted bandwidth for best-effort traffic. In comparison to a commercial ILP solver limited to a 5 h runtime, their algorithm produces 3 % shorter schedules on average, the worst case is 5 % longer than the ILP solution, taking only up to 3.2 h to calculate a solution for 1.500 flows.

Note that the NW-PSP ILP model introduced in [DN16] is the basis of this work and is explained in more detail in Chapter 3.

2.2.2 Incremental Flow Scheduling and Routing in Time-Sensitive Software-Defined Networks

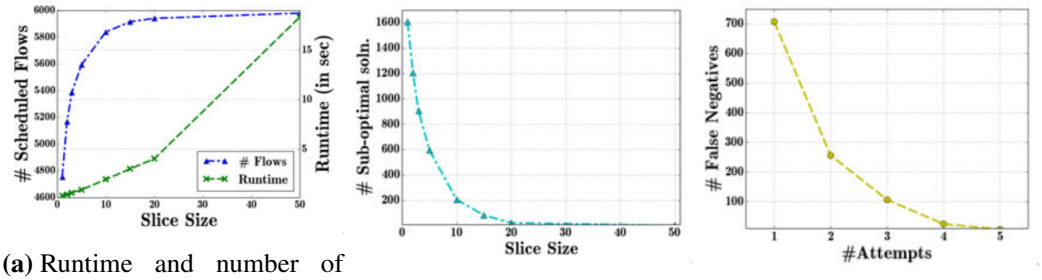
In [NDR17], researchers from IPVS present a dynamic scheduling approach that is based on a so-called Time-Sensitive Software-Defined Network (TSSDN). Software-Defined Networks are a fairly novel network management architecture based on the separation of data and control planes as well as a central network controller with a global view. SDN switches forward regular traffic on the so-called *data plane* and are administered via a separated *control plane* interface. A logically central controller server has a global view onto the network, i.e. it always knows the current topology, switch configurations and can even access live traffic. Based on this information, it can dynamically reconfigure switches at runtime, e.g. modifying forwarding tables, VLANs, or priorities of network traffic.

Nayak et al. make use of the global view in order to calculate TSN schedules for time-triggered traffic and configure the schedules on all respective switches based on the aforementioned management mechanisms. In contrast to most other work that allows both time-triggered and best-effort traffic, the scheduling is implemented only at the time-synchronized hosts, i.e. all TSN gates are opened at all times and the TSN gating mechanism is therefore not used. This allows for easier switching between schedules, because switches only need to be informed about the routes of added flows, but do not have to adapt their port gate schedules. Instead, they use standard priority scheduling to differentiate time-triggered and best-effort traffic, accepting some introduced delay by at most one MTU-sized (*maximum transmission unit*) best-effort frame interfering right before the time-triggered frame would be sent at switch egress ports. They refer to TSN Frame Preemption [Ins16a] to further reduce the end-to-end delays, a mechanism that *preempts* low-prioritized frame transmissions in order to forward high priority frames. This way, at the cost of a slightly higher worst-case delay, they impose no restrictions on other traffic and do not waste bandwidth by closed gates or guard bands. Therefore, switches only need to know the routes of time-triggered flows and hosts only need to send their frames at the right times.

The dynamic scheduling approach is based on an admission control scheme, i.e. a host requests to schedule a flow from the controller, which calculates a schedule based on the flow parameters. If the flow can be placed, the controller informs the host about the timing it must adhere to and reconfigures the SDN switches in order to use the calculated route for this flow based on an exact match on the respective UDP packets. Their system model uses a slot-based mechanism that reserves the full flow path for the entire time the single-frame UDP packet is in transmission. A global base period bp is used across the network, forcing every flow period to be an integral multiple of it. Flows using a higher individual period, say $n \times bp$ only use every n-th phase of their slot, leaving the other phases unoccupied for other flows to be scheduled.

Based on their previously published [NDR16] ILP definitions for static flow scheduling in TSSDNs, [NDR17] presents two heuristic-based algorithms to further narrow down the ILP solution space in order to achieve sub-second flow admission times in the aforementioned dynamic scheme. The *scheduling with unconstrained routing* (S/UR) ILP formulation presented in [NDR16] considers all possible paths between source and destination nodes, i.e. also non-shortest paths, and serves as baseline to compare scheduling quality and execution times against.

As flows are scheduled one-by-one, a single ILP solution cannot take into account future flow requirements, but can try to use the available time and routing space in a way that keeps as many placements open for future flows as possible. Their first algorithm, *Shortest Available Path* (SAP)



(a) Runtime and number of scheduled flows for varying slice sizes

(b) Suboptimally placed flows based on the slice size

(c) False negative rate after reattempts.

Figure 2.4: Solution quality of incremental TSSDN scheduling depending on the slice size (left and center); Reduction of false negative results using reattempts (right). [NDR17]

aims to choose the shortest path that has slots or phases left to accommodate the flow, reasoning that this way the least number of slots are used up by the new flow. The second algorithm *Mini-Max* (MM) tries to spread the traffic load across all instead of only the shortest links, which may block some future flows. This is achieved by configuring the ILP's objective to minimize the maximum number of used slots across all links in the network.

Solving aforementioned dynamic scheduling ILPs still took several seconds, which is why [NDR17] further reduced the solution space and thereby the runtime by use of heuristics. The first optimization prunes all links to hosts that are not involved in the currently to be scheduled flow, which does not impose any limits on schedulability. The second optimization (slicing) may reduce schedulability but drastically reduces the runtime, by limiting the possible slots and phases the scheduler can choose. This is achieved by first ordering the possible slot and phase combinations by the number of core links that are free with this combination and then only allowing the best slice of this ordered array, thereby trying the most promising combinations. While this approach drastically reduces solving times, it may lead to false negatives as not all slot and phase combinations are tried. When admission control fails, further iterations that exclude previously tried combinations may still lead to a viable flow placement.

In evaluations with flow periods equal to the base period and 20-110 flows in networks with a 7-hop diameter, the aforementioned dynamic scheduling formulations SAP and MM achieve an average quality of 72% and 64%, respectively, compared to static S/UR scheduling. As to be expected, higher flow transmission periods leads to more placement options and therefore an increased runtime, as long as the slicing optimization is not employed. The first presented optimization, edge pruning, leads to a runtime reduction of 45%, assuming a realistic fat-tree topology.

To evaluate the more complex slicing optimization, [NDR17] scheduled 10,000 flows on an Erdős–Rényi random graph topology with 20 switches, 500 hosts and 131 core links using SAP both with and without slicing enabled. As can be seen in Figure 2.4a, the runtime increases linearly with increased slice sizes, while only the first few increments lead to large improvements on schedulability. They also still achieve sub-second runtimes for slice sizes less than five. Of course, suboptimally routed flows and false negatives are the main drawbacks of the slicing approach, but both can be controlled using parameters. Figure 2.4b shows the number of subobtimally routed flows depending on the slice size, with about 900 and 600 flows being suboptimally routed with a slice size of 3 and 5, respectively. Using multiple attempts after a negative scheduling execution,

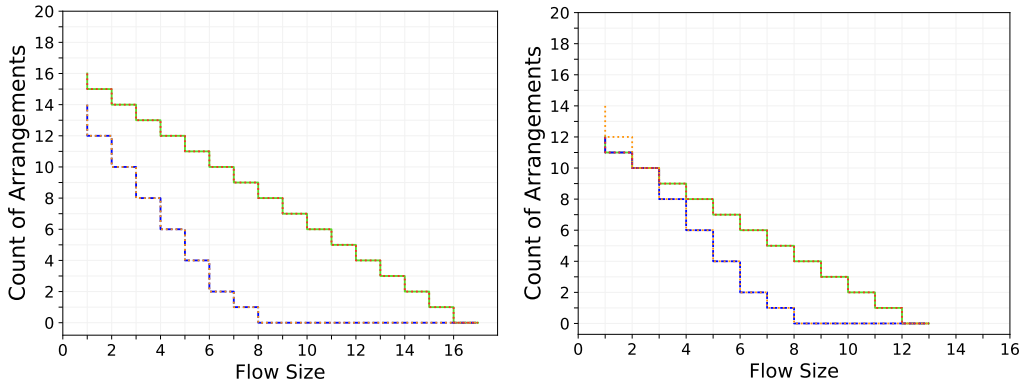


Figure 2.5: Flexibility curves of four routes within an example network, before and after scheduling of another flow that reduces link capacity of all represented routes and therefore affects all flexibility curves. Source: [GRK+21, Fig. 3b, 3c].

the false negative rate can be reduced drastically, as shown in Figure 2.4c. Two reattempts lead to a drastic reduction of false negatives from over 700 to just 106 and nearly remove them in total by a third reattempt. Using the described configuration, a slot size of five and three reattempts, the optimization decreases the number of schedulable flows by only four percent.

2.2.3 Leveraging Flexibility in TSN for Dynamic Reconfigurability

[GRK+21] uses the concept of so-called *flexibility curves* to describe possible ways of embedding new flows into an existing TSN TAS schedule, allowing to perform fast admission control based on precomputed flexibility curves. Their system model assumes dynamically arising tasks that consist of multiple real-time flows, therefore creating the need of an admission control or scheduling mechanism able to insert multiple flows at a time. Dividing the continuous cycle time into Δ-sized slots allowed them to calculate a discrete number of possible embedding times given an existing link schedule and a new flow of known length to be admitted. Each flow start timing along those slots adhering to all constraints leads to a possible embedding, e.g. if there is a block of size 6Δ left unoccupied on a link schedule, three possible start times exist to embed a flow of length 4Δ. The flexibility curve for a source to destination path is now obtained by counting all possible embeddings along a path for all flow lengths of up to one cycle length, as shown in Figure 2.5. Note that the curves are always constrained by the respective bottleneck link on each path, which may be shared with other paths. Therefore, a flow placement not only decreases the residual capacity of that path, but may also affect the curves of other paths that share some or multiple links with the new flow.

This approach mainly targets applications with existing schedules and therefore does not mandate adaptations of the scheduling process. [GRK+21] used an existing SMT (*satisfiability modulo theories*) solver to obtain an initial schedule, based on which they precalculate flexibility curves as basis for their admission control algorithm. Given the flexibility curves for all flows to be admitted, their algorithm decides on admission control of a new flow set based on two approaches: The first part simply checks if a contiguous block for all flows sharing at least one link is free. If such a free block does not exist, a greedy approach is used to schedule flows purely based on flexibility

curves. Flow admission is tried in decreasing flow length, subtracting the current flow length from all affected flexibility curves. Both approaches are only sufficient approaches, i.e. if the algorithm does not confirm placement for the flow set, a classic scheduler still may.

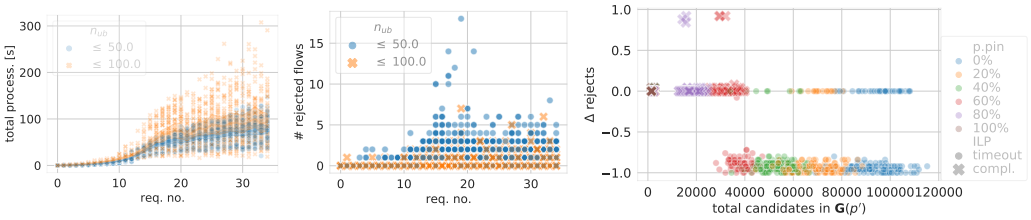
Benchmarked on a simple example (nine used ports, four flows, cycle length of 20Δ), their admission control algorithm schedules the five flow set in under three seconds, compared to an SMT solver requiring over 170 seconds. They also suggest using flexibility curves as a heuristic or objective in initial scheduling in future work, e.g. if multiple routes are possible for a flow, the one reducing the area of all flexibility curves the least could be chosen for better embeddability of future flows.

2.2.4 Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows with Conflict Graphs

In [FGD+21], researchers from IPVS proposed a dynamic traffic planning algorithm based on a conflict graph that represents possible paths and start times of flows, allowing to generate schedules by solving the independent colorful set problem with a greedy algorithm based on custom heuristics. They consider many features that complicate the basic scheduling problem and are therefore left out in many similar works, including wrap-around hyper-cycles, different-sized flows and non-static flow paths.

All this is possible using their data structure combined with a few basic operations and flags: Each possible flow configuration—a tuple containing the flow ID, a route and the discrete start time relative to the cycle—is represented as a vertex in their conflict graph. An edge is added between any two vertices if the configurations would conflict, i.e. placing both of them in the schedule would lead to queuing or interfering transmissions. Using a stateful generator, multiple configurations per flow are added to the graph, iterating over possible paths and phases, i.e. start times within the cycle. The generator is based on paths retrieved from Yen's k-shortest path algorithm [Yen71] and shifting the start time by $\Delta\phi$ each time all k paths were considered. The number of configurations added per flow is configurable, which leads to a trade-off between solution quality and time as well as space complexity. To consider the wrap-around of flows, two hyper cycles are looked at when deciding whether two configurations are conflicting.

In general, a valid schedule for a set of flows is represented by an independent set that contains a configuration vertex for each flow with no edge between any of them. Switching to a new schedule version is assumed to be possible by real-time control channels and synchronized clocks. By allowing wrap-around of flows, not only do the flows within the current configuration have to be conflict-free, but also any new flow must be checked against interference with frames from the previous schedule version that are still present in the network after a schedule change. In *offensive planning*, where modification of existing flows is allowed, configurations that interfere with previous versions of existing flows are locked i.e. forbidden. *Locking* a configuration is only temporary to avoid interference when installing the new schedule, and will be reverted (unlocked) in the next round. In offensive planning, locking is also used to adhere to jitter restrictions, forbidding configurations that would move the flow by a greater amount of time than the jitter limit. A similar, but slightly different operation is *pinning* a configuration, which means that all alternative configurations of a flow are removed, enforcing that a specific flow will never be moved or changed. Pinning all currently active flows leads to a *defensive planning* mechanism, forbidding any path or phase change to active flows.



(a) Execution times for 50 and 100 candidates used per flow

(b) Number of rejected flows by candidate configurations

(c) Comparison of graph approach to ILP solver for a varying amount of pinned flows.

Figure 2.6: Runtime (left), number of rejected flows (center) of the conflict graph approach [FGD+21], comparison of graph approach to ILP solver (right).

The iterative greedy scheduling algorithm presented in [FGD+21] first orders all flows by the number of remaining candidates with the total vertex degree and the flow ID as tiebreaker. To avoid removing already active flows, the set of currently active flows is processed before processing any newly requested flows. If configurations without any edges exist, i.e. no possible conflict to other flows, these are selected prior to the following algorithm. For every flow in this sorted list, the best-rated configuration is selected, using a rating function that assigns a high cost to configurations that shadow (i.e. forbid) a high percentage of the remaining candidates of other flows. If the algorithm terminates without admitting all flows, it can be rerun with a different ordering of flows: Both sets of active and new flows are split by whether they were successfully admitted. Then, a new run starts from scratch, first processing the not admitted active flows, then the admitted active flows and proceeds in the same way for the new flows. After a configurable number of reruns, the best result in terms of the number of admitted flows is used, with each active flow weighing more than all new flows combined. The algorithm first runs with all active flow alternatives locked, trying to find a defensive planning solution. Only if not all requested flows can be placed, offensive planning is used by unlocking the aforementioned configurations.

Falk et al. evaluate their algorithm in a round-based setup that selects either 500 or 800 flows as base set, then swapping 25, respectively, 50 flows in each round. The flows are randomly chosen from a Poisson distribution and have discrete, but uniformly random flow lengths between 125 B and 1,500 B within cycle times between 250 µs and 2,000 µs. They compare adding either 50 or 100 candidates in each step and evaluate both runtime and solution qualities. Scenarios with ±500 existing and 25 swapped flows are solved in under 10 seconds on average on a desktop-grade machine (8 cores, 16 GB of memory). Bigger scenarios with ±800 flows take about 100 seconds to solve as can be seen in Figure 2.6a, with higher outliers, all times including set up and initial graph generation. Adding 100 instead of 50 candidates per flow of course increases the runtime, but reduces the number of rejected flows significantly, from on average 0.84 per round to 0.23, as can be observed in Figure 2.6a and Figure 2.6b, respectively. At last, Falk et al. compared their algorithm to a state-of-the-art ILP solver, limited to 16 Threads, 256 GB RAM and a runtime of 300 s. Figure 2.6c shows whether the ILP or the aforementioned algorithm were more successful in placing the same set of flows, with a varying percentage of pinned flows. While the ILP solver achieves better results for higher (> 60%) percentages of pinned flows, the greedy algorithm beats the ILP for most scenarios with fewer pinned flows.

3 System model and problem statement

In this chapter, we introduce the system model, i.e. we describe all assumptions on the environment, including topology, networking behaviors and flow requirements. Afterwards, we present the problem statement of dynamically scheduling TSN flows in this environment.

3.1 System Model

The main component in our work is an ILP model for calculating a schedule for time-triggered flows across an Ethernet network, amended by algorithms that pre- and post-process the input and output of a ILP solver based on this model.

The ILP model we work with assumes a network topology that can be represented as an undirected graph $G(V, E)$ where nodes $n_i \in V$ represent switches and hosts, and edges $e_{i,j} = (n_i, n_j), e_{i,j} \in E, n_i, n_j \in V$ between nodes portray the full-duplex Ethernet links between them. Host nodes only have one link or edge, respectively, connecting them to a switch node. Switch nodes build the backbone of the network, interconnecting hosts and other switches such that a path exists between every two nodes in the network. Only host nodes may be the source or destination of a flow, while switches forward the flow in between. Switches may have an unrestricted amount of links connecting them to other nodes, with the stipulation that at most one link exists between any two nodes, i.e. no link aggregation is employed.

While our ILP model works with any topology adhering to the conditions in the previous paragraph, we used a topology analogous to a factory network for developing, testing and evaluating our work. We will further describe this topology in Section 5.1.

We assume that every switch in the network is of the same make and model, leading to nearly-static processing delays d^{proc} imposed by every switch on the Ethernet frames it forwards. We further consider Ethernet links of the same speed and length, leading to static propagation delays d^{prop}, and transmission delays d^{trans} that are only dependent on the frame length. Note that these are common assumptions in schedule calculation as explained in Chapter 2. Our ILP model could easily be adapted to make all three described delay values configurable per switch or link. The processing delay could even be calculated for each frame and switch combination if necessary. The forwarding scheme used in our model is the *store and forward* scheme, which fully receives a frame before parsing its headers and forwarding it to the respective target switch port. Therefore, the processing delay begins only after the full frame is propagated over the link.

Each time-sensitive flow F_i of the flow set $F = \{F_1, \ldots, F_n\}$ is modeled as one Ethernet frame sent from one source host to a destination host, transmitted through a per-flow predefined path. The flow length is configured globally, i.e. every frame sent has the same size. This part of model could also easily be extended in order to support flows of different payload sizes. Alternatively,

MTU-sized frames could be scheduled and the actual (shorter) frames padded if needed. Note that it is not possible to simply reserve the path for a full-sized frame and then send a smaller one, as the transmission delay depends on the frame size and therefore a smaller frame would arrive earlier as modeled, leading to interference or queuing delay.

The path of every flow is precomputed externally and submitted to the scheduler as input parameter. All paths are required to be free of loops and must connect source and destination. Other than that there are no restrictions. The scheduler must place all flows exactly along their paths and is therefore not allowed to use the spatial domain for scheduling. It can only move the flows temporally, i.e. it can adapt the start time of every flow which synchronously moves the flow timing across all links of the flow path. The ILP's goal is therefore to compute a start time t_i for every flow $F_i \in F$, without conflicts along any network link on its path. With the resulting start times, a global conflict-free schedule can easily be computed.

In real-world applications, the maximum tolerated end-to-end delay is one of the specification parameters of each flow. We only schedule flows in such a way that they arrive to an empty queue at every switch on their path, introducing no queuing delay whatsoever. In combination with supplied paths for every flow, we do not need to concern ourselves with end-to-end delays, because we always achieve the shortest delays possible, given the flow requirements. We therefore assume that all supplied flow paths are short enough to fulfill the respective flow's end-to-end delay requirements. As the provided paths may not be shortest paths or lie along the path that standard Ethernet forwarding would choose by learned MAC forwarding tables or determined by the Spanning Tree Protocol, we assume that switches are employed with ways to ensure explicit path routing, e.g. via VLAN tagging, SDN explicit forwarding tables or Explicit Path Control (IEEE 802.1Qca), as mentioned in [FGD+21].

Our model supports regular Unicast communication, i.e. Multicast Ethernet or other methods of multi-destination frame distribution is not supported. Unfortunately, adding a multi-destination feature is not as simple as copying the flow requirement for every destination, because this would require the source node to also send the frame multiple times. Instead, a mechanism could be devised that sends frame copies along some kind of distribution tree within the network graph, reserving the times needed for every single transmission in the global schedule.

Internally, every undirected link of the full-duplex network is treated as two directed links. Limiting the number of links between any two nodes to one therefore allows us to treat switch ports equivalently to the outgoing link connect to them. In the following, we will therefore refer to links and link occupations only, instead of egress ports.

Referring to the delay explanations in Subsection 2.1.2, a link is free of conflicts if the transmission time periods do not overlap. In the beginning of Chapter 4 we will explain the constraints we use to ensure conflict-free scheduling of flow transmissions.

We do not include best-effort or other low prioritized traffic patterns in our model, but they may easily be added to the network using other TSN queues combined with a guard band around scheduled traffic and possible even Frame Preemption [Ins16a].

The scheduler follows a supplied global cycle time t^{cycle} for all flows, i.e. no sub-cycles, hyper-cycles or base periods are employed, and every flow sends exactly one frame every cycle iteration. We also do not allow flow placements that wrap around the cycle, i.e. all transmissions must be completed with the cycle end at the latest. Because model properties such as link speed and

length, switch processing duration and frame size can all be represented as delays, the scheduler directly takes d^{trans}, d^{prop} and d^{proc} as inputs. All time variables and supplied constants are of integer value and while the model makes no assumption of a specific time unit, we interpret them as nanoseconds, because this resolution is sufficient to represent all types of delays encountered in a Gigabit-Ethernet network with fine enough granularity.

Hosts and switches must have synchronized clocks in order to guarantee the correct practical implementation of the computed schedules, i.e. opening transmission gates and transmitting frames exactly at the specified times. Time synchronization can be achieved via the PTP protocol as mentioned in the TSN background in Chapter 2. The term *synchronized time* translates to clocks running at the same speed and a common time reference at which the schedule cycle starts at every host and switch. While TSN enables switches to control their gates at specific times, standard hosts must rely on other features to achieve time-triggered transmission of frames. We assume hosts to be embedded devices with a tightly-controlled network stack, enabling them to send frames at specified times. Standard linux hosts could also be used, making use of Launch Time feature supported by the Linux kernel and some newer network interface cards.

We further treat machine failures, loss of power and physical link disturbances as out of scope, therefore experiencing no frame loss and correct obedience of all nodes to the calculated schedule.

3.2 Problem Statement

In this section, we describe the dynamic scheduling problem we try to solve in our work.

We define the dynamic ILP scheduling of TSN flows as a process with at least two steps, explained on the example of a factory network: Initially the time-sensitive network is set up with all machines and applications needed so far and an offline schedule is computed and installed for the flows needed at this point in time. We call the flows included in this initial scheduling step *initial flows*.

At a later point in time, the network administrator may want to add more hosts to the TSN network, or even just enable further applications on existing hosts, leading to more flows in the network. Therefore, a new schedule needs to be calculated in a second step, based on the existing schedule and network topology, amended by new flows or even new switches and hosts. We refer to these new flows as *additional flows*.

In this second scheduling round, the scheduler must place all initial flows already included in the existing schedule and additionally place as many of the additional flows as possible. Removing flows in the second round is as simple as removing the flow configuration and all reservations from the existing schedule and therefore will not discussed further. It is desirable to keep the cyclic start times of existing flows unchanged, because moving the start times leads to jitter that may affect running applications. Therefore, it is a goal of the scheduler to either not move existing flows at all, or move them as little as possible. We call changing the cyclic start time of already placed flows *rescheduling* and will devise strategies to select the set of flows to be rescheduled in the next chapter. Similar to [FGD+21], we call a scheduling strategy *defensive* if it may not reschedule any flows and refer to *offensive scheduling* when rescheduling is allowed.

In realistic scenarios, even real-time applications are able to deal with a small amount of jitter, but with a varying level of synchronization per application, it must be possible to configure the maximum amount of rescheduling per flow.

As soon as the new schedule is calculated, it is installed on all devices and then enabled once the current cycle is completed. Propagation of a new configuration of course amounts to a consensus problem, which is out of scope in this work as we assume reliable communication from the network controller to all devices. We assume a way to preconfigure the new schedule on all nodes in such a way that the operations can change to the new version at a defined point in time. Our restriction that all flows must arrive at their destination within the cycle guarantees that at the time the cycle ends, no frame is in flight and the network is empty of traffic, allowing an easy transition into a new schedule at the cycle boundary. Again, VLANs or adapted SDN forwarding tables can be employed to prepare switches for a new schedule before actually switching to it. The point in time when the new schedule is activated has to be communicated and coordinated, but this also amounts to the requirement of reliable communication, as all devices already have synchronized clocks as it is required for synchronized time-triggered frame transmission. We therefore strictly deal with computing a schedule of same cycle length fulfilling the requirements described above.

Dynamically calculating such a schedule poses multiple challenges, which we try to address in the following chapter. Of course, it is desirable to achieve a short runtime for the dynamic scheduling calculation, because new devices or additional flows can only be put into operation once a respective schedule is calculated and installed. Additionally, the dynamic requirement also may complicate the initial scheduling process, because the initial flows restrict the schedulability of flows added at later times. Therefore, the initial scheduling round shall optimize its schedule in a way to best accommodate new flows, which is tricky without knowing the requirements a priori.

4 Approach

In this chapter, we present our approach for dynamic scheduling of TSN flows using ILPs, in particular an ILP model considering previously scheduled flows, objectives to increase future schedulability and a heuristic-based selection algorithm to select a set of flows to be rescheduled.

4.1 Base ILP for static scheduling

First, we introduce the ILP model our work is based on, which in turn is adapted from NW-PSP [DN16]. Given the nodes on the predefined path $(n_0, n_1, \ldots, n_{k-1}, n_k)$ of a flow $f_i \in F$, we define the edges of that path as sequence $P_i = (e_{0,1}, \ldots, e_{k-1,k})$. The path P_i includes both the link from source node n_0 into the network and the link from the network to destination node n_k.

We further define the cumulated delay occurring from transmission start at n_0 until transmission start at hop k of flow f_i as:

$$D_{i,e} = k(d^{trans} + d^{prop} + d^{proc}) \quad \text{with} \quad e = (n_k, n_l) \tag{4.1}$$

Together with the flow start times t_i, t_j we can define the main *no-collision* constraint, ensuring that no two transmissions on a link overlap:

$$|(t_i + D_{i,e}) - (t_j + D_{j,e})| \geq d^{trans} \quad \forall f_i, f_j \in F, i \neq j, \forall e \in P_i \cap P_j \tag{4.2}$$

This constraint ensures that between every pair of transmission start times occurring on a link, there is at least a gap of d^{trans}, ensuring no transmission conflicts. At the same time, together with Equation 4.1, the no-wait property is fulfilled, as the transmission of all flows is timed in relation to the per-hop delay, which deliberately includes no queuing delay.

Further, we define the finishing time of a flow as $C_i = t_i + D_{i,e_{k-1,k}} + d^{trans} + d^{prop}$, i.e. the start time, the cumulated network delay up until the last link, and one last transmission and propagation delay. Note that no processing delay is included here, as a time-critical frame is treated as delivered once it fully reaches the destination NIC.

Using C_i, NW-PSP [DN16] define the so-called *flowspan* C^{max} as the maximum value of C_i across all flows f_i, inspired by the term *makespan* of NW-JSP.

$$C^{max} = \max\{C_i | f_i \in F\} \tag{4.3}$$

We refer to the objective minimizing C^{max} as *min-flowspan*.

With some variable ranges added and all constraints brought into standard form, we get the following full linear program, which we refer to as *base ILP model*:

$$\min \quad C^{max} \tag{4.4}$$

subject to

$$\forall f_i, f_j \in F, i \neq j, \forall e \in P_i \cap P_j, x_{e,i,j} \in 0,1 :$$

$$-t_i - D_{i,e} + t_j + D_{j,e} + d^{trans} \leq M x_{e,i,j} \tag{4.5}$$

$$-t_j - D_{j,e} + t_i + D_{i,e} + d^{trans} \leq M(1 - x_{e,i,j}) \tag{4.6}$$

$$\forall f_i \in F :$$

$$C_i \leq t^{cycle} \tag{4.7}$$

$$t_i \in \mathbb{N}_0 \tag{4.8}$$

Note that Equation 4.2 has been split into two equations as the absolute value function cannot be used in a standard form LP. Further, the split equation has been negated and an auxiliary binary variable $x_{e,i,j}$ together with a Big-M, e.g. $M = 2t^{cycle} + d^{trans}$ is employed to transform the disjunction to a conjunction.

We take this base ILP model and a C++ reimplementation using Gurobi from Frank Dürr as basis for our approach and extend it to support keeping old flows, rescheduling them, and/or optimizing the schedule for better schedulability of future flows. The scheduler uses straightforward file formats for flow definitions and solutions and expects delays, the transmission cycle and possibly a runtime limit as command line arguments. In particular, the flow set input is given as comma-separated-value (CSV) file containing for every flow an ID followed by the node IDs of its path. The output file lists the flow start time t_i for every flow as well as status information and resource statistics from solving the ILP. The existing implementation written by Frank Dürr already supports solving the NW-PSP as described above and in Subsection 2.2.1.

In the following sections we describe the adaptations and amendments to this base ILP model and further contributions that we implemented.

In Section 4.2 we add support for dynamic scheduling to the ILP model, i.e. combining initial flows with previously determined start times with completely new, additional flows. We supply the known feasible start times of initial flows as start values to the scheduler and narrow the allowed start time range for these initial flows in order to achieve strict bounds on their jitter in the case that initial flows are temporally moved by the solver in order to accommodate additional flows. As it may not be possible to admit all additional flows given the jitter constraints, we create a new objective that first places as many flows as possible, but at least the ones scheduled initially. The previously described `min-flowspan` objective will only be pursued after this `min-disabled-flows` objective has been optimized.

We also introduce two alternatives to the `min-flowspan` strategy: In Subsection 4.3.1 we define the `max-freeblock` objective that aims to create a large gap of free temporal space on every link in the network. This strategy may seem similar to `min-flowspan`, but `max-freeblock` creates a large gap *on every link schedule* instead of establishing a large gap *in the global schedule*.

In Subsection 4.3.2, we define our second strategy and objective called `link-desync`. It aims to maximize the distance between every pair of flows on a link, in order to create free gaps between every consecutive flow within a link schedule. This allows for placing flows early within the cycle,

while the previous two strategies mostly allow for placing flows at the cycle end, which might not suffice to traverse large-diameter network. Additionally, in offensive scheduling, where initial flows may be moved temporally to accommodate additional flows, we expect the `link-desync` strategy to have a better chance at placing additional flows while rescheduling fewer flows at the same time. For both strategies we describe example problem instances where the new strategy can still place flows while `min-flowspan` could not.

Lastly, we adapt the model to reduce the number and length of reschedulings by adding a respective objective function in Subsection 4.3.3.

Figure 4.3 on page 42 gives an overview over the choices and order of ILP objectives in our model.

4.2 Model adaptation for dynamic flow addition

First, we will describe the ILP model adaptations that make dynamic scheduling possible at all, in the sense of adding flows to an existing schedule.

As mentioned before we differentiate *initial* and *additional* flows. We refer to *initial flows* when talking about the initial set of flows F^{init} that is scheduled from scratch by an offline scheduler, having no restrictions beside the initial flows themselves. Per our system model and problem statement, we assume a set of flows—referred to as *additional flows* F^{add}—shall be added at a later point in time to receive a combined schedule containing both sets and respecting the constraints of both sets. We call the union of both initial and additional flows *combined (flow) set*:

$$F^{comb} = F^{init} \cup F^{add} \quad \text{with} \quad F^{init} \cap F^{add} = \emptyset \tag{4.9}$$

The Gurobi optimizer supports supplying *hints* or *start values* for variables contained in an MIP model [Gur21, p.723f]. The hint feature is useful to generally guide the solver in its internal heuristics and branching decisions when exploring the MIP search tree and is intended if the user knows which rough values the variables should have after optimization terminates. Start values are in turn only used to find an initial feasible solution and in contrast to hints must directly be feasible allocations to the respective variables. This may be suitable when a domain-specific external algorithm can compute an initial feasible variable allocation faster than the exploration algorithms of the general purpose optimizer Gurobi. If the provided start values are not all feasible or if some are missing, the solver still tries to complete the provided basis to a feasible solution for a short amount of time.

A simple approach for calculating a schedule for the combined flow set F^{comb} would be to parse the initial result file and provide the flow start times t_i^{init} as hints to the solver. However, when scheduling F^{comb}, other values for t_i than the provided t_i^{init} might form the optimal solution, possibly moving initial flows temporally and thereby introducing an uncontrollable amount of jitter. Therefore, we also need to supply strict lower and upper bounds per initial flow in order to limit the amount of jitter and refer to the *maximum allowed jitter of flow* f_i as s_i^{max}:

$$t_i^{init} - s_i^{max} \quad \leq \quad t_i \quad \leq \quad t_i^{init} + s_i^{max} \quad \forall f_i \in F^{init} \tag{4.10}$$

For example, a flow f_0 with $s_0^{max} = 100$, which initially started transmission at $t_0^{init} = 500$ will lead to a restriction of the flow start variable t_0 to the range $[400, 600]$ in the combined scheduling round. Note that when setting $s_i^{max} = 0$, t_i will be fixed to its provided hint start time t_i^{init}. The flow will be treated as statically placed and the solver may even remove its variables as part of the presolving process to simplify the ILP model.

Of course, the combined scheduler still needs the IDs and paths of all flows in F^{comb} as explained above. In addition to that, we introduce a *start config file* that allows to supply the maximum allowed jitter s_i^{max} and the initially determined start times t_i^{init} per flow. With this information, start time hints and jitter constraints as defined in Equation 4.10 can be added for all initial flows. Jitter constraints may be omitted by supplying $s_i^{max} = -1$. We treat the set of flows that have an entry in the start config file as the initial flow set F^{init}.

By varying s_i^{max}, the scheduler can be configured to perform defensive scheduling:

(4.11)
$$\forall f_i \in F^{init} : \quad s_i^{max} = 0$$

or offensive scheduling:

(4.12)
$$\exists f_i \in F^{init} : \quad s_i^{max} > 0$$

Disabling all jitter bounds essentially amounts to initial scheduling, with the minor difference that all flows in F^{init} get start time hints t_i^{init} supplied:

(4.13)
$$\forall f_i \in F^{init} : \quad s_i^{max} = -1$$

This described model extension alone allows both the fixed and dynamic placement of initial flows F^{init} together with additional flows F^{add}, which the scheduler may place freely. This approach may suffice if a feasible solution for this combined flow set F^{comb} exists and the scheduler is given enough runtime to find it. However, if the scheduler cannot place all flows, either because there is no feasible solution given the constraints, or the provided runtime limit does not suffice to compute a feasible solution in time, it will fail to provide any result at all. This is undesirable, because at least some additional flows may have been schedulable and the network administrator might be interested in admitting at least this subset.

Therefore, we amended the model by auxiliary *disable flow variables* $z_i \in 0, 1$ for every flow to allow the scheduler to not place some flows. We adapt the existing no-conflict constraints Equation 4.5 and Equation 4.6 to provide an "easy way out" for the scheduler, as disabling all flows with $z_i = 1 \forall f_i \in F^{comb}$ fulfills all constraints:

(4.14)
$$\forall f_i, f_j \in F^{comb}, i \neq j, \forall e \in P_i \cap P_j, z_i, z_j, x_{e,i,j} \in 0, 1 :$$

(4.15)
$$-t_i - D_{i,e} + t_j + D_{j,e} + d^{trans} - z_i M - z_j M \leq 3M x_{e,i,j}$$

(4.16)
$$-t_j - D_{j,e} + t_i + D_{i,e} + d^{trans} - z_j M - z_i M \leq 3M(1 - x_{e,i,j})$$

Therefore, if a flow cannot be placed conflict-free, the scheduler will choose to disable the flow at first, keeping the model feasible even if some flows cannot be placed. Of course, we want to place as many flows as possible and therefore define a corresponding objective:

(4.17)
$$\min \quad \sum_{f_i \in F} z_i$$

Remember that our precondition is that all flows in the initial flow set F^{init} need to be scheduled. With the objective in Equation 4.17 alone, the scheduler may choose to disable some initial flows in order to place more new flows. We simply forbid this by adding a constraint forcing all disable variables of the initial flows to zero:

(4.18)
$$z_i \leq 0 \quad \forall f_i \in F^{init}$$

Therefore, if the scheduler finds a solution, it will contain all initial flows F^{init}—possibly moved within their given jitter range $\pm s_i^{max}$—and as many additional flows from F^{add} as possible.

As we do not want to replace our existing objective from Equation 4.4 that minimizes the flowspan, we make use of the Multi-Objective feature provided by Gurobi: The Gurobi optimizer allows defining more than one objective function to be optimized [Gur21, p. 899]. Instead of specifying a single objective function and optimizing it, multiple ones can be provided, each with a *priority* and *weight*. The objectives will then be optimized hierarchically in the order given by the priorities. After all objectives of the highest priority are optimized, Gurobi will optimize the objectives on the next hierarchy level defined by the next-lower priority value. Note that when optimizing an objective, the scheduler may only consider solutions that do not degrade the objective function value of any higher-prioritized objective. If the same priority value is assigned to more than one objective, Gurobi will create a linear combination of all objectives of that priority value and solve it in one step. In this so-called *blending together of objectives*, a weight may be provided for each objective, leading to the multiplication of that linear expression by the weight. Every objective function may be provided a separate Gurobi environment, allowing to specify parameters such as the runtime limit or thread count that shall be considered for that objective only. When no objective-specific environment is provided, the values specified in the model's environment are used.

We take advantage of Gurobi's Multi-Objective feature and assign the highest priority to the `min-disabled-flows` objective introduced in Equation 4.17, followed by the lower-prioritized `min-flowspan` objective from Equation 4.4. The solver will only start working on the second `min-flowspan` objective once the `min-disabled-flows` objective is solved to optimality or the solver runs into a runtime limit provided specifically for this first objective. The solver may only choose solutions that do not increase the number of disabled flows in the `min-flowspan` optimization step, or in any further optimization steps. The second objective might not be started at all, if the solver runs into a provided global runtime limit before solving the `min-disabled-flows` objective.

Another advantage of this approach is that the second or further objectives can be left out if desired, which may be suitable for combined scheduler runs. Although solving only the first objective `min-disabled-flows` will not lead to a schedule with a minimal flowspan, it will still place as many flows as possible and terminate immediately after, reducing the runtime to admit further flows into the network to the technical minimum. Depending on the use case, this may be shortsighted though,

as placing flows suboptimally may reduce the schedulability of future flows even more, e.g. when flows are to be added to a schedule containing F^{comb} in a third scheduling round. As so often, this poses a trade-off between runtime and solution quality, but may be tuned by the administrator by setting a runtime limit for the whole execution or even for the single second objective. Alternatively, Gurobi's so-called `MIPGap` parameter [Gur21, p. 714] may be set, commanding the solver to only continue looking for solutions until the gap between the best found feasible solution and the known bound of an optimal solution is smaller than the provided `MIPGap` percentage value.

4.3 Strategies for increasing schedulability of future flows

Next, we propose two strategies in form of additional constraints and solver objectives that may lead to a better schedulability than `min-flowspan`, which aims at creating a large block of unoccupied time across the whole network, at the end of the schedule, to provide a good environment for best-effort traffic within that block. In the following subsections, we explain our two strategies `max-freeblock` and `link-desync`. By use of command line flags, either one of the three objectives—`min-flowspan`,`max-freeblock` or `link-desync`—can be selected for the second hierarchy level of the solver, disabling the other two objectives, respectively.

4.3.1 max-freeblock strategy

First, we describe our `max-freeblock` strategy, which aims to leave maximum-sized block of unoccupied (*free*) time in the schedule specific to every link. At first, this description may seem to be the same as `min-flowspan`, which moves flows to the beginning of the cycle in order to leave a large block of time in which the whole network is free of scheduled traffic, to be occupied by best-effort traffic, as described in [DN16] and Subsection 2.2.1. As the `min-flowspan` strategy reduces C_{max}, i.e. the arrival time of the last-finishing flow in the cycle, obviously the block of "free time" may not be reduced further for the links traversed by this last, critical flow. This characteristic may not hold for all links though, especially for links that are only used by a few flows. The `min-flowspan` strategy is still able to move around flows that do not affect the critical (last-finishing) flow within a margin, possibly placing it suboptimally with respect to the free block at the end of some *link* schedules, as the objective only assesses at the critical flow for its objective function value.

Therefore, we introduce variables and expressions to reduce the start times t_i of the last-finishing flow *per link instead of per global schedule*. We accomplish this by defining one variable for every link that captures the last start time on that link.

(4.19)
$$t_{i,e} = t_i + D_{i,e}$$

(4.20)
$$t_e^{max} = \max \quad \{t_{i,e} | f_i \in F, e \in P_i\}$$

Then, we add the `max-freeblock` objective that minimizes the sum of all t_e^{max} variables, creating an as-large-as-possible unoccupied block at the end of each link schedule.

(4.21)
$$\min \sum_{e \in \bigcup_{f_i \in F} P_i} t_e^{max}$$

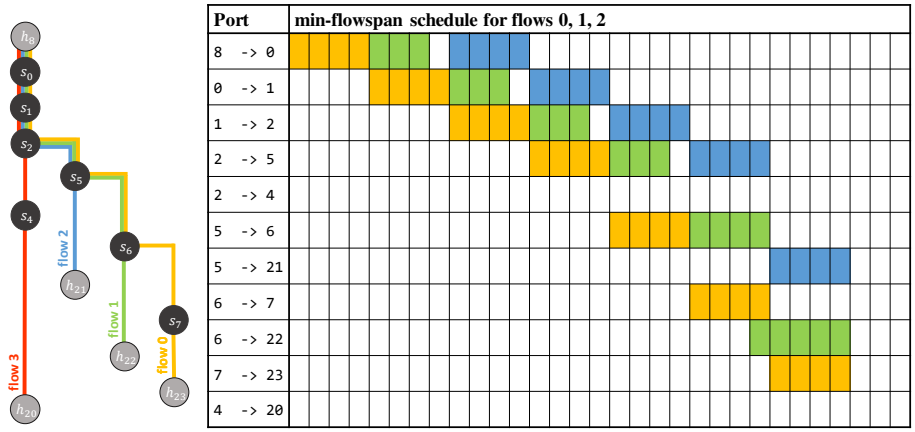
We hope to achieve a better schedulability of additional flows, especially on links that are not heavily used by initial flows, and especially in defensive scheduling. Heavily-used links will likely have a free block of similar lengths compared to a `min-flowspan` solution, as the flows on highly-occupied links have a higher probability of being critical in the sense of `min-flowspan`. We think that this strategy might perform well in defensive scheduling because a large free block on as many links as possible increases the chances of finding a placement within these time frames. In comparison, if some flows are placed in the middle of free blocks, additional flows are forced to be placed before or after the disrupting flow. If such a placement occurs on multiple links of the additional flow, and in an unsynchronized way, there might be no possible placement without rescheduling initial flows, whereas moving all flows to the beginning of their link cycle in the initial scheduling would create such a possibility without rescheduling.

Figure 4.1 shows an example scenario where the aforementioned problem occurs. Initially, three 1,200 B-sized flows are scheduled with a cycle time of 90 µs using both strategies `min-flowspan` and `max-freeblock`. In the `min-flowspan` schedule depicted in Figure 4.1a, a small gap between Flow 1 and Flow 2 is introduced, because further moving flow 2 in direction of the cycle start does not further increase the objective function value of `min-flowspan`. Now, the additional flow 3 cannot be placed in a second scheduling round, because placing it after flow 2 would be too late for it to finish within the cycle.

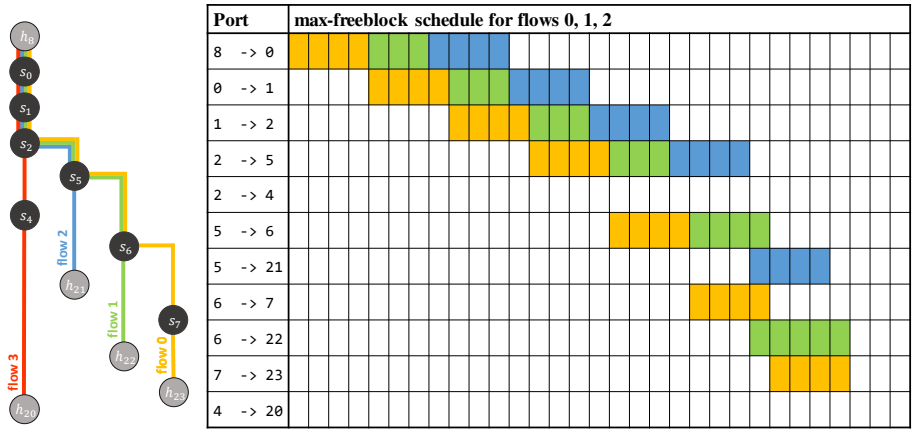
On the other hand, `max-freeblock` maximizes the free block *on each link*, removing the aforementioned gap, as can be seen in Figure 4.1b. With the gap closed, the `max-freeblock` schedule can accommodate the additional flow 3, as depicted in Figure 4.1c.

We chose to use a highly-granular slot size of a time unit, e.g. nanoseconds instead of d^{trans}-sized slots, because other delays may not be a integral multiple of d^{trans} and we wanted to allow maximum flexibility in the supplied delay lengths. Therefore, the scheduler may place flows in a way that is not aligned to d^{trans}, rendering a short gap at the beginning of a link cycle completely useless for future flow placements. Note that when scheduling flows of same payload size, a small gap at the start of a link schedule may not be an issue when an additional flow is placed from the same host, as long as the gap is at least of size d^{trans}. The gap may very well pose a problem for flows that start e.g. one hop earlier, requiring the flow to start before $t_i = 0$ in order to use a gap at time 0 on its second hop. For those flows, the first slot at the cycle start is unusable as our system model forbids cycle wrap.

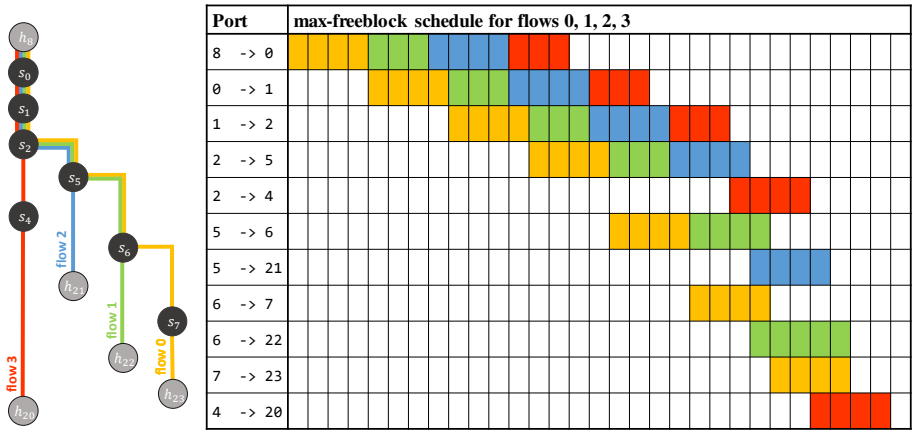
As explained in Subsection 2.1.3, ILP formulations must consist of inequalities of a specific form and natively do not support e.g. the assignment of a maximum value of a set to a variable, or even the sum of two variables to another variable. Of course, we can still force the scheduler to assign specific dynamic values to a variable, e.g. by adding more constraints to create lower and upper bounds for the target value. Workarounds for other operations such as `sum`, min, max, etc. are not that simple and straightforward anymore. Modern ILP solvers such as Gurobi support so-called *general constraints*, allowing to specify mathematical operations other than the mentioned



(a) Initial schedule resulting from the base `min-flowspan` strategy, not able to fit the red flow 3 because per-link schedules are not optimized for a large free block at the end.



(b) Initial schedule produced by our `max-freeblock` strategy, maximizing the size of free space at the end of each link schedule. Without knowing the properties of flow 3, this schedule can still accommodate once it is added.



(c) Combined schedule calculated by our `max-freeblock` strategy, fitting the three initial flows 0-2 and additionally flow 3.

Figure 4.1: Example flows (1200B payload) and 90 µs schedules produced by the base strategy `min-flowspan` and our `max-freeblock` strategy. Flow 0-2 are scheduled initially with both strategies, but only the `max-freeblock` schedule can accommodate the additional flow 3. Note that the differing number of blocks stem from rounding in the depiction algorithm.

inequalities. Internally, Gurobi will add the necessary auxiliary variables and constraints to achieve the intended higher-level computation, e.g. the assignment of a maximum value to a variable as we did in Equation 4.19 or Equation 4.21. Therefore, we no longer will transform all equations in the standard ILP form, instead we just write the intended higher-level calculations and implement them using Gurobi's general constraints.

4.3.2 link-desync strategy

Next, we describe our `link-desync` strategy, which aims to desynchronize flow embeddings within each link schedule, aiming to leave room for additional flows, especially when using offensive scheduling. By the term desynchronization we mean a temporal placement of flows that leaves an as-large-as-possible gap on link schedules between every two consecutive flows. For every pair of flows f_i, f_j that use the same link e, we create a variable capturing the distance between their start times on that specific link. As we do not know the ordering of the two flows within the link schedule, we must take the absolute value of their start time difference to get a gap size $l_{e,i,j} > 0$.

(4.22)
$$l_{e,i,j} = |t_{i,e} - t_{j,e}|$$

Additionally, we introduce variables for every flow on every link on its path, capturing the distance from its flow start time on this link $t_{i,e}$ to the cycle end time t^{cycle}:

(4.23)
$$l_{e,i} = t^{cycle} - t_{i,e}$$

For every link, we again use Gurobi's general constraints to get the minimum value of these absolute distances:

(4.24)
$$l_e^{min} = \min \quad \left\{ \bigcup_{\substack{f_i, f_j \in F, i \neq j \\ e \in P_i \cap P_j}} l_{e,i,j} \cup \bigcup_{\substack{f_i \in F \\ e \in P_i}} l_{e,i} \right\}$$

At last, we take the sum of the minimal distance l_e^{min} of each used link e and maximize it as our `link-desync` objective:

(4.25)
$$\max \sum_{e \in \bigcup_{f_i \in F} P_i} l_e^{min}$$

With this strategy, we aim to achieve a better schedulability in both defensive and offensive scheduling, as additional flows may either be placed within the created gaps on each link. Specifically, a gap of size d^{trans} must exist along an additional flow's path P_i, in order to accommodate it. If some gap is shorter than d^{trans}, it might still be possible to place the additional flow when one or both neighboring flows are moved in the temporal domain by a margin enough to increase the gap length to d_{trans}. Therefore, this strategy might be particularly useful in offensive scheduling, which allows

moving flows by as much as s_i^{max} in one direction within the temporal domain of a link schedule. Of course, as our scheduler exclusively considers no-queuing solutions, moving a flow in the temporal domain to create a greater gap to its preceding or consecutive flow on a specific link, this affects the gaps on all links the flow is transferred through. With the same reasoning, moving a flow to increase the gap on one link might not be possible if on another link of the to-be-moved flow a flow is placed right after it, or too shortly after it to move it by the desired margin.

This scenario may be remedied if all affected flows have a large enough s_i^{max} value to collectively "make room" for additional flows. As the gap constraints exist for every link, we assume that moving whole flows is possible by at least some extent, as long as no link of the to-be-moved flows is fully occupied.

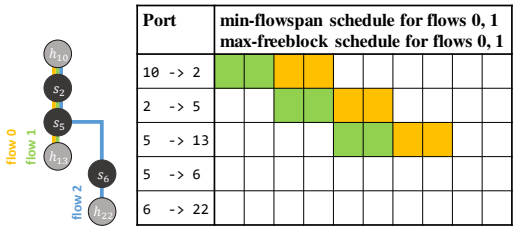
With a rising number of flows in a schedule, there will be a tipping point where on congested links, many gaps will be shorter than d^{trans}, leaving those links unable to accommodate any more flows without offensive scheduling. Therefore, in defensive scheduling scenarios with a high number of flows, the original `min-flowspan` strategy or our first proposed `max-freeblock` strategy might be bettersuited. The two aforementioned strategies would have a higher chance to place additional flows in the described scenario, as they optimize for a large free gap at the end of the link schedule or global schedule, leaving a lot more gaps of size d^{trans} on every link.

On the other hand, there may even be additional flows that would be rejected by the first two strategy whereas the `link-desync` strategy could still accommodate them: Consider a network with a medium-sized or large diameter and a relatively short cycle time. If the first 50% of the schedule are occupied by existing flows in a back-to-back manner, a new flow that has its source in the same area as this congestion, it might not be admitted to the schedule. That is because the first free gap for this flow would be in the middle of the cycle, after a long block of scheduled traffic containing initial flows, leaving only half of the cycle to traverse the network, which might not be enough to cross all hops. As wrap-around of flows across cycle boundaries is not possible in our system model, this flow cannot be placed in this scenario. In comparison, the `link-desync` approach could accommodate this flow, as it produces gaps in every temporal section of the cycle time, allowing the flow to start early within the cycle, leaving enough cycle time left to traverse all hops on its path.

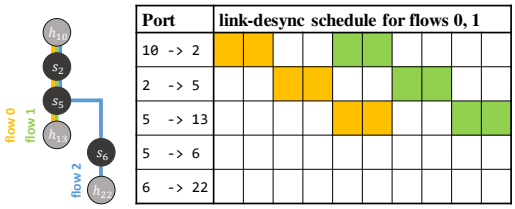
Figure 4.2 shows such an example, where the initial schedules resulting from `min-flowspan` and `max-freeblock` are not able to accommodate a third flow, whereas the third strategy `link-desync` is able to do so. In this example with 50 µs cycle time and 1,000 B flows, placing the blue flow 2 after the first two flows in Figure 4.2a is not possible, as it would not arrive within the cycle boundary. Using the `link-desync` strategy for initial scheduling, as seen in Figure 4.2b, leaves a gap large enough to fit the blue flow 2, early enough for it to finish before the cycle time ends, which is depicted in Figure 4.2c.

4.3.3 Reducing rescheduling instances

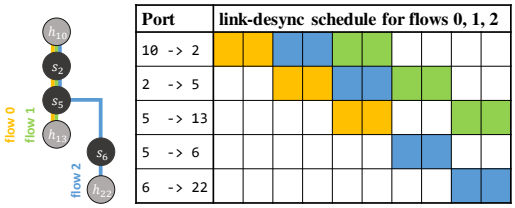
While offensive scheduling *allows* the temporal movement of initial flows by as much as s_i^{max}, it surely is desirable to keep these movements to a minimum, affecting as few flows as possible and affect them as little as possible. We therefore introduce another set of constraints and objectives



(a) Initial schedule resulting from the base `min-flowspan` strategy as well as our `max-freeblock` strategy. This schedule is not able to fit the blue flow 2, because when placing it after flow two, it would not finish within the cycle.



(b) Initial schedule produced by our `link-desync` strategy, maximizing the distance between flows placed within each link. Without knowing the properties of flow 2, this schedule can still accommodate it once it is added, as a large enough gap is left in the middle.



(c) Combined schedule calculated by our `link-desync` strategy, fitting the two initial flows 0 and 1 as well as the additional flow 3 into it.

Figure 4.2: Example flows (1000B payload) and 50 µs schedules produced by the base strategy `min-flowspan` and both our strategies `max-freeblock` and `link-desync`. Flow 0 and 1 are scheduled initially with all three strategies, but only the `link-desync` schedule can accommodate the additional flow 2.

that reduces instances of rescheduling, independently of the strategies presented before. We start by defining variables for each initial flow $f_i \in F^{init}$, capturing the amount of temporal movement:

(4.26)
$$m_i = |t_i - t_i^{init}|$$

We then define three objectives, minimizing the number of rescheduled flows, the maximum rescheduling amount and the sum of all rescheduling amounts, respectively:

(4.27)
$$\min \quad |\{f_i | f_i \in F^{init}, m_i > 0\}|$$

(4.28)
$$\min \quad \max\{m_i | f_i \in F^{init}\}$$

(4.29)
$$\min \quad \sum_{f_i \in F^{init}} m_i$$

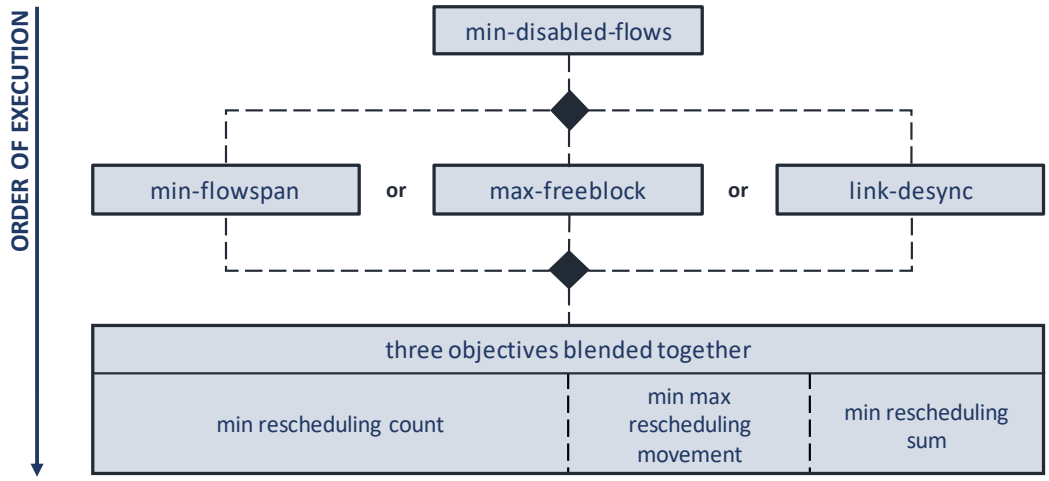


Figure 4.3: Hierarchy of scheduling objectives in our ILP model. The order of execution is from top to bottom and follows the objectives' priorities. From the second level, one or no objective may be selected. The objectives on the third level are optimized together in one step.

We again make use of Gurobi's multi-objective features, and this time create a *blended objective* for these three objective functions. Assigning the weights 2, 1 and 1 to Equation 4.27, Equation 4.28 and Equation 4.29, respectively, leads Gurobi to create a linear combination of the three, each multiplied by their weight. Therefore, we reduce the rescheduling instances in three different ways at the same time, using only one optimization stage.

4.3.4 Hierarchy of objectives

Lastly, we want to reiterate the hierarchy levels of our proposed ILP strategies or objectives, respectively. Figure 4.3 shows all supported objectives of our ILP model together with their hierarchy level, optionality, alternative choices and weights. At the highest level, the mandatory `min-disabled-flows` objective introduced in Equation 4.17 is optimized at the very first step. On the second-highest level, the main objective can be chosen mutually-exclusively from `min-flowspan`, `max-freeblock` and `link-desync`, but may be left out completely if so desired. The objectives on the third level are blended together and optimized together with respect to their weights as described in the previous section.

4.4 Selecting flows to be rescheduled

In this section, we introduce static and dynamic strategies to select the set of initial flows to be rescheduled when scheduling additional flows. As explained in Section 4.2, the scheduler may reschedule all flows that have $s_i^{max} > 0$. We call the subset of initial flows F^{init} that the scheduler may reschedule F^{res}:

(4.30)
$$F^{res} = \{f_i | f_i \in F^{init} \wedge s_i^{max} > 0\}$$

Varying the maximum allowed amount of jitter s_i^{max} not only allows us to select the set F^{res}, it also enables us to let some flows be rescheduled by a greater amount than others, i.e. we can provide a per-flow bound on the amount of rescheduling. This leaves potential for higher-level strategies as proposed in Subsection 4.4.2. All selection strategies manifest themselves as calculations executed before the additional scheduling round, utilizing results and knowledge gathered from the initial scheduling round.

Strategies varying the composition of F^{res} as well as the per-flow s_i^{max} work towards the following goals:

Admitting more additional flows Increasing the degree of temporal movement for initial flows increases the solution space of the additional scheduling round, possibly allowing to place more flows.

Limiting the rescheduling effects of initial flows As described in Section 3.2, it is desirable to keep rescheduled flows and the rescheduling amount to a minimum. Of course, we proposed ILP objectives to reduce these values in Subsection 4.3.3, but as mentioned before, the considered solutions for a lower-prioritized ILP objective must not degrade the objective function value of higher-prioritized objectives. Therefore, the rescheduling objectives are very confined in their solution space and may only result in solutions that are also optimal with respect to the chosen strategy in the hierarchy level above. Instead, if the rescheduling selection strategy imposes narrower jitter bounds or reduces F^{res} in general, the same amount of additional flows may still be schedulable, but with a lower rescheduling impact on initial flows.

Reducing the runtime to schedule additional flows Compared to defensive scheduling as defined in Equation 4.11, every increase of s_i^{max} leads to a greater solution space and thus likely also to a higher runtime. Therefore, every limitation of F^{res} or s_i^{max} reduces the runtime of the additional scheduling round.

4.4.1 Static rescheduling selection

We refer to *static rescheduling selection* when all initial flows get the same static value S assigned to s_i^{max}:

(4.31)
$$s_i^{max} = S \quad \forall f_i \in F^{init}, \quad 0 \leq S \leq t^{cycle}$$

By setting $S = 0$ we get the aforementioned *defensive scheduling*, where no initial flow may be moved at all. As soon as we choose $S > 0$, we speak of *offensive scheduling*, as this allows the scheduler to move all initial flows by a time span of up to S, as defined in Equation 4.10. Setting S to values greater than zero may be useful to provide an upper bound to the amount of jitter experienced by any flow.

Choosing a small value for S may significantly speed up placement of flows, because the smaller S is, the smaller our ILP solution space will be. Of course, if S is chosen to be too small, the scheduler may not be able to place all additional flows.

On the other hand, setting S to a large value increases the schedulability of additional flows at the cost of a greater solution space and therefore longer execution times. Additionally, a large S may move initial flows more than necessary or desired.

The rescheduling occurrences and amounts will likely increase with a higher S, regardless of our rescheduling reduction measures described in Subsection 4.3.3: That is because the scheduling reduction objectives can only act within the solution space left by the best objective value to the chosen strategy before. For example, consider that `min-flowspan` is chosen as the second-level strategy and reaches its best objective function value when moving some flows by S. Remember that lower-level objective functions may only consider solutions that do not degrade the objective function values of higher-level objectives. Therefore, the third-level rescheduling reduction objective can only reduce the rescheduling amount if it finds a solution that is equally good with respect to the objective function value for `min-flowspan`.

Setting a high value to s_i^{max} may be necessary to accommodate all additional flows, but it is undesired to do so for all initial flows. Therefore, we propose dynamic rescheduling selection strategies in the following section.

4.4.2 Dynamic rescheduling algorithm

In the following, we describe our dynamic rescheduling algorithm that aims to allow just enough rescheduling in order to place additional flows, but keeps $|F^{res}|$ as small as possible. Our algorithm performs three steps:

1. mark critical links We count the number of initial or additional flows using each link and mark links as *critical* based on a percentual threshold.

2. select rescheduling set We check how many critical links each initial flow uses and select the set of flows to be rescheduled, F^{res}, based on a static or percentual threshold thereof.

3. set rescheduling amount For each flow in F^{res}, we set either a static amount S^{res} for all s_i^{max} or scale s_i^{max} based on the number of critical links it uses. The flows that were not selected to be rescheduled get a static value S^{no-res} assigned for s_i^{max}.

The algorithm tries to increase the slack on heavily-used links in order to reschedule flows on them to make room for more flows, while leaving s_i^{max} on zero or another static value for the remaining flows. We do this on the basis that heavily-used links are likely to pose a bottleneck for placing additional flows and therefore moving them by a greater amount should increase schedulability.

The algorithm can be tuned in different places by supplying percentual or numeric thresholds, static or factor-based jitter values and by configuring the set of flows considered when marking links as critical.

In the following sections, we explain our algorithm in order of the three aforementioned steps and along the line numbers of Algorithm 1. All variables used within Algorithm 1 are also explained in Table 4.1 in order of appearance.

Marking critical links

In the first step, we mark links with the number of flows traversing them. We do this in order to get a utilization metric for every link, which we later use to determine if flows along heavily-used links are to be granted more slack in form of higher s_i^{max} values. The set of flows that are considered for counting, F^{mark}, is configurable to be the initial flows F^{init}, the additional flows F^{add}, or both. F^{mark} is initialized in lines 4 through 9 of Algorithm 1, depending on the boolean variables $MARK^{init}$ and $MARK^{add}$.

We leave the composition of F^{mark} configurable in order to support different approaches: Setting only $MARK^{init} = 1$ creates slack along the links that are already heavily-used by initial flows. Setting only $MARK^{add} = 1$ achieves the opposite in that the links that will be used by additional flows are "freed up". Setting both $MARK^{init} = 1$ and $MARK^{add} = 1$ leads to greater slack along all links that are heavily-used by the combined flow set $F^{comb} = F^{init} \cup F^{add}$ in the additional scheduling round. The actual counting is done for every used link of F^{mark} in line 12 of Algorithm 1.

Once every link is annotated with the number of F^{mark} flows that use it, we sort the links by this count value and denote the $CUTOFFPERC^{mark}$ percentile of links with the highest counts as *critical links* E^{crit} in line 13 and 14 of Algorithm 1.

Selecting the rescheduling set

Based on the set of critical links created in the previous section E^{crit}, we now pursue a similar approach to select the set of flows allowed to be rescheduled, F^{res}. In line 19 of Algorithm 1, we count the number of critical links on the path of each initial flow $f_i \in F^{init}$. Per the condition in line 18, we only include flows in F^{count} if they use at least one critical link. We only count critical links used by *initial* flows, as only those already have a feasible start time and therefore s_i^{max} values are only useful for F^{init}.

Now, we again support two modes: reschedule all initial flows above a given threshold of critical links, or reschedule the $CUTOFFPERC^{select}$ percentile of flows, sorted by number of critical links. If $CUTOFF^{select}$ is set to zero or higher, all flows using more than $CUTOFF^{select}$ critical links are added to F^{res} in lines 20 through 23 of Algorithm 1.

Otherwise, the cutoff percentile approach is pursued, similar to the cutoff process in the previous step: In line 25 and 26 of Algorithm 1, the flows F^{count} are sorted by their number of critical links and cut off after the given percentile value $CUTOFFPERC^{select}$. Only the $CUTOFFPERC^{select}$ percentile with the highest number of critical flows are added to F^{res}.

Algorithm 1: Dynamic rescheduling algorithm

Data: F^{init}, F^{add};
$S^{res}, S^{no-res} \in [0, t^{cycle}]$;
$MARK^{init}, MARK^{add}, FACTORIZE\_JITTER \in \{0, 1\}$;
$CUTOFFPERC^{mark}, CUTOFFPERC^{select} \in [0, 100]$;
$CUTOFF^{select} \in [-1, |E|]$
Result: $s_i^{max}\ \forall f_i \in F^{init}$

$E^{count} \leftarrow \{\}$; // map: link → number of flows in F^{mark} using it
$F^{mark} \leftarrow \emptyset$
$F^{res} \leftarrow \emptyset$
for $e \in \bigcup_{f_i \in F^{comb}} P_i$ **do**
  $E^{count}[e] \leftarrow 0$; // Initialize E^{count} to 0 for all used links
if $MARK^{init}$ **then**
  $F^{mark} \leftarrow F^{init}$
if $MARK^{add}$ **then**
  $F^{mark} \leftarrow F^{mark} \cup F^{add}$
for $f_i \in F^{mark}$ **do**
  for $e \in P_i$ **do**
    $E^{count}[e] \leftarrow E^{count}[e] + 1$; // Increase number of flows in F^{mark} using this link
$start\_idx^{mark} \leftarrow |E^{count}| \times \frac{100 - CUTOFFPERC^{mark}}{100}$
$E^{crit} \leftarrow sorted(E^{count})[start\_idx^{mark} :]$; // Sort marked links by count and take $CUTOFFPERC^{mark}$ percent with the highest count
$F^{count} \leftarrow \{\}$; // map: flow → number of critical links it uses
for $f_i \in F^{init}$ **do**
  $s_i^{max} \leftarrow S^{no-res}$; // Initialize s_i^{max} to non-rescheduling value S^{no-res}
  if $|P_i \cap keys(E^{crit})| > 0$ **then**
    $F^{count}[f_i] \leftarrow |P_i \cap keys(E^{crit})|$
if $CUTOFF^{select} \geq 0$ **then**
  for $f_i \in F^{init}$ **do**
    if $F^{count}[f_i] \geq CUTOFF^{select}$ **then**
      $F^{res} \leftarrow F^{res} \cup f_i$
else
  $start\_idx^{select} \leftarrow |F^{count}| \times \frac{100 - CUTOFFPERC^{select}}{100}$
  $F^{res} \leftarrow sorted(F^{count})[start\_idx^{select} :]$; // Sort flows by number of critical links and take $CUTOFFPERC^{select}$ percent with the highest count
for $f_i \in keys(F^{res})$ **do**
  if $FACTORIZE\_JITTER$ **then**
    $s_i^{max} \leftarrow F^{count}[f_i] \times S^{res}$
  else
    $s_i^{max} \leftarrow S^{res}$

Variable	Description
F^{init} (set of flows)	The set of flows scheduled in the initial round
F^{add} (set of flows)	The set of flows to be additionally scheduled in the additional scheduling round
S^{res} (integer)	Maximum allowed jitter value to assign to the flows in F^{res} (or a multiple thereof if $FACTORIZE\_JITTER = 1$)
S^{no-res} (integer)	Maximum allowed jitter value to assign to all flows in $F^{init} \setminus F^{res}$
$MARK^{init}$ (boolean)	Whether to consider the initial flows F^{init} when marking links
$MARK^{add}$ (boolean)	Whether to consider the additional flows F^{add} when marking links
$FACTORIZE\_JITTER$ (boolean)	Whether to assign S^{res} multiplied by the number of critical links a flow in F^{res} uses to s_i^{max}
$CUTOFFPERC^{mark}$ (integer)	The percentile of links ordered by number of flows in F^{mark} that shall be considered *critical*
$CUTOFFPERC^{select}$ (integer)	The percentile of initial flows ordered by number of critical links used that shall be included in F^{res}. This is only applied if $CUTOFF^{select} < 0$
$CUTOFF^{select}$ (integer)	The minimum number of critical links used by an initial flow to be included in F^{res}. To be used alternatively to $CUTOFFPERC^{select}$.
E^{count} (map: link → integer)	Map to count the number of flows from F^{mark} that use a specific link
F^{mark} (set of flows)	Set of flows to be considered for marking links, see $MARK^{init}$ and $MARK^{add}$
F^{res} (set of flows)	Set of flows that get a (higher) maximum jitter value s_i^{max} assigned
F^{comb} (set of flows)	Combination of F^{init} and F^{add}
f_i (flow)	Reference to a flow
P_i (sequence of links)	Path of flow f_i
$start\_idx^{mark}$ (integer)	Index calculated using $CUTOFFPERC^{mark}$, used to cut off E^{count} to keep only the higher-used portion in order to mark them as *critical*
E^{crit} (map: link → integer)	E^{count}, ordered ascending by flow count on the links, cut off before index $start\_idx^{mark}$.
F^{count} (map: flow → integer)	Map to assign number of critical edges used per flow
s_i^{max} (integer)	Maximum allowed jitter value of flow f_i
$start\_idx^{select}$ (integer)	Index calculated using $CUTOFFPERC^{select}$ or $CUTOFF^{select}$, used to cut off F^{count} to keep only the higher-valued portion in order to include the respective flows in F^{res}

Table 4.1: Variables used in Algorithm 1 in order of appearance

Setting the rescheduling amount

In the last step, the flows selected for rescheduling F^{res} get jitter bounds s_i^{max} assigned. In line 27 through 31 of Algorithm 1, either a static or a factorized jitter value is assigned. If the flag $FACTORIZE\_JITTER$ is set, each flow in F^{res} gets the jitter value S^{res} multiplied by the number of critical links it uses assigned. Otherwise, all flows in F^{res} get the same jitter value S^{res} assigned to their respective s_i^{max}. The remaining initial flows not contained in F^{res} all get a configurable jitter value S^{no-res} assigned to their s_i^{max} variables in line 17 of Algorithm 1. Note that values > 0 are also allowed for S^{no-res}, allowing to reschedule the not-so-critical flows by at least some smaller value.

5 Evaluation

In this chapter, we present the evaluation of our contributions explained in Chapter 4. We begin by describing the setup, which includes the general procedure, scenarios and the hardware specifications of the server we used to calculate schedules.

5.1 Setup

In this section, we describe the setup of our evaluations including the procedure utilized as well as the network topology, delays and flow configurations used.

5.1.1 Network and flow properties

We chose the network topology depicted in Figure 5.1 as a basis for our evaluations. It consists of nine sub-topologies (ring or line) and is inspired by networks commonly found on production lines in factories. The central redundant sub-topology is a ring displayed in the center of Figure 5.1. Attached to the central ring are further, smaller, ring topologies which in turn may have further line or ring topologies attached to them. As a general rule, we only connected hosts to switches that do not interconnect different sub-topologies in order to use these switches only to forward traffic between the respective sub-topologies. In total, our topology consists of 37 switches and 24 hosts, connected by Full-Duplex Gigabit Ethernet links.

Based on this topology, we generated *feasible flow sets* using a generator developed prior to this work by the author and Patrick Schneefuss in [SWHD20]. The generator creates a set of random flow specifications that are guaranteed to be schedulable given a topology and a schedule length. It accomplishes this task by randomly placing no-queueing flows in free temporal sections of a schedule and returns the flow specifications as soon as no more flows can be placed. At first, the generator tries to place flows with source and destination nodes from different sub-topologies, ensuring that the central ring topology has a high utilization, creating a fairly complicated scheduling problem in this network section. Only after no more flows can be placed in this manner are flows with source and destination nodes from the same segment generated. The flow specifications are returned in randomized order, ensuring that the different kinds of flows—placed within sub-topologies or across sub-topologies—are distributed uniformly within the flow set.

We chose a cycle length of $t^{cycle} = 100\,\mu s$ and generated three flow sets with 600 random feasible no-queuing flows each. We selected the smallest payload size the generator supports, leading to frames of 88 B length, including all necessary headers. As Gigabit Ethernet specifies an inter-frame gap (IFG) of at least 12 B, we added this amount to the frame length and therefore configured a transmission delay of $d^{trans} = 800\,ns$, based on the Gigabit Ethernet transmission speed of 1,000,000 bit/s. We selected this relatively small frame length to in order to get as many feasible

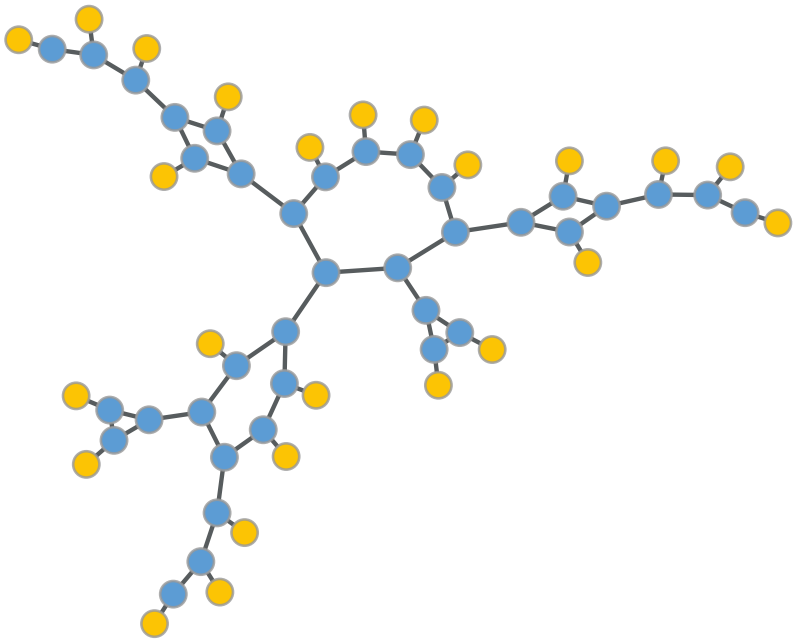


Figure 5.1: The *factory backbone* network topology used in our evaluations. Blue nodes represent the 37 switches while the 24 hosts are depicted in yellow.

flows as possible, creating a hard-to-compute scheduling problem that allows us to better examine performance and quality differences of the different scheduling strategies proposed. We configured a propagation delay of $d^{prop} = 200\,\text{ns}$, corresponding to 40 m of copper links [20b]. The processing delay is static across all switches and chosen to be $d^{proc} = 2{,}000\,\text{ns}$ in accordance with the processing delays experienced in state-of-the-art switches.

5.1.2 Evaluation pipeline

In this section we describe the pipeline and hardware specifications employed to evaluate our contributions. Recall that per our problem statement, we want to initially calculate an offline schedule admitting all flows provided to the scheduler. At a later point in time, the schedule is to be amended by additional flows, keeping all initial flows and admitting as many additional flows as possible. This additional scheduling process preferably terminates after a short time and reschedules as few initial flows as possible.

We had a server with dual AMD EPYC 7451 24-Core processors and 512 GB of memory at our disposal, which allows parallel computation on a total of 48 cores with 96 threads. As not everyone might have access to such a powerful server, we partitioned CPUs and memory to test our scheduler on more realistic resource restrictions. For this, we used the `cpuset` and `memory` options of Docker in order to create two types of resource-constrained virtualized containers: For initial scheduling, we chose to assign eight CPU cores and their corresponding eight hyper-threads as well as 80 GB of memory, representing the performance of a small server or a current workstation. We chose this partition size as we assume that at least in the initial setup procedure of a TSN network, such a machine will be available to calculate the TSN schedule.

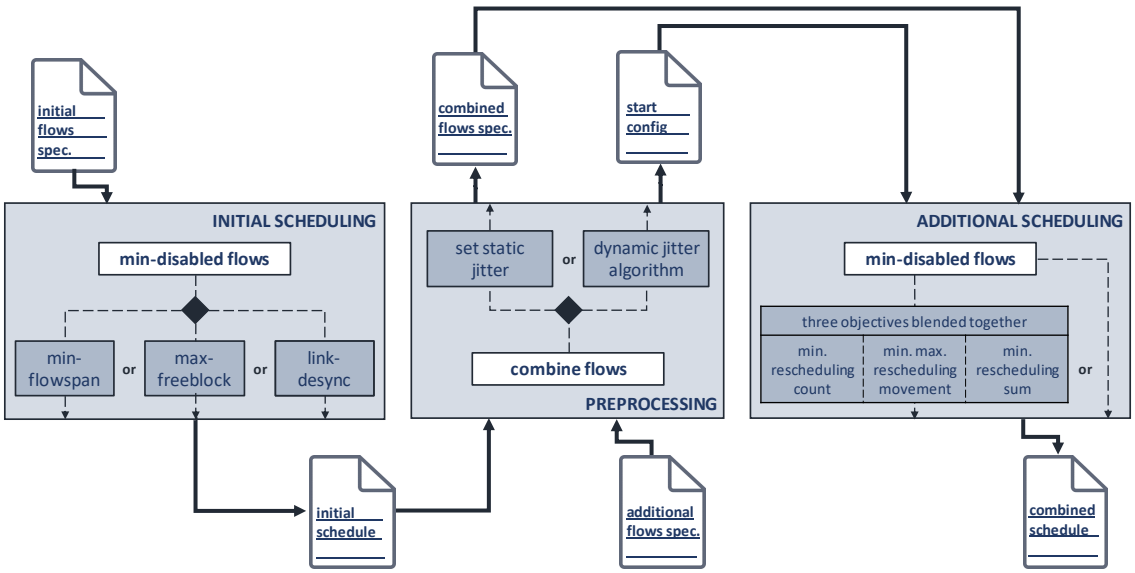


Figure 5.2: Pipeline of our evaluation. In the initial scheduling step, a subset of our flows are scheduled. The resulting schedule is then combined with additional flows and, together with start times and jitter limits, scheduled in the additional scheduling round, resulting in a combined schedule.

For additional scheduling, we opted for even more constrained partitions of two CPU cores, the corresponding two hyper threads and 20 GB of memory, representing a small workstation or even a powerful notebook. We think that at the time additional flows are to be scheduled, a powerful server may not be available and therefore opted to benchmark this process on this rather low-range performance configuration.

We now describe the evaluation pipeline used to inspect and evaluate our proposed scheduling strategies. Depicted in Figure 5.2 are the three basic steps from left to right: initial scheduling, preprocessing and additional scheduling.

We begin by taking a subsequence of the input flow set, e.g. the first 100 flows of a 600 flow set, and supply it to our scheduler in order to calculate an initial schedule on 16 CPU threads and up to 80 GB of memory. This initial scheduling step is pictured on the left of Figure 5.2. As described before, the first ILP objective executed is `min-disabled-flows`, which tries to find a feasible start time for all flows without any further intention. We initialize our ILP solver—Gurobi[1] (version 9.1.2)—with only the variables and constraints necessary for this objective, and solve the objective to optimality without any runtime limit, ensuring that all flows in the input set get scheduled. We do this in order to allow each strategy the same amount of time to perform their objective and not waste their limited execution time on placing flows, which is the same task regardless of the strategy.

Next, we chose one of the three described scheduling strategies `min-flowspan`, `max-freeblock` and `link-desync`, add the necessary ILP variables and constraints and optimize it. For this objective we impose a runtime limit of either one or two hours, which we will highlight later in this chapter. This was necessary as we experienced execution times in the order of days to solve some of the strategies

[1] https://www.gurobi.com/products/gurobi-optimizer/

to completion. As we wanted to examine many different scenarios in our limited time frame, we chose to impose this limit. Of course, we also don't want to leave TSN network administrators with an approach requiring exorbitant execution times. We applied the runtime limit only to the actual strategies to allow a fair comparison, i.e. every strategy starts its computation time with a feasible solution for all flows.

We executed this initial scheduling step with 100 up to 600 flows, from three feasible flow sets, in steps of 100 flows. We kept all of these schedules in order to evaluate different configurations for additional scheduling on the same input data.

In preprocessing, depicted in the center of Figure 5.2, we take the initial flow set and combine it with 100 up to 500 additional flows of the same feasible flow set (in steps of 100 flows) in order to perform additional scheduling afterwards. Based on the flow start times in the initial schedule, we also create the *start config file* described in Section 4.2. The start config file contains the start time t_i^{init}, the allowed amount of jitter s_i^{max} and the initialization mode (*hint* or *start value*) of the start time variable for every initial flow. Based on the evaluation configuration, we choose one of the initialization modes for all flows and either set a static jitter value S for all flows, or execute our *dynamic rescheduling algorithm* as described in Subsection 4.4.2.

At last, we perform the additional scheduling round—displayed on the right side of Figure 5.2—on four CPU threads and up to 20GB of memory. Recall that this objective keeps all initial flows scheduled and tries to admit as many additional flows as possible. Again, the first objective is `min-disabled-flows`, but this time with a runtime limit between 60 s and 2 h. We set this time limit because we assume that in realistic scenarios, the inclusion of new flows into the schedule must be finished in a rather short time frame. After placing as many flows as possible, optionally the amount of rescheduling is reduced, using the three strategies proposed in Subsection 4.3.3 blended together into one optimization step, also with a time limit. As a result of this additional scheduling round, we receive the start times for the combined flow set, in which at least the initial flows are placed.

We collected all execution logs, results and interim calculations in order to analyze and evaluate the different approaches and configurations, as presented in the next section.

5.2 Results

In this section we present the results of our evaluation.

5.2.1 Runtime of the base scheduling problem

We start by giving an overview over the runtime and ILP complexity of the scheduling problem of placing flows without further strategies, i.e. our objective `min-disabled-flows` introduced in Section 4.2.

Figure 5.3 shows the average runtime in seconds with 95 % confidence intervals for a rising number of flows displayed on the x-axis. Note that the runtime on the y-axis is plotted with a logarithmic scale. We can see that with a linearly rising number of flows to place, the runtime increases in a quadratic fashion (recognizable by the parabolic growth within this log-linear plot).

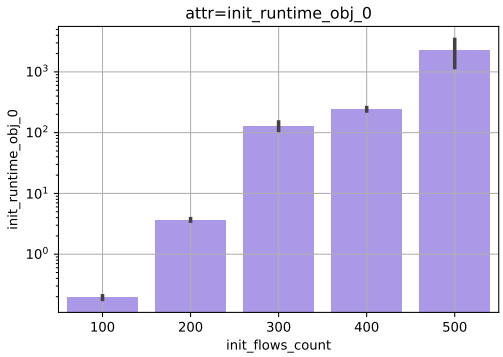


Figure 5.3: Runtime to place an increasing number of flows without further strategy (objective `min-disabled-flows`, log-linear scale, 95 % confidence intervals).

It is not surprising that the runtime grows quadratically, as we create LP constraints for every pair of flows traversing the same link (c.f. Section 4.1), which is a quadratic amount to the base of the flow count, per link. Placing 500 feasible 100 B flows within the 100 µs cycle took between twenty minutes and one hour on the described 16-core machine.

5.2.2 Complexity of the base scheduling problem

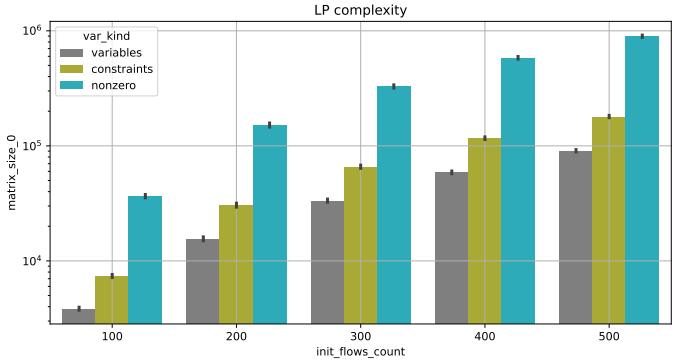


Figure 5.4: LP complexity in terms of the number of variables, constraints and nonzero entries in the matrix for an increasing number of flows (log-linear scale, 95 % confidence intervals).

We now present the LP complexity of the initial scheduling problem of only placing flows feasibly without further strategy. Figure 5.4 show the logarithmic number of LP variables, constraints and nonzero entries within the LP matrix for a linearly rising number of initial flows. The underlying ILP model only contains the necessary variables and constraints to place all flows without conflict, but no elements for further optimization strategies. We again see a parabolic growth for every metric displayed within this log-linear plot, which we interpret as a quadratic growth of the LP complexity, depending on the linear growth of the input flow set.

As mentioned before, we create a quadratic amount of LP constraints, which explains the quadratic growth of constraints and non-zero LP matrix entries in Figure 5.4. It might at first be confusing that the number of variables also increases in a quadratic fashion for each linear increment of the flow number. This can be explained by binary helper variables, which we need and add in every *no-conflict* constraint like in Equation 4.5 within Section 4.1.

The ILP instance to schedule 500 flows without conflict contains just under 100,000 variables, nearly 200,000 constraints and more than 900,000 nonzero entries in the LP matrix.

5.2.3 Complexity of our strategies

Now, we want to examine the ILP model complexity inflicted by the three scheduling strategies which we presented in Section 4.3. Recall that for optimization of the strategies, the variables and constraints from the first objective, `min-diabled-flows`, are kept and amended by additional constraints specific to the strategies. Therefore, the complexity metrics discussed in this section are influenced by both the first objective `min-disabled-flows` and the respective scheduling strategy.

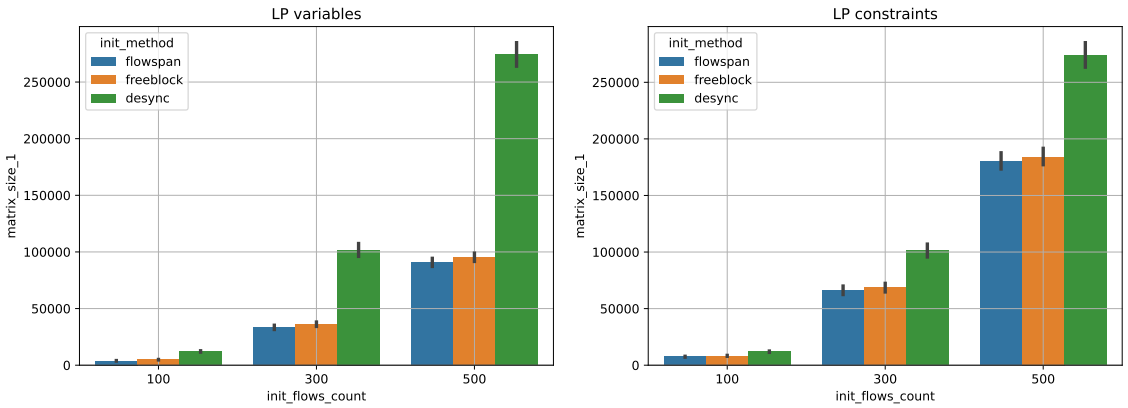


Figure 5.5: LP complexity of our three strategy ILP models in terms of the number of variables and constraints, plotted side-by-side for an increasing number of flows, distincted by the strategy chosen (log-linear scale, 95 % confidence intervals).

Figure 5.5 again plots the LP complexity on the y-axis with an logarithmic scale, for a linear increment of the input flow set size on the x-axis, with a strategy distinction within each plot. We created separate plots for the previously explained number of LP variables and LP constraints.

At first glance, we already see that the complexity metrics of the first two strategies `min-flowspan` and `max-freeblock` are always in vicinity of each other. The third objective, `link-desync`, has the highest complexity in every metric, sometimes even double the amount of the first two strategies.

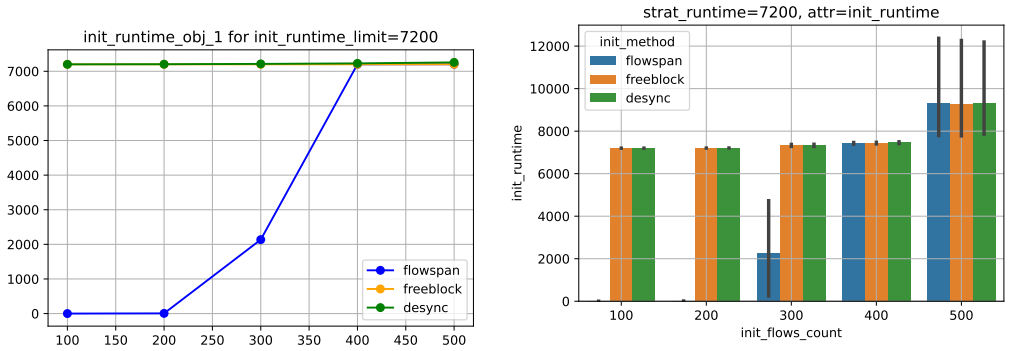
Knowing that the simplest `min-flowspan` objective depends only on the start times and static delays, the strategy adds only a linear amount of constraints (to the number of flows) and therefore is not significantly more complex than the base scheduling problem discussed in the previous section.

The second strategy, `max-freeblock`, adds a linear amount of variables and constraints (distance to cycle end for each flow on each link) and is therefore only slightly more complex than the already quadratic base scheduling model.

The third strategy is the most complicated of all, as it adds a quadratic amount of variables and constraints per flow and link, which leads to a significant increase in both metrics.

To schedule and optimize 500 flows, the first two strategies `min-flowspan` and `max-freeblock` both create a model with just under 100,000 variables and just under 200,000 constraints, similar to the base model. In comparison, our `link-desync` strategy uses almost 300,000 variables and constraints for the same amount of flows, highlighting the complex nature of it.

5.2.4 Runtime of our strategies



(a) Mean runtime to solve the objective of our three strategies for an increasing number of flows with a runtime limit of 7,200 s.

(b) Total runtime for initial scheduling with our three strategies, for an increasing number of flows and a strategy objective runtime limit of 7,200 s.

Figure 5.6: Mean runtime for initial scheduling

We now look into the runtimes of our three scheduling strategies `min-flowspan`, `max-freeblock` and `link-desync`.

In Figure 5.6a, we plotted the mean execution time necessary to execute and optimize the three objective functions of the three presented strategies `min-flowspan`, `max-freeblock` and `link-desync` for an increasing number of flows. Note that these runtime values only include the optimization of the strategies, but not the placement of flows, which we described before. Also, we imposed a runtime limit of 7,200 seconds (2 h) for the strategy optimizations.

We can see that nearly all executions use the full runtime of 7,200 s, meaning that in these cases the objective was not solved to optimality. The `min-flowspan` strategy poses an exception to this and is able to calculate an optimal solution within the mentioned time limit for up to 300 flows, but also runs into the time limit for greater amounts. Both `max-freeblock` and `link-desync` always needed the full runtime limit, which means they were not able to calculate an optimal solution for any of the evaluated flow subsets in that time. We will further look into the optimality of our strategies in the next section.

In Figure 5.6b, we displayed the total runtime necessary for initial scheduling with our three strategies and a strategy runtime limit of 7,200 seconds (2 h). The plot is simply the combination of the execution times in Figure 5.3 and Figure 5.6a, but we included it nonetheless to present the time necessary for initial scheduling at a glance.

5.2.5 Optimality of our strategies

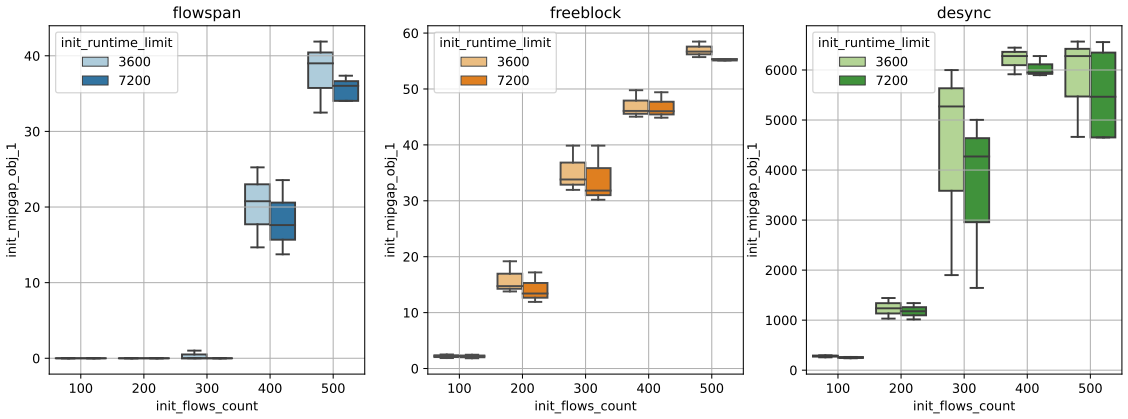


Figure 5.7: MIP gaps of our strategy objective functions for an increasing number of flows with a varying strategy runtime limit of one and two hours, respectively (95 percent confidence intervals). From left to right: `min-flowspan`, `max-freeblock`, `link-desync`

We now look into the *MIP gap* of the three strategies for a varying runtime limit. Recall that the MIP gap is the percentual difference between the current solution and the best known solution bound with respect to the objective function value, i.e. the percentage value states how far from optimality the current solution is. Figure 5.7 shows the MIP gap for varying time-constraints and an increasing number of flows to be scheduled. We created a separate plot for each of the three strategies `min-flowspan`, `max-freeblock` and `link-desync`. From left to right, we now describe and interpret the effect of an increased runtime limit on the MIP gaps of our strategy objective functions.

Starting with the left plot in Figure 5.7, we look at the MIP gaps of the `min-flowspan` strategy. This strategy is the simplest one, as its objective function value depends only on the start time variables of every flow, which only increases linearly with an increased number of flows to be scheduled. We again see that this strategy could solve problem sets with up to 300 flows to optimality within our time limit of two hours. For greater problem instances, we can see that doubling the time limit from one to two hours significantly reduces the MIP gap, as can be seen by the non-overlapping box plots at $x = 400$ and $x = 500$ flows, leading to solutions with a MIP gap of at most 40 %.

In the center plot of Figure 5.7, we can observe the 95 % confidence intervals for the MIP gap of our `max-freeblock` strategy. We recall that the objective function value of `max-freeblock` depends on a quadratic number of variables, namely the flow start time for each flow on each link. The strategy achieves an MIP gap in vicinity to the MIP gap of `min-flowspan`, but does not achieve optimality for any of the flow subsets. We can observe that the MIP gap worsens in a linear fashion for every linear increment of the input flow set size. Although the doubled runtime (from one to two hours) slightly decreases the MIP gap, the improvement is almost never significant.

The MIP gap of the last and most complicated `link-desync` strategy is plotted on the right side of Figure 5.7 and is orders of magnitude greater than the MIP gaps of the previous two strategies. Even for the smallest flow set of size $x = 100$, `link-desync` only achieves a MIP gap of over 240 %, while the largest instances of $x = 500$ flows lead to a MIP gap of over 6,000 %. For this strategy, the

MIP gap even increases polynomially for a rising number of flows in the input flow set. This is in accordance with the complexity of the objective function, which depends on a polynomial amount of variables, namely the relation of every flow start time to every other flow start time on every link, as described in Subsection 4.3.2. Except for the smallest problem instance $x = 100$, we can observe no significant improvement of the `link-desync` MIP gap after doubling the runtime from one to two hours.

We could see that for all three strategies, the MIP gap after a limited execution time directly relates to the ILP complexity described in the previous subsection.

5.2.6 Gap characteristic of the strategies

We now analyze the characteristics in terms of flow placement experienced in initial scheduling with the three strategies `min-flowspan`, `max-freeblock` and `link-desync`. First, we explain the metric used in this section: The *Inter-Frame Gap* (IFG) denotes the timespan between two transmissions on an Ethernet link, i.e. the period of time where the link is unoccupied after one frame has been fully transmitted and before transmission of the next frame begins. For examples of free gaps we refer to the white boxes within examples presented in Figure 4.1 and Figure 4.2 in Subsection 4.3.1 and Subsection 4.3.2, respectively.

We recall that the `min-flowspan` strategy minimizes the last-finishing flow arrival time, i.e. it maximizes the smallest occuring gap length at the cycle. Similarly, the `max-freeblock` strategy maximizes the length of the free block at the end of the cycle *for every link schedule*. The third strategy `link-desync` aims to maximize the gap length between every two consecutive flow placements on every link.

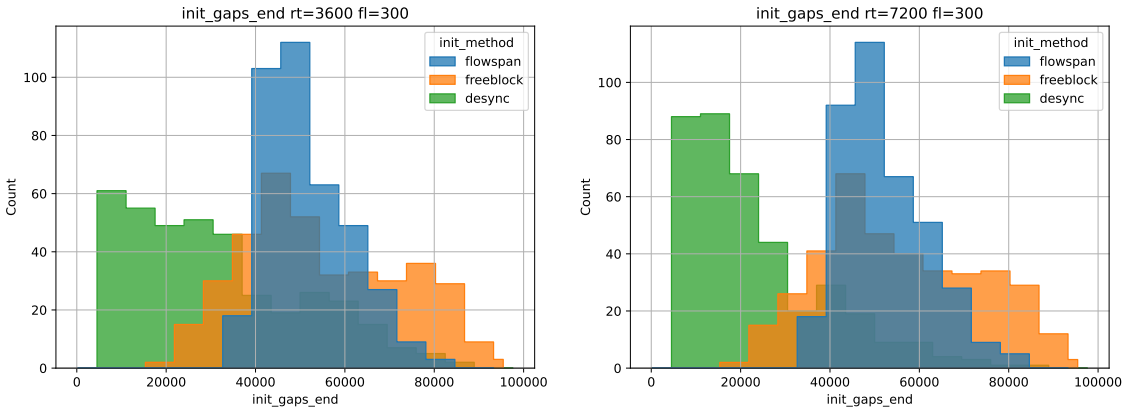


Figure 5.8: Distribution of the Inter-Frame gap lengths at the cycle end, resulting from initial scheduling with the three strategies `min-flowspan`, `max-freeblock` and `link-desync`. The x-axis indicates the length of the gaps while the y-axis of this histogram shows the number of occurrences of gaps of this length, color-distinguished for the different strategies. Both plots consider the last gap within the cycle of each link and are based on a flow count of 300 with a varying initial strategy runtime of one and two hours, respectively.

Now, with this in mind, we present the distribution of free gap lengths depicted in Figure 5.8. Both plots are histograms over the end gap lengths resulting from applying the three strategies. On the x-axis, we denote the end gap size, which is in the range of $[0, t^{cycle} = 100{,}000\,\text{ns}]$. On the y-axis, we count the number of free gaps of the given sizes for each of the three strategies. Note that we only counted the free gaps on links that are occupied by at least one flow, avoiding skew into the direction of $t^{cycle} = 100{,}000\,\text{ns}$.

Both plots in Figure 5.8 compare the length distributions of *the last gap within the cycle* when scheduling 300 flows with a varying runtime of 3,600 s and 7,200 s, respectively. We call the last free gap within the cycle the *end gap*.

Looking at the left plot in Figure 5.8, we can see that the end gaps of the `min-flowspan` strategy are concentrated in the center of the available range, most having a length between 40 µs and 60 µs. We also observe that there is a sharp cutoff on the left side and no end gaps shorter than 30 µs exist. This observation is in line with the objective of this strategy, that is, increasing the shortest occurring end gap on any of the occupied links. When comparing the blue, `min-flowspan` area in the left and right plot of Figure 5.8, we can see that there has been a slight shift to the right. For example, in the right plot, the higher runtime (2 h instead of 1 h) leads to more end gaps of length 50 µs and fewer of length 40 µs. This phenomenon is in line with the objective and emphasizes the relation of a higher strategy runtime to a shorter flowspan, i.e. to longer end gaps.

The similar strategy `max-freeblock` has a more even distribution of end gap sizes in the left plot of Figure 5.8, ranging from 20 µs to 90 µs. Although there are some instances of short end gaps between 20 µs and 40 µs, the strategy leads to a greater amount of long end gaps, as can be seen in the right area of the left plot. Again, we compare the described `max-freeblock` end gap length distribution to the same problem sets, but with a higher runtime of 2 h, i.e. the orange area in both plots of Figure 5.8. We again notice a slight skew to the right, i.e. with longer strategy runtime for `max-freeblock`, more large end gaps occur. This can especially be observed when comparing the right areas on the right side of the plots, between the lengths of 50 µs and 90 µs. The behavior again is in line with the objective of this strategy, which maximizes the sum of all end gaps.

The end gap length distribution of the third strategy, `link-desync`, deviates greatly from the first two approaches. In the left plot of Figure 5.8, we can observe that the `link-desync` strategy—depicted in green—leads to many short end gaps with lengths between 5 µs and 40 µs, and almost no end gaps longer than 60 µs. Considering the second plot in Figure 5.8, the distribution is even more skewed to the left side, with most end gap lengths ranging from 5 µs to 30 µs, and almost doubling the amount of end gaps of these lengths. This is not surprising, as the strategy aims to evenly distribute flow placements within the temporal domain of every link schedule. Naturally, this leads to shorter end gaps, as flows are pushed to the end of the cycle to increase the gap to their predecessors on each link.

Next, we look at the plot in Figure 5.9, which shows the distribution of *all* free gaps present in the global schedule, when scheduling 100 flows with a strategy runtime of 7,200 s. This means that also the gaps at the beginning of each link cycle and between consecutive flows are included, but we reiterate that unused links are excluded from this plot. Clearly, most gaps within all cycles are relatively short with lengths between 0 µs and 20 µs. This is especially true for the first two strategies, `min-flowspan` and `max-freeblock`, as both produce the largest amount of all gaps in the length range [0 µs, 10 µs]. This can easily be explained by the respective strategy of both approaches. `min-flowspan` pushes flow placements together and to the beginning of the cycle in order to reduce

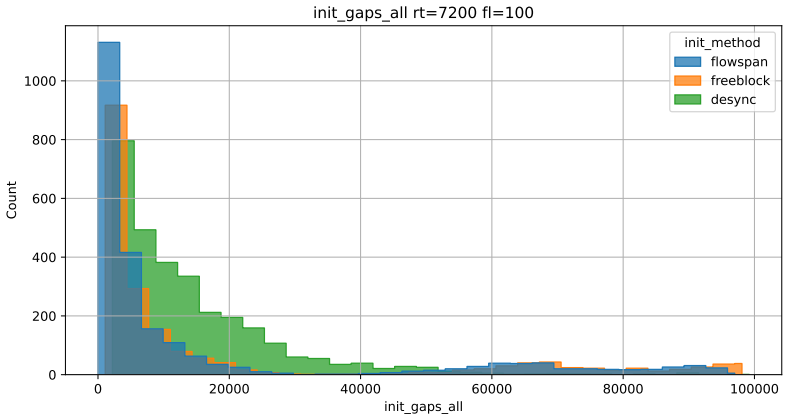


Figure 5.9: Distribution of all Inter-Frame gap lengths, resulting from initial scheduling with the three strategies `min-flowspan`, `max-freeblock` and `link-desync`. The x-axis indicates the length of the gaps while the y-axis of this histogram shows the number of occurrences of gaps of this length, color-distinguished for the different strategies. This plot considers all gaps and results of scheduling 100 flows with a strategy runtime limit of two hours.

the finishing time of the critical flow, which creates blocks of back-to-back scheduled flows and leads to a large amount of short gaps, most having zero length. The previously examined end gaps can be found as small elevations on the right side of Figure 5.9. `max-freeblock` works very similar to `min-flowspan` and leads to gap lengths in the same vicinity accordingly. In comparison to `min-flowspan`, the `max-freeblock` area is slightly shifted to the right side of the plot, meaning there are no gaps of zero length and marginally longer gaps. The increments in the right area of the plot obviously stem from the longer end gaps described before. We explain the shift in the left side with our assumption that the `max-freeblock` strategy only pushes flows so far to the beginning of the cycle that the end gap increases, but has no benefit in pushing flows so far together that the gaps in the beginning and in between flows get zero size.

The `link-desync` strategy again diverges greatly from the other two strategies in Figure 5.9, as its histogram is even shorter and more stretched to the right side, with all gaps having lengths between 5 µs and 50 µs. This observation is in accordance with the overall strategy that creates maximum-sized gaps between every two consecutive flows, leading to more gaps of higher lengths. As described in Subsection 5.2.5, the `link-desync` strategy could only be solved with a very large MIP gap in the time limit of two hours, i.e. the solutions discussed here are by far not optimal. We think that the shape of the green `link-desync` area in Figure 5.9 is a result of an initial scheduling solution that contains many short gaps, which the strategy optimized subsequently, leading to some greater gaps in the range of $[10\,\mu s, 30\,\mu s]$. Because of an intended even distribution of flows across the temporal domain of each link schedule, no very large gaps result from this strategy, and the few gaps of almost 50 µs length likely result from links that are only occupied by a single flow right in the middle.

5.2.7 Hints vs. start values

In this section, we compare the impact of using hints or start values when solving the ILP model of the additional scheduling round. As explained in Section 4.2, Gurobi supports two ways to guide the solving process based on external input: hints and start values. Hints are used both to aid feasible solutions and to guide the heuristics and branching decisions in the solving process. Start values are directly used in the start vector to find the initial feasible solution and must form a feasible solution vector. Although to some degree, variables can be left out to be populated by the solver.

We recall that the additional scheduling round starts with a number of already placed (initial) flows, for which start times are known, and attempts to place as many new flows as possible. We provide the start times of the initial flows to the scheduler as hints or start values and in the following evaluate the results thereof.

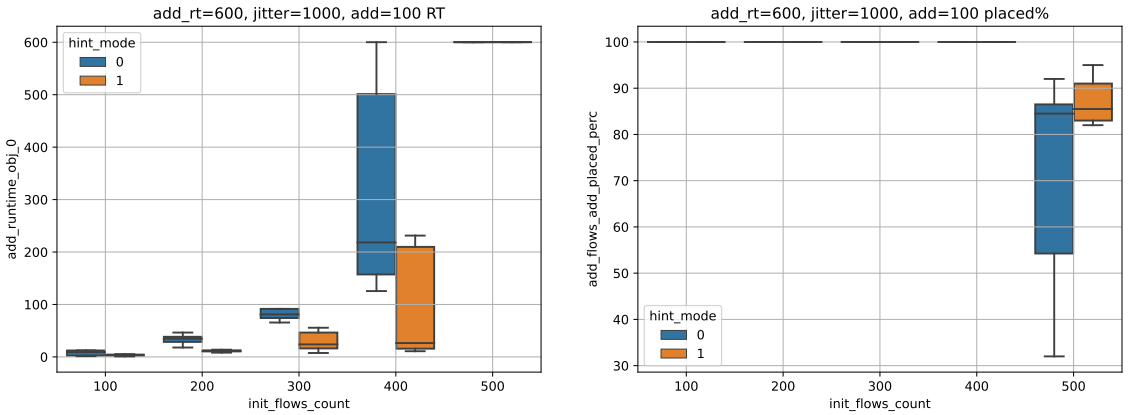


Figure 5.10: Runtime and solution quality comparison between solver hints and start values. For an increasing amount of initial flows, 100 additional flows were scheduled with a runtime limit of 600 s, allowing up to $S = 1{,}000$ ns of jitter. Left: runtime of this additional scheduling round, right: percentage of additional flows placed in the additional scheduling round.

Figure 5.10 shows the runtime (left) and success rate (right) of placing additional flows with start values in comparison to hints. Both plots are based on the same evaluations where 100 flows were added to an increasing number of initial flows (drawn on the x-axis), with a runtime restriction of 600 s and a maximum allowed jitter value of $S = 1{,}000$ ns.

The left plot in Figure 5.10 displays the runtime used to attempt placement of 100 flows, in addition to the initially scheduled flow set size drawn on the x-axis. We recall that in additional scheduling, the scheduler only tries to place flows in a way that violates no constraints and maximizes the number of placed flows, but pursues no further strategy. We can see that for up to 400 initial flows, the solver terminates within the time limit, which means that the `min-disabled-flows` objective was solved to optimality and all 100 additional flows were successfully placed. We further notice that evaluations with start values take a significantly shorter timespan to solve the problem instances, compared to evaluations using hints. Only in the last case with 500 initial flows both approaches run into the time limit.

The right plot in Figure 5.10 shows the success rate of the same evaluations, i.e. on the y-axis the percentage of additional flows that could be placed is drawn. In relation to the respective runtimes on the left, we can see that both approaches are able to place all 100 additional flows as long as no more than 400 initial flows are already included in the schedule. In the rightmost value set of 500 initial flows, the start value approach managed to place slightly more additional flows on average, and did so with a much better worst case results (82 vs. 32).

Note that we chose offensive scheduling problem sets for this evaluation, as when allowing no jitter, all initial flow start variables are by definition constrained to the previous start time, trivializing the task of finding an initial feasible solution for these variables.

We conclude from both plots that in case of our ILP models, using start values leads to reduced runtimes or, in case of reaching the time limit, to a better solution quality. We assume that start values lead to a faster initial solution, leaving a higher amount of runtime to further improve the initial solution. We assume that Gurobi locks the provided variable start values and searches for an initial solution in the reduced space consisting of the remaining—added for additional flows—variables. This would be in contrast to using the general heuristics to find initial feasible solutions and amending them by guiding decisions using the provided hints. Therefore, we provide the initial flow start times as start values in all further evaluations.

5.2.8 Additional flow placement with defensive scheduling

In this section, we examine the runtime and solution quality of dynamic schedules. Based on the initial scheduling results of two-hour executions of the three strategies `min-flowspan`, `max-freeblock` and `link-desync` discussed previously, we performed defensive scheduling with a varying amount of additional flows.

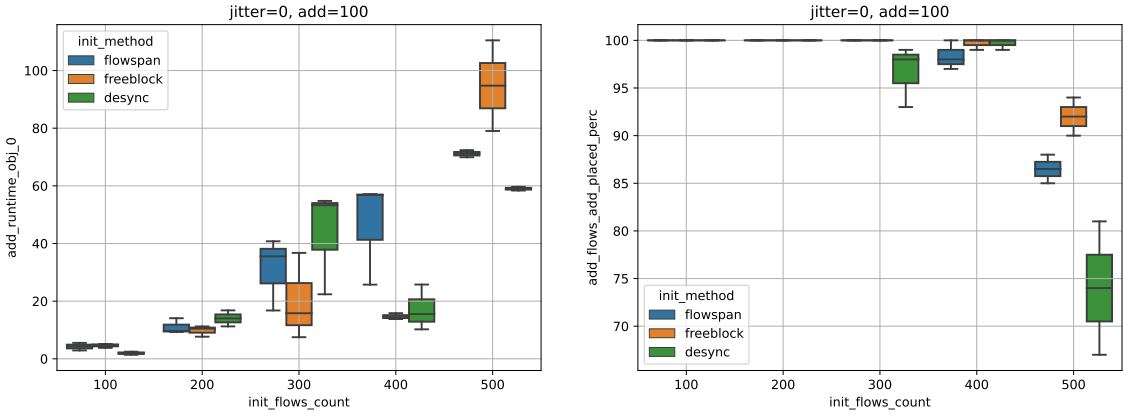


Figure 5.11: Runtime to place additional flows (left) and success rate (right) with defensive scheduling (maximum allowed jitter $S = 0$, runtime limit 600 s). For scenarios with an initial runtime of two hours and an increasing number of initial flows on the x-axis, the plots draw the runtime to place 100 additional flows (left) and the percentage of additional flows placed (right) on the y-axis. The results of initially applying the three scheduling strategies are distinguished by color. Both plots show 95 % confidence intervals.

In Figure 5.11, we plotted the runtime and percentage of additional flows placed for an increasing amount of initial flows on the x-axis with color-distinguished initial strategies. The y-axis denotes the used runtime to place additional flows and the percentage of successfully placed additional flows, respectively. Both plots are sourced from the same defensive scheduling evaluations with a runtime limit of 600 s and 100 additional flows to be placed. We used initial scheduling rounds with a runtime of 7,200 s as a basis.

The left plot of Figure 5.11 shows the runtime to place additional flows, i.e. optimize `min-disabled-flows` and no further objective. A trend of increasing runtimes is generally noticable, which is in accordance with the increased problem complexity, as from left to right, more flows are to be placed in the same limited schedule of cycle $t^{cycle} = 100\,\mu s$. We can see that all executions finish within 120 s, meaning that for defensive scenarios, a network administrator will either receive the full schedule containing all flows after that time, or one containing the highest possible amount.

Up until 300 initial flows, the runtimes differ significantly depending on the initial strategy used, as the interquartile ranges—displayed as colored boxes—do not overlap. In these cases, scenarios with initial strategy `max-freeblock` are generally the fastest, with `min-flowspan` being average and `link-desync` taking the most time. We infer that this is effect is related to the length and amount of free gaps produced in the initial schedules. With `max-freeblock` creating the largest free blocks, it allows the most room for the scheduler to place additional flows without moving initial flows. Next, `min-flowspan` also creates large free gaps towards the end of the cycle, but does not remove gaps before that, leaving the scheduler with a fairly large block at the end, but also small and medium-sized gaps in the beginning and the middle of the cycle, complicating the problem. `link-desync` per definition creates many small free gaps. We assume that it is harder for the scheduler to find assignments for additional flows to a large amount of free gaps, in comparison to a long free gap which can be partitioned for multiple additional flows. For higher numbers—400 and 500—of initial flows, the order of the three methods within the left runtime plot differs from the order expressed for lower initial flows, and no clear trend is noticeable for the strategies in these scenarios.

The right plot in Figure 5.11 shows the corresponding success rate of the same scenarios, i.e. the percentage of additional flows that could be placed when confronted with the additional flow set and an existing schedule based on initial scheduling with the three strategies. We can see that for up to 200 initial flows, all 100 additional flows could be accommodated. Except for the `link-desync` strategy, this is also the case for 300 initial flows. For higher initial flow counts of 400 or 500 flows, not all additional flows could be accommodated, to a varying degree based on the initial scheduling strategy. In the higher three instances of 300 to 500 flows we can see that our `max-freeblock` strategy generally places the highest amount of flows, with either `min-flowspan` or `link-desync` following right after.

As explained in the previous paragraph, we think that the `max-freeblock` strategy performs best in defensive scheduling because of the large free block on every link, which allows a large degree of freedom for the scheduler to place flows in. The strategy only restricts further placement in the case that a flow with a long path starts in an area where all needed links are occupied at the beginning of the cycle. Similarly, `min-flowspan` performs either on par or slightly worse to `max-freeblock`. As already discussed, we think that because of the also relatively large gaps towards the end of all link cycles, the strategy can place a great amount of additional flows. In comparison to `max-freeblock`,

`min-flowspan` is somewhat at a disadvantage in placing additional flows, as the early part of the resulting schedule exhibits gaps that are not removed by the strategy, leading to suboptimal use of the temporal space.

Except for the scenario of 400 initial flows, the `link-desync` strategy leads to the worst results in defensive scheduling, which we anticipated when devising the strategy. The general idea of leaving gaps between every consecutive flow transmission allows for flexibility of moving flows and also for placement of some flows, as long as the gaps are long enough. In defensive scheduling no initial flows may be moved, therefore the scheduler can only use exactly the existing gaps. If these gaps are generally too short, or waste temporal space by not being multiples of the transmission timespan d^{trans}, scheduling additional flows will be inefficient. We reiterate that this strategy was devised mainly for offensive scheduling, which we will evaluate in the next section.

In other defensive scenarios not pictured for space reasons, we evaluated higher amounts of additional flows and got mostly similar results, with `max-freeblock` generally performing best and either `min-flowspan` or `link-desync` following.

5.2.9 Additional flow placement with offensive scheduling

We now examine the performance and quality of our approaches in offensive scheduling. Again, we initially scheduled varying amounts of flows using the three scheduling strategies `min-flowspan`, `max-freeblock` and `link-desync`, this time with a runtime limit of one hour (3,600 s). In the additional scheduling round, we then placed a varying amount of additional flows, in some cases leading to the full set of 600 flows being present for scheduling. We executed and evaluated scenarios with varying runtime limits and allowed amounts of jitter.

We found that when allowing only small jitter values such as $S = 500$ or $S = 1000$, almost no rescheduling occurs and the results are essentially the same as with defensive scheduling, but at the cost of an increased runtime resulting from the higher problem complexity. We also noticed that the scheduler almost never performs rescheduling for shorter runtimes such as 60 s or 600 s, and generally returned poor results compared to higher runtimes or defensive scheduling with these time limits. Therefore, we allowed both a higher runtime of 3,600 s (one hour) and higher jitter values such as $S = 20{,}000$ ns.

Figure 5.12 shows the runtime to try to place additional flows (left), the percentage of successfully placed additional flows (center), and the number of initial flows rescheduled (right), all three with an increasing amount of initial flows drawn on the x-axis. The plots chart the same scenarios of placing 300 additional flows with a maximum allowed jitter of $S = 20{,}000$ ns and a runtime limit of 3,600 s for both the initial and additional scheduling round.

The left plot of Figure 5.12 again shows the runtime of the `min-disabled-flows` objective of the additional scheduling round on the y-axis, with initial scheduling strategies displayed in different colors, for an increasing amount of initial flows on the x-axis. The center plot displays the success rate in percent of the additional flows for the same scenarios. In the right plot of Figure 5.12, the number of initial flows that were rescheduled in additional scheduling, i.e. temporally moved.

Even with the relatively long runtime limit of 3,600 s, scenarios with $x = 200$ flows sometimes were not able to produce an optimal solution within that time while for $x = 300$ initial flows this was never the case. This is no suprise as through allowing the high amount of allowed jitter $S = 20{,}000$ ns,

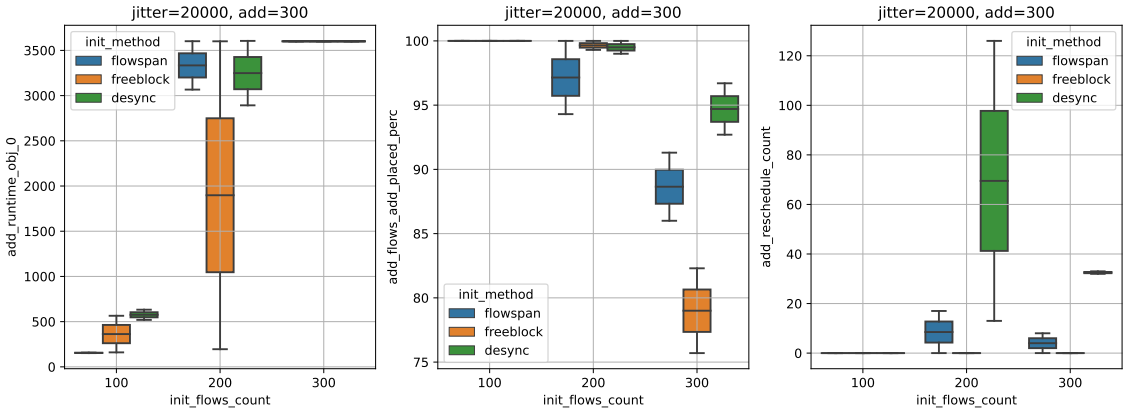


Figure 5.12: Runtime to place additional flows (left), success rate (center) and amount of rescheduling (right) for offensive scheduling (maximum allowed jitter S = 20,000 ns, runtime limit 3,600 s). For scenarios with both initial and additional runtimes of one hour and an increasing number of initial flows on the x-axis, the plots draw the runtime to place 300 additional flows (left), the percentage of additional flows placed (center) and the number of rescheduling occurences (right) on the y-axis. The results of initially applying the three scheduling strategies are distinguished by color. All three plots show 95 % confidence intervals.

we allow a large degree of freedom for every start variable. Although a feasible solution for at least initial flows exists and only a restricted amount of movement is allowed for them, the problem complexity is of the same magnitude as with initial scheduling, as discussed in Subsection 5.2.3.

As for $x = 100$ all flows were placed throughout the strategies with no rescheduling whatsoever, we mainly look at the higher initial flow counts $x = 200$ and $x = 300$. In the left runtime plot of Figure 5.12, it is noticeable that both initial strategies `min-flowspan` and `link-desync` require a significantly greater amount of time to place additional flows than `max-freeblock`. We think that this is connected to the amount of flows rescheduled, as the shorter-running round based on an initial `max-freeblock` solution leads to no rescheduling—as seen in the right—whereas the other two do. This would also be in line with the conclusion of the previous section, wherein the `max-freeblock` strategy is best at placing flows without offensive scheduling (rescheduling). In the center plot we can see that `max-freeblock` was not always able to place all additional flows, but the same can be said for the other two strategies that lead to rescheduling. We conclude that `max-freeblock` for $x = 200$ even beats the other strategies if they are allowed to reschedule flows in this combined flow set containing 500 out of 600 possible flows.

As soon as all 600 flows are to be scheduled ($x = 300$ initial flows combined with 300 added flows), our `link-desync` strategy performs best, placing between 91 % and 98 % of additional flows, as can be seen in the center plot of Figure 5.12. This is in line with our expectation that desynchronizing flow placements within all link schedules allows for a great amount of flexibility. This flexibility was used to move initial flows in order to fit more additional flows, as can be seen in the right plot, which draws the number of rescheduling occurrences. Specifically, the `link-desync` strategy leads to up to 130 rescheduling occurrences for $x = 200$ and almost 40 for $x = 300$.

`min-flowspan` achieved the next best results for $x = 300$, placing between 87 % and 92 % of additional flows. We think that this is the result of the previously described behavior that the strategy only minimizes the last finishing time and leaves all flows placed previously in the cycle as is. In particular, the strategy does not move the flows in the beginning of the cycle together in a back-to-back manner, leaving free gaps that also allow for some degree of flexibility. The result of this flexibility can be observed in the right plot, where `min-flowspan` rescheduled up to 20 flows for $x = 200$ and $x = 300$.

The `max-freeblock` strategy performed worst in offensive scheduling and was not able to move a single flow in order to find better placements for additional flows. This is to be expected as the strategy pushes all flows to the beginning of the cycle for every link, scheduling the flows in a back-to-back manner. With this kind of occupation, rescheduling would be possible at the end of the scheduled traffic block at the most, which is only beneficial in seldom scenarios where additional flows routed over many hops could not be placed because the large free gap at the end would be too short to traverse its path. We conclude that the `max-freeblock` strategy should only be used in defensive scheduling scenarios for which it was designed.

Although allowing jitter of up to $S = 20000$ allowed placing more flows than in defensive scheduling, execution times of an hour or even more are not desirable for dynamic scenarios. Combined with the fact that only a relatively small amount of flows were even rescheduled, this emphasizes the need for a more dynamic approach that strategically selects only some subset of initial flows to reschedule. For this reason, we implemented our dynamic rescheduling algorithm as discussed in Subsection 4.4.2, which we evaluate in the next section.

5.2.10 Dynamic rescheduling algorithm

In this section, we evaluate the performance and solution quality of our dynamic rescheduling algorithm presented in Subsection 4.4.2. We recall that our algorithm counts the number of flows traversing each link in order to mark links as *critical*. Only the flows using the highest number of critical links will get a positive amount of allowed jitter assigned, hopefully reducing the complexity to only the necessary areas and therefore drastically reducing the execution time, or increasing the solution quality calculated when running within a fixed amount of time.

As described in detail in Subsection 4.4.2, the algorithm is tunable in different ways:

- The set of flows to consider when marking links (counting flows on them) can either be the initial flows, the additional flows or both combined.
- We can choose the percentile of highly-occupied links that shall be marked *critical*
- The flows considered for rescheduling are selected based on either a threshold or as a percentile when sorting the flow set by the number of traversed critical links.
- The allowed jitter can either be static for all selected flows, or be calculated by multiplication of the number of used critical links by a provided value

In the following, we present the results of applying varying values for the mentioned tuning parameters and analyze which settings lead to the best results given our scenarios.

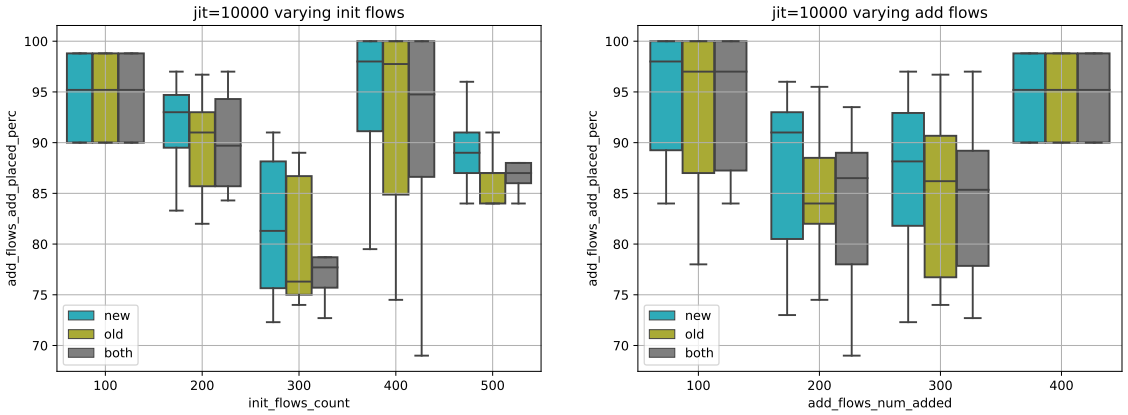


Figure 5.13: Success rates of placing additional flows using the dynamic rescheduling algorithm, varying the set of flows used to mark links as critical. The y-axis plots the percentage of additional flows placed in additional scheduling after applying the dynamic jitter algorithm, with an increasing number of initial (left) or additional (right) flows. All scenarios are based on initial scheduling with 3,600 s runtime and a maximum jitter of $S = 10{,}000$ ns (only for initial flows selected for rescheduling). The dynamic jitter algorithm was applied for all combinations of $CUTOFFPERC^{mark} \in \{10, 25, 50\}$ and $CUTOFFPERC^{select} \in \{10, 25, 50\}$

Figure 5.13 shows the 95 % confidence intervals for the success rate of additional scheduling after the dynamic rescheduling algorithm was run and marked links using either F^{init}, F^{add} or F^{comb}. Although none of the interquartile ranges are free of overlap, which would indicate a *statistically significant* difference, we can see a trend in relation to the flow set used to mark links: On average, marking just with new (additional) flows achieves the highest success rate in every measurement encountered. This makes sense, as a high degree of flexibility along the paths of new (additional) flows gives a higher chance of accommodating those flows. Of course, this is not exclusively the case, as can be seen by the interquartile ranges of the other two modes, i.e. of marking using the initial or the combined flow set. We can also think of cases where marking using e.g. the initial set would be beneficial: Consider initial flows that are scheduled mostly on the outer area of the network graph and only use one or two links in the central network area. Marking using new (additional) flows running along this edge will most likely not lead to the initial flows being moved, as with only one critical link, the flows fall below the cutoff percentile. Now, if this scenario were marked with the combined or initial flow set, the highly used outer edges would likely be critical and the aforementioned initial flows likely be selected for rescheduling as they use a great amount of them.

Based on the highest mean performance, we chose to continue exploration of tuning parameters only with algorithm configurations that mark using *new* (additional) flows.

In terms of the two cutoff percentile we executed additional scheduling rounds for all combinations of $CUTOFFPERC^{mark} \in \{20, 40, 60, 80, 100\}$ and $CUTOFFPERC^{select} \in \{20, 40, 60, 80, 100\}$. Using the heatmaps depicted in Figure 5.14, we interpreted the results in order to select parameters to continue with. The heatmaps are based on scenarios with 400 initial flows and the results of 100 and 200 additional flows. The average of these executions for our three flow sets are expressed with a color scale.

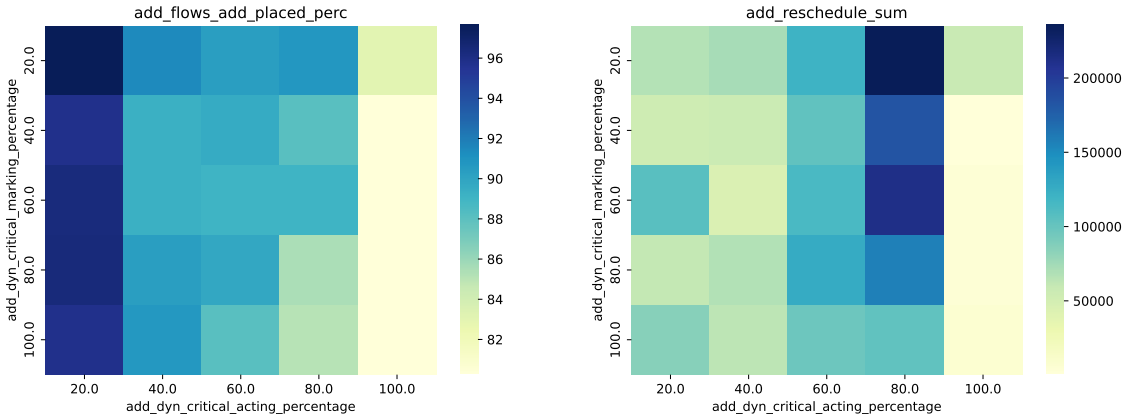


Figure 5.14: Heatmaps for tuning the dynamic rescheduling algorithm based on scenarios with 400 initial and both 100 and 200 additional flows. Both plots explore varying settings for $CUTOFFPERC^{mark}$ (y-axis) and $CUTOFFPERC^{select}$ (x-axis), i.e. the cutoff percentile parameters for marking links and selecting flows within our dynamic rescheduling algorithm. The coloring indicates the success rate in placing flows (left) and the overall sum of rescheduling movements experienced (right).

We can see in the left column of the left plot that for this scenario, $CUTOFFPERC^{select} = 20$ lead to the highest percentage of additionally placed flows, expressed by the dark blue pattern. The overall highest placement ratio was achieved when also setting $CUTOFFPERC^{mark} = 20$, displayed in the top left corner of the left plot. Conveniently, this selection also lead to a relatively small amount of rescheduling, as the top left corner on the right plot expresses a light green pattern, indicating a small sum of rescheduling amounts.

We must note that the heatmaps express a different allocation for other numbers of initial flows. We conclude that the chosen parameters are of course not universally optimal and must be chosen in line with the real-life problem sizes at hand.

Now, we look into the solution quality and characteristics of using our dynamic rescheduling algorithm. Figure 5.15 shows the success rate (left) and number of rescheduled flows (right) for scenarios with static vs. dynamic jitter. In the scenarios, we initially scheduled 400 flows with our three strategies, limited to 3,600 s runtimes. Then, we added 200 flows with a time limit of 600 s and allowed up to $S = 10{,}000$ ns of jitter. As all depicted executions used the whole 600 s, we included no runtime plot. We recall that when statically applying the jitter value, every initial flow can be rescheduled by amount S, whereas in our dynamic rescheduling approach, we only apply this limit to a subset of initial flows, leaving the remainder fixed. As discussed previously in this section, we chose $CUTOFFPERC^{select} = 20$ and $CUTOFFPERC^{mark} = 20$ as parameters for our algorithm.

In the left plot in Figure 5.15, it can be seen that the dynamic approach (x = "new") achieves significantly better results, as the interquartile ranges within the box plot do not overlap. In particular, all three strategies always placed at least 92 % of the additional flows, whereas almost all static executions could never place more than 90 %. We deduct that applying our dynamic rescheduling algorithm reduced the problem complexity in such a way that the scheduler was able to try more promising assignments and thereby place more flows in the same limited amount of time.

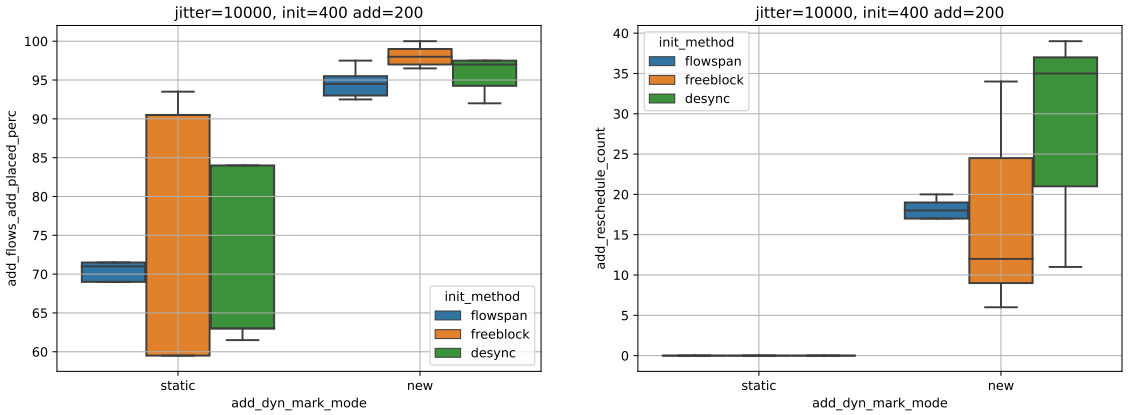


Figure 5.15: Success rate and rescheduling count when using static vs. dynamic jitter values in a scenario of adding 200 flows to 400 initial flows with a runtime of 600 s after initial scheduling limited to 3,600 s. On the x-axis, the static and dynamic ("new") approach is sketched in. The y-axis denotes the percentage of additional flows placed (left) and the number of flows rescheduled (right).

The right plot in Figure 5.15 shows the number of rescheduled flows by approach and strategy. For the static approach, no flows were rescheduled at all, while the dynamic approach utilized the gained degrees of freedom to find better flow placements by rescheduling up to 40 initial flows. This indicates that statically allowing the rescheduling of all initial flows increases the problem complexity so much, that the scheduler is not able to find a solution improving flow placement in comparison to defensive scheduling, which essentially resulted in these cases.

5.3 Summary

We now briefly summarize the findings of our evaluation.

First, we analyzed the complexity and runtime of the base scheduling problem, that is, placing up to 500 flows without conflict, disregarding any higher-level strategies. We found that the runtime increases in a quadratic fashion, in line with the quadratic growth of LP variables, LP constraints for linearly increased flows. Scheduling the highest amount of initial flows took up to one hour on our 8-core, 16-thread machine.

Regarding the LP model complexity of the three strategies under test, we saw that the simpler formulated strategies `min-flowspan` and `max-freeblock` exhibit model sizes in the same vicinity as the base scheduling problem, while the more complex `link-desync` strategy expressed 50 % greater model sizes. To no surprise, the latter two, more sophisticated strategies could not be solved to optimality within the time limit of two hours, while the baseline `min-flowspan` accomplished this at least for smaller flow sets of up to 300 flows. Nonetheless, the effects of the three strategies could be observed in the distribution of free gap lengths. Both `min-flowspan` and `max-freeblock` created many very short gaps resulting of flows scheduled in a back-to-back manner, while `link-desync`

lead to a large number of longer gaps, resulting from the desynchronization objective. This effect was even noticeable when comparing the distributions for a varying initial runtime, showing a clear shift in the end gap lengths.

Before evaluating the actual dynamic scenarios, we compared two modes of providing Gurobi with start times of an existing schedule. We concluded that using Start times leads to shorter execution times and better solution quality as opposed to hints.

Next, we examined the effect of our initial strategies on the task of placing additional flow into an existing schedule in both defensive and offensive scenarios. In defensive scheduling scenarios, our `max-freeblock` strategy lead to a better placement of additional flows, averagely accommodating 92 out of 100 additional flows into a 500-flow schedule, whereas the baseline `min-flowspan` only managed to place 87 % on average. For offensive scenarios, our `link-desync` strategy devised for it achieves the best results for highly-contended scheduling problems, placing about 95 % of 300 additional flows on average, compared to 88 % achieved by the baseline `min-flowspan`.

For our dynamic rescheduling algorithm, we first explored promising parameters and with them, evaluated offensive scheduling. In comparison to allowing the same amount of jitter for all initial flows, our dynamic rescheduling algorithm was able to place significantly more flows in the same time frame of 600 s. Using the dynamic algorithm lead to placement of 97 % of 200 additional flows into a 400-flow schedule, while the static approach only placed 72 %, on average.

6 Conclusion and outlook

Time-Sensitive Networking brings real time functionality into standard IEEE 802.3 Ethernet, especially with the Time-aware shaper (TAS), a TDMA scheme for controlling access to the transmission links. In order to guarantee real-time properties for time-sensitive traffic flows, a global schedule coordinating their transmission times must be calculated, which poses an $\mathcal{NP}$-hard problem. For a static network and a priori known flows, different solutions based on constraint satisfaction solvers have been proposed which produce optimal solutions at the cost of long execution times. For dynamic scenarios such as extending an existing TSN network by further flows or machines, those long-running approaches are unsuitable because of their runtime. Also, creating a new schedule from scratch may assign existing flows differently, introducing an unbounded amount of jitter. Therefore, dynamic approaches need to take original schedules into account and amend them with the additional flow set and either not move originally scheduled flows at all, or only move them by a restricted amount. Most approaches for the dynamic TSN scheduling problem rely on heuristics, which often do not consider the complete solution space and may lead to sub-optimal solutions. In contrast, using commercial ILP solvers to solve the problem results in provable optimal solutions and at the same time brings support for adding higher-level optimization objectives.

In this work, we proposed an ILP model for solving the dynamic TSN scheduling problem based on the NW-PSP, which includes different objectives for improving future schedulability as well as an algorithm that strategically limits the set of flows to be rescheduled. We started by defining a model that supports incrementally adding new flows to an existing schedule, taking original flow placements into account. The model supports both defensive scheduling, in which existing flows are not moved, as well as offensive scheduling, where existing flows may be rescheduled by a limited amount of time. In addition to a base objective from the literature, which minimizes the flowspan analogous to the makespan of the NW-JSP, we proposed two further objectives intended to increase future schedulability. The first objective, `max-freeblock`, attempts to create a large gap of unoccupied time towards the end of each link schedule, intended to create enough space to place future flows. The second objective, `link-desync`, aims to maximize the gap between every two consecutive flows within a link schedule, generating flexibility for future offensive scheduling, as rescheduling a neighboring flow by a small amount may suffice to place additional flows. Further, we formulated ILP objectives that reduce the number of flows rescheduled as well as the maximum and total rescheduling timespan. This objective further reduces the jitter experienced by existing flows when the new schedule is applied. To decrease the solver's runtime and further limit rescheduling, we proposed a heuristic-based algorithm that strategically selects a subset of flows for rescheduling based on link occupancy.

In our evaluations, we first analyzed the complexity and runtime of the base scheduling problem as well as for our proposed strategies. Both the ILP complexity and the corresponding execution times increased quadratically in relation to the number of flows to be scheduled. We verified the correct formulation and implementation of our strategies by studying the distribution of free gap lengths within schedules created by the strategies. Specific to the ILP solver we used, Gurobi,

we found that supplying existing schedules as start values of variables outperforms contributing them via so-called *hints*. Next, we examined the efficacy of our scheduling strategies in defensive and offensive scheduling scenarios. Our `max-freeblock` strategy outperformed both the baseline `min-flowspan` and our second strategy `link-desync`. For offensive scenarios, the `link-desync` strategy lead to the highest placement ratio of additional flows in the scenarios discussed. At last, we tuned the parameters of our dynamic rescheduling algorithm and evaluated the resulting solution quality. Using our algorithm, the scheduler was able to place an average of 97 % of flows in offensive scenarios. In comparison, only 72 % of flows could be placed when the set of flows to be rescheduled was unrestricted.

For future work, we suggest further exploration the iterative nature of our dynamic scheduling approach beyond the two-phase addition of flows. We noticed that in some instances, scheduling a divided flow set in two phases was faster than scheduling the complete set in one step. Following that observation, there may be performance gains in also using the incremental scheduling process for initial offline scheduling scenarios.

For another interesting research question, we thought of a self-healing scheduling automatism that constantly improves and deploys the global schedule of an active TSN network. Such an approach would further reduce the runtime for scheduling initial or additional flows, because the flows could be placed in a suboptimal manner at first, to be gradually optimized later-on, with each optimization step adhering to the jitter constraints.

Bibliography

[14] "IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks". In: *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)* (2014), pp. 1–1832. DOI: 10.1109/IEEESTD.2014.6991462 (cit. on p. 15).

[20a] "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. DOI: 10.1109/IEEESTD.2020.9121845 (cit. on pp. 11, 16).

[20b] *Propagation Delay and Its Relationship to Maximum Cable Length*. https://www.liveaction.com/docs/glossary/ethernet-ieee-802-3/propagation-delay/. May 2020. URL: https://www.liveaction.com/docs/glossary/ethernet-ieee-802-3/propagation-delay/ (cit. on p. 50).

[BDE+18] R. Belliardi, J. Dorr, T. Enzinger, F. Essler, J. Farkas, M. Hantel, M. Riegel, M. Stanica, G. Steindl, R. Wamßer, et al. *Use Cases IEC/IEEE 60802 v1. 3*. 2018 (cit. on p. 12).

[COCS16] S. S. Craciunas, R. S. Oliver, M. Chmelik, 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: ACM, 2016, pp. 183–192. ISBN: 978-1-4503-4787-7. DOI: 10.1145/2997465.2997470. URL: http://doi.acm.org/10.1145/2997465.2997470 (cit. on p. 12).

[CYH+17] B. Chen, Z. Yang, S. Huang, X. Du, Z. Cui, J. Bhimani, X. Xie, N. Mi. "Cyber-physical system enabled nearby traffic flow modelling for autonomous vehicles". In: *2017 IEEE 36th international performance computing and communications conference (IPCCC)*. IEEE. 2017, pp. 1–6 (cit. on p. 11).

[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*. 2016, pp. 203–212 (cit. on pp. 11, 12, 16, 19, 20, 31, 36).

[Eth16] T.-T. Ethernet. "AS6802™". In: (2016) (cit. on pp. 11, 15).

[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)*. IEEE. 2018, pp. 136–146 (cit. on p. 12).

[Fel04] J. Feld. "PROFINET-scalable factory communication for all applications". In: *IEEE International Workshop on Factory Communication Systems, 2004. Proceedings.* IEEE. 2004, pp. 33–38 (cit. on pp. 11, 15).

[FGD+21] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. “Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows with Conflict Graphs”. In: *arXiv preprint arXiv:2105.01988* (2021) (cit. on pp. 12, 24, 25, 28, 29).

[FHHW03] T. Fuehrer, R. Hugel, F. Hartwich, H. Weiler. *FlexRay-the communication system for future control systems in vehicles*. Tech. rep. SAE Technical Paper, 2003 (cit. on p. 15).

[Glo90] F. Glover. “Tabu search: A tutorial”. In: *Interfaces* 20.4 (1990), pp. 74–94 (cit. on p. 20).

[GRK+21] C. Gärtner, A. Rizk, B. Koldehofe, R. Hark, R. Guillaume, R. Steinmetz. “Leveraging Flexibility of Time-Sensitive Networks for dynamic Reconfigurability”. In: (2021) (cit. on pp. 12, 23).

[Gur21] O. T. Gurobi. *Gurobi optimizer reference manual, Version 9.1, 2021*. Gurobi Optimization, LLC. 2021. URL: https://www.gurobi.com/wp-content/plugins/hd_documentations/documentation/9.1/refman.pdf (cit. on pp. 19, 33, 35, 36).

[HMFH+00] F. Hartwich, B. Müller, T. Führer, R. Hugel, et al. “CAN network with time triggered communication”. In: *7th international CAN Conference*. Citeseer. 2000 (cit. on p. 15).

[Ins16a] Institute of Electrical and Electronics Engineers Inc. “IEEE Standard for Ethernet Amendment 5: Specification and Management Parameters for Interspersing Express Traffic”. In: *IEEE Std 802.3br-2016 (Amendment to IEEE Std 802.3-2015 as amended by IEEE St802.3bw-2015, IEEE Std 802.3by-2016, IEEE Std 802.3bq-2016, and IEEE Std 802.3bp-2016)* (Oct. 2016), pp. 1–58. DOI: 10.1109/IEEESTD.2016.7900321 (cit. on pp. 21, 28).

[Ins16b] Institute of Electrical and Electronics Engineers Inc. “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— as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q—/Cor 1-2015)* (Mar. 2016), pp. 1–57. DOI: 10.1109/IEEESTD.2016.7572858 (cit. on p. 15).

[Ins20] Institute of Electrical and Electronics Engineers Inc. *Time-Sensitive Networking Task Group*. http://www.ieee802.org/1/pages/tsn.html. July 1, 2020. URL: http://www.ieee802.org/1/pages/tsn.html (cit. on p. 15).

[LS10] I. Lee, O. Sokolsky. “Medical cyber physical systems”. In: *Design automation conference*. IEEE. 2010, pp. 743–748 (cit. on p. 11).

[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 (1999), pp. 2585–2607 (cit. on p. 20).

[NDR16] N. G. Nayak, F. Dürr, K. Rothermel. “Time-sensitive software-defined network (TSSDN) for real-time applications”. In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 193–202 (cit. on p. 21).

[NDR17] N. G. Nayak, F. Dürr, K. Rothermel. “Incremental flow scheduling and routing in time-sensitive software-defined networks”. In: *IEEE Transactions on Industrial Informatics* 14.5 (2017), pp. 2066–2075 (cit. on pp. 12, 21, 22).

[PHL+20] Z. Pang, X. Huang, Z. Li, S. Zhang, Y. Xu, H. Wan, X. Zhao. “Flow Scheduling for Conflict-Free Network Updates in Time-Sensitive Software-Defined Networks”. In: *IEEE Transactions on Industrial Informatics* 17.3 (2020), pp. 1668–1678 (cit. on p. 12).

[PSRH15] F. Pozo, W. Steiner, G. Rodriguez-Navas, H. Hansson. “A decomposition approach for SMT-based schedule synthesis for time-triggered networks”. In: *2015 IEEE 20th conference on emerging technologies & factory automation (ETFA)*. IEEE. 2015, pp. 1–8 (cit. on p. 12).

[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*. 2017, pp. 8–17 (cit. on p. 12).

[SF20] S. Funke. *Lecture notes for Discrete Optimization*. Feb. 4, 2020. URL: https://fmi.uni-stuttgart.de/files/alg/teaching/dopt/DiskOpt.pdf (cit. on pp. 17, 19).

[SL86] C. Sriskandarajah, P. Ladet. “Some no-wait shops scheduling problems: Complexity aspect”. In: *European Journal of Operational Research* 24 (3 1986), pp. 424–438. ISSN: 0377-2217 (cit. on pp. 11, 16, 19).

[SWHD20] P. Schneefuss, M. Weitbrecht, D. Hellmanns, F. Dürr. “Benchmarking TSN Schedulers”. 2020 (cit. on p. 49).

[XYGG18] H. Xu, W. Yu, D. Griffith, N. Golmie. “A survey on industrial Internet of Things: A cyber-physical systems perspective”. In: *IEEE Access* 6 (2018), pp. 78238–78259 (cit. on p. 11).

[Yen71] J. Y. Yen. “Finding the k shortest loopless paths in a network”. In: *management Science* 17.11 (1971), pp. 712–716 (cit. on p. 24).

All links were last followed on February 14, 2022.

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