Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Future-proof Scheduling Heuristics for TTEthernet

Rico Haas

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Christian Becker
Supervisor:	Heiko Geppert, M.Sc.
Commenced:	April 2, 2024
Completed:	October 2, 2024

Abstract

Time-triggered networks run on a premade schedule to ensure that all messages arrive at their destination on time and without being dropped. They are deployed in safety-critical environments like airplanes, cars, but also factories. Online scheduling algorithms can create schedules for such networks in mere seconds while also being able to modify an existing schedule. With an algorithm of this kind, new devices can be plugged into its time-triggered network and start interacting with it in seconds, without manual setup. We propose modifications for improving two already existing online scheduling algorithms: Hierarchical Heuristic Scheduling (H2S) and Cost-Efficient Lazy Forwarding (CELF). We call these modifications "relaxed jitter", "backward rifts", and "gap closing". Our modifications achieved about 550 MBit/s more aggregated throughput while only increasing solving time by 2.5ms in a network with 25 switches.

Kurzfassung

Zeitgesteuerte Netzwerke nutzen einen vorgefertigten Zeitplan um sicherzustellen, dass alle Nachrichten ihr Ziel rechtzeitig erreichen ohne verloren zu gehen. Sie werden in sicherheitskritischen Umgebungen wie Flugzeugen, Autos, aber auch Fabriken, eingesetzt. Online Planungsalgorithmen können Zeitpläne für solche Netzwerke in wenigen Sekunden generieren und auch bereits existierende Zeitpläne anpassen. Mit einem solchen Algorithmus können neue Geräte an ein zeitgesteuertes Netzwerk angeschlossen werden und sind dann innerhalb von Sekunden ohne manuellen Eingriff einsatzbereit. Wir schlagen Verbesserungen für zwei bereits existierende Online Planungsalgorithmen vor: Hierarchical Heuristic Scheduling (H2S) und Cost-Efficient Lazy Forwarding (CELF). Wir nennen diese Verbesserungen "Relaxed Jitter", "Backward Riftsünd "Gap Closing". Unsere Verbesserungen erreichten etwa 550 MBit/s zusätzlichen aggregierten Netzwerkdurchsatz, bei einer nur leicht erhöhten Berechnungszeit von 2,5ms in einem Netzwerk mit 25 Switches.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ALAP as late as possible. 39

ASAP as soon as possible. 36

BE best effort. 20

CELF Cost-Efficient Lazy Forwarding. 27

H2S Hierarchical Heuristic Scheduling. 8

OSI Open Systems Interconnection. 17

RC rate constrained. 21

TSN Time-Sensitive Networking. 16

TT time triggered. 21

TTE TTEthernet. 16

TTN time-triggered network. 15

1 Introduction

Communication over the internet is inherently unreliable: The internet is an event-triggered network, meaning that switches simply forward incoming frames as they arrive without planning ahead [KAGS05]. This means that if the traffic in the network becomes unusually high, the switches might not be able to forward incoming frames fast enough; frames might get severely delayed or dropped entirely. For everyday internet use, this is fine. Protocols like TCP work around this unreliability by re-sending frames until the receiver acknowledges them [Edd22]. But re-sending frames causes delay that some applications cannot tolerate: In an industrial network for example, there could be a critical periodic sensor reading that the system needs to respond to within a strict time frame [KLR94].

The sensor reading frame in our example requires real-time communication. When we send a frame with real-time requirements, we do not necessarily mean it must arrive "very fast" but "without violating the given deadline". Deadlines can differ for different frames. To mathematically ensure that no deadline is ever violated, we can utilize time-triggered networks (TTNs) instead of event-triggered ones. These do not forward frames as they come in but instead operate on a stringent schedule [KAGS05]. When constructing such a schedule, the network topology, all communication flows, and their requirements, have to be known in advance. Once a schedule has been devised, every end device has precise preassigned moments in time when it is allowed to send Ethernet frames. Switches also have precise instructions on 1) when they have to forward the frames they receive and, 2) over which egress (outgoing) ports. If every act of communication is preplanned, unexpected congestion becomes impossible. As long as there is no hardware failure, all switches will forward every frame on time, every time.

In order to properly function, every device that is part of a TTN needs a copy of the schedule it operates on and all switches need to be synchronized. There exist several ideas for achieving network time synchronization; some of them send additional dedicated frames over the network [WAA+15], others piggyback synchronization metadata to frames that would have been sent anyway [RAK10]. In the context of this thesis, we rely on the person setting up the network to ensure it is properly synchronized. If they want to use dedicated frames, they could have the scheduler explicitly accommodate those. If they want to use piggybacking, they could instead have the scheduler increase the size of already existent frames.

TTNs can be static, e.g., inside a car. The devices inside a car are known and the topology does not change, so it suffices to precalculate a schedule which the network then uses for its entire lifetime. But theoretical smart industrial networks have a higher need for flexibility: Machines can be added or removed from their factory network, to modernize or scale up assembly processes, or even to enable the manufacturing of highly customized products. The factory network should be able to handle these modifications and incorporate new devices into the network without the need for manual reconfiguration. Schedules need to adjust themselves to accommodate the new communication flows, a new schedule has to be devised in a timely manner and be rolled out into

the network to make sure every device is operating on the most recent schedule. In this thesis, we mainly focus on the quick calculation of an updated schedule. In static TTNs, it does not matter when calculating the schedule takes hours or days. In dynamic TTNs, there is no time for heavyweight optimizations and the scheduling algorithms need to make use of heuristics to generate schedules for large TTNs in seconds.

In the context of the simulations we run later in this thesis, it should be noted that our network topologies never change while a simulation is running on them. We argue that, instead of adding a new end device to the topology and then adding new flows for that end device, simply adding the new flows to an an existing end device that is connected to the desired switch has a similar effect.

When implementing a TTN, there are several protocols to choose from. Most prominent are the Time-Sensitive Networking (TSN) standards, provided by the IEEE 802.1 working group, which offer standardized means of time synchronization, traffic shaping, and the rolling out updated schedules to all network devices. Though in this thesis, we mainly focus on the TTEthernet (TTE) protocol, which offers these benefits as well, while also being less restrictive when it comes to scheduling: TTE allows network switches to buffer incoming frames and have random (unhindered) access to them when it is time to forward them. TSN is more restrictive, instead of a flexible buffer, every switch has eight egress queues per egress port. Switches cannot forward a received frame at any time but must instead assign it an egress queue, which sends the frame as soon as possible. The switches do have control over a gate control unit which can block egress ports at scheduled times. By blocking all queues except one, they can emulate TTE's random access to some degree, but only as long as a switch does not need to buffer more than eight frames at a time.

2 Preliminaries

We begin by introducing the models and protocols on which we base the abstractions in this thesis. These are the OSI model, describing how different network protocols interact with each other; the Ethernet protocol, a very widespread fundamental networking protocol; and TTE, an extension of the Ethernet protocol for TTNs. Finally, we introduce terms and concepts related to TTN scheduling.

2.1 OSI Model

Since networking is a complex process with varying requirements, it is appropriate to not establish a single monolithic protocol for networking but to divide the problem into smaller parts. For this purpose, the Open Systems Interconnection (OSI) reference model defines a stack of protocols [KDD14], several of them layered atop each other as seen in Figure 2.1. In practice, a network message is built from top to bottom. Initially, the network message is just an instance of the highest layer protocol in the stack. This could be a HTTP packet for example, containing the data we want to send. This packet then is encapsulated by the protocol one layer below, possibly TCP, which ensures that the receiver acknowledges received packets and maintains the order that the packets were sent

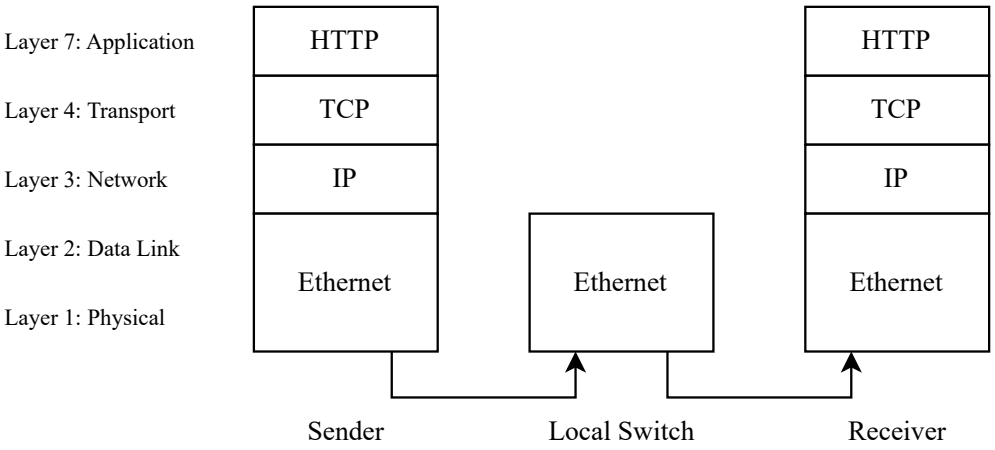

Figure 2.1: An example protocol stack and the respective OSI layers. Ethernet covers both OSI layers 1 and 2. Not all layers are always required: Layer 5 and 6 are omitted in this example. The local switch only needs to be able to process the Ethernet protocol to do its job, the other protocols are just payload that it will forward as is.

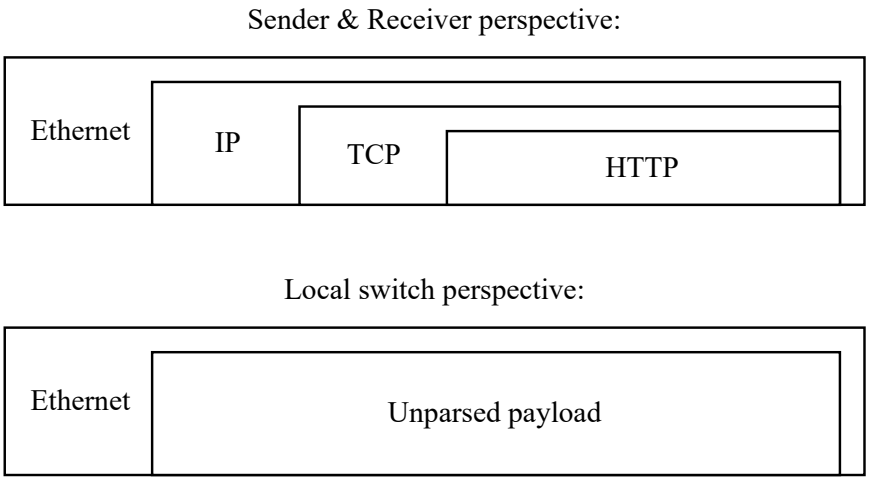

Figure 2.2: This is what a packet of the protocol stack shown in Figure 2.1 actually looks like. To construct this package, the sender starts with an HTTP package in the center and repeatedly encases it with several different protocols. The receiver then unpacks everything in reverse, working through the protocols outside-in until they end up with just the HTTP package in the center. A local switch would just look at the outermost Ethernet part without interpreting any of the layers within.

in. So now we have a HTTP packet inside a TCP packet. This continues down the protocol stack, encapsulating the message further and further as seen in Figure 2.2. The receiver of the message reverses this process, unpacking the message from the bottom to the top of the protocol stack. In the end, the receiver unpacks the original HTTP packet with the actual data that this whole process served to deliver.

Each protocol layer has different responsibilities: Higher layer protocols provide benefits that are close to the application context, like standardized data formats and automated resending of messages lost in transit. Lower layer protocols provide benefits closer to the hardware context like routing a message through a network or ensuring that there are no bit errors during transit. All protocols in the stack are replaceable, as long as all network devices support the protocols they are supposed to process. In our example, the TCP protocol has unnecessary overhead if we do not care about the sender acknowledging our message. To save resources, we could replace TCP with UDP. As long as both end devices support UDP, this works without trouble.

In this thesis, we primarily focus on OSI layer 2 protocols, which mainly handle the organized forwarding of packets through the switches of a local network, as well as ensuring the integrity of these packets.

Preamble	7B
SOF	1B
Destination MAC	6B
Source MAC	6B
VLAN tag	4B
EtherType	2B
Payload	42-1500B
FCS	4B
IFG	12B

Figure 2.3: Structure of an Ethernet frame

2.2 Ethernet

Ethernet is defined in the IEEE 802.3 standards [18], a collection of standards that has been expanded over time, making up the ubiquitous Ethernet we use today. They cover both layer 1 and layer 2 of the OSI model. At layer 1, the standards define the technical specifications of various Ethernet cables used for data transfer. In the early days of Ethernet, multiple network devices shared a single coaxial cable. When one network device sent messages over this cable, all other devices received that message and then had to discern whether the message was directed at them or at another device ("multidrop"). Later on, networks shifted from multidrop to a point-to-point architecture where each cable only ever connects two devices. Point-to-point cables can be used in either half-duplex or full duplex mode. In half-duplex, both connected devices can still interfere with each other if they attempt to send messages to each other at the same time. In full duplex, both devices can use the Ethernet cable simultaneously.

On OSI layer 2, the "data link layer", Ethernet ensures a reliable transfer of Ethernet frames in the scope of a single cable. Each device is assigned a MAC address, where each MAC address is unique in the scope of the network. Using this address, Ethernet packets can be forwarded through the network to the desired end device.

The makeup of an Ethernet frame is visualized in Figure 2.3, with each segment explained here:

1. Preamble: 7 bytes of alternating bits: 10101010... Historically, the preamble served to synchronize bit intervals, allowing the device to determine the exact tact at which bits were sent.
2. SOF: 1 byte indicating the end of the preamble: 10101011
3. Destination MAC: The 6 byte MAC address of the device that is supposed to receive the Ethernet frame.
4. Source MAC: The 6 byte MAC address of the device that is sending the Ethernet frame.

5. VLAN tag: This tag is optional and 4 bytes long, used to support virtual networks (VLANs) and provide additional information about frame priority and congestion management.
6. EtherType: 2 bytes long information about the type of payload, which essentially indicates the protocol that is stacked on top of Ethernet in OSI layer 3.
7. Payload: The actual content that's supposed to be sent. Can be between 42 and 1500 bytes long (between 46 and 1500 bytes if the VLAN tag is not present). If the payload is smaller than the required minimum, it needs to be padded with zeroes to fulfill the requirement.
8. FCS: The frame check sequence, 4 bytes long, allowing the receiver to confirm that no bit errors occurred during transmission. It is essentially a checksum of the frame.
9. IFG: A 12 byte long interframe gap, clearly separating this Ethernet frame from the next.

In total; when counting the preamble, SOF, and IFG; the size of an Ethernet frame cannot be smaller than 84 bytes and not be larger than 1542 bytes. The preamble, SOF, and IFG are required for sending consecutive Ethernet frames but they are, depending on the context, often not considered part of the Ethernet frame.

2.3 TTEthernet

Compared to other protocols for TTNs, TTEthernet's (TTE) main focus is to create mixed-criticality networks that not only have stringently scheduled traffic, but also regular Ethernet (or "legacy Ethernet") traffic, all in a single network. To this end, scheduled traffic always has priority over legacy Ethernet traffic. Like in conventional Ethernet networks, legacy Ethernet traffic is handled as best effort (BE) traffic, i.e., there are no guarantees on how quickly the frames reach their destination or whether they get there at all. Like in regular Ethernet networks, higher layer protocols like TCP are required to ensure reliability for applications that utilize BE traffic.

The TTE standard was developed by SAE International, primarily for applications in the aerospace domain. In an airplane, there is usually a network for electronic navigation and guidance and a separate network for passenger amenities, like internet access. Combining these two into a single network that does not jeopardize the safety and security requirements of critical applications reduces the amount of wires and network components needed; reducing the weight and general operating cost of the airplane.

TTE operates on OSI layer 2, commonly known as the data link layer [Ade22]. The physical layer below TTE must provide a constant communication latency [KAGS05], which renders wireless transmissions unsuitable for TTE.

2.3.1 Traffic Classes

In TTE, traffic can be categorized into three classes [Fan]:

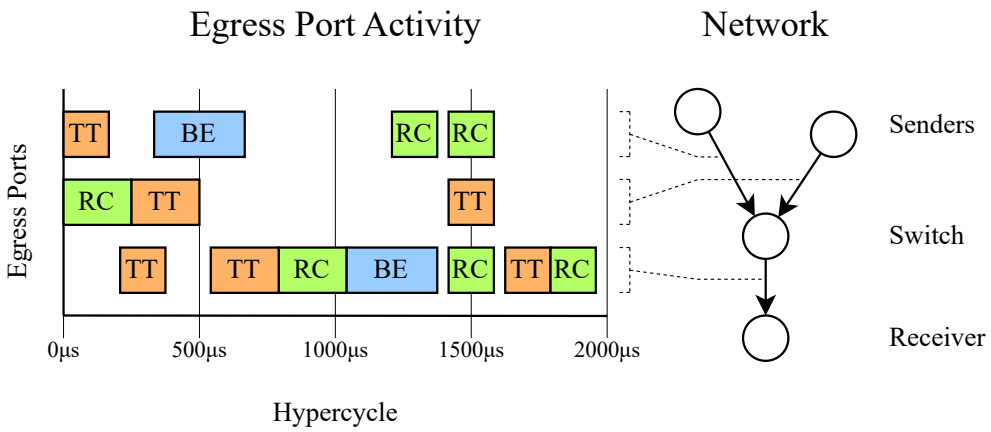

Figure 2.4: A TTE switch forwards messages of different types form two ingress ports into one egress port.

- Time triggered (TT) traffic is stringently scheduled. Every sender has a schedule for sending frames of these types and each switch has a schedule for receiving and forwarding them; TT traffic has unconditional priority over the other two traffic classes. Because of this, latency and jitter for this type of traffic is tightly bounded (i.e., it will not ever exceed a precalculated maximum).
- Rate constrained (RC) traffic is not scheduled, but every sender node has a budget of data it is allowed to send as RC traffic in a given time frame, along with a burst limit. The network has to reserve bandwidth to make sure the latSency of this traffic is bounded and lossless; the network must always ensure this, even in worst case burst scenarios. When using RC traffic instead TT traffic, latency bounds are a lot higher. This fact may make RC traffic unsuited for applications centered around safety, but still ideal for reliable audio and video streaming applications.
- Best effort (BE) traffic is legacy Ethernet traffic, as defined in IEEE 802.3 [18]. Network switches can accept traffic of this kind but will delay or drop it if TT and RC traffic already take up all their available capacity.

Apart from that, there is also traffic for network startup and upkeep, mainly intended for the synchronization of clocks and schedules. A TTE schedule defines the precise send and receive times of each TT frame passing through the network. When the end of a schedule is reached, it begins anew from the start, looping indefinitely.

2.3.2 Frame Forwarding

A full network schedule can be broken up into schedules for each individual node. A node's schedule can then be broken up further into receive schedules for each ingress port (i.e., which TT frames arrive at what time) and send schedules for each egress port (i.e., which TT frames have to be sent out at what time).

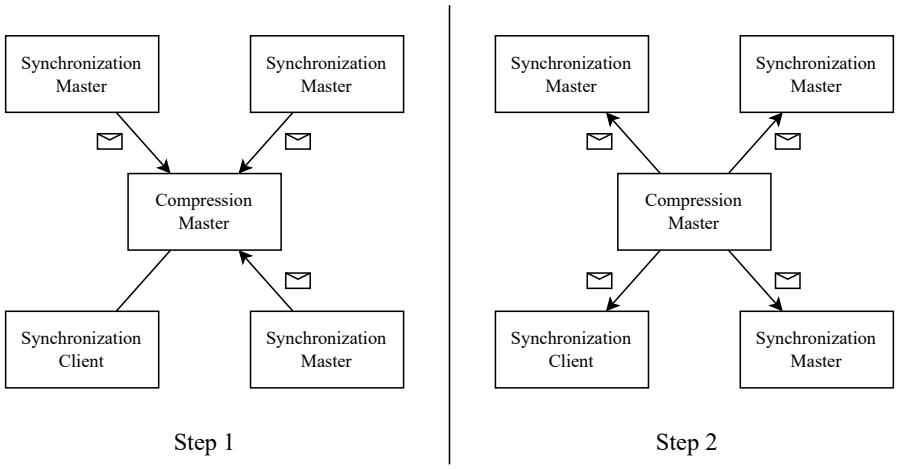

Figure 2.5: Time synchronization in two steps (derived from [Ade22])

When a network switch receives a frame, it first categorizes the frame according to its traffic class [TPS12]. If it is a TT frame, the switch checks whether the received frame is valid according to its receive schedule. If the frame arrived at the correct time, it is stored in a preassigned buffer, to be forwarded later as specified by the send schedule. RC traffic is instead buffered inside a queue, to be sent gradually as soon as there are sufficiently large gaps in the send schedule of the corresponding egress port. Then, if there are still gaps in which the switch is idle, BE traffic will fill the those gaps. See Figure 2.4 for an example of a switch forwarding mixed traffic.

2.3.3 Structure of a TTE Frame

Each TTE frame is also a valid Ethernet frame with a value of `0x88d7` in the EtherType field. The frame's payload has a separate TTE header prepended to the actual payload, containing additional metadata for time triggered communication specifically [KAGS05].

2.3.4 Time Synchronization

Since the switches in a TTE network have to receive and send frames at stringently defined times, they must use the same global time. A set of synchronized clocks is always susceptible to clock drift and will thus become more and more desynchronized if left unattended. If the network does not synchronize the clocks of all switches regularly, it becomes impossible for them to operate on a precise schedule. End devices that only send RC or even just BE traffic through the network do not need to be as stringently synchronized as devices that send TT traffic. Highly synchronized clocks also enable most other TTE features, like event chain reconstruction and taking network snapshots for diagnostic purposes.

To withstand Byzantine failures (i.e., some network components either fail or provide incorrect information), a TTE network can have multiple synchronization masters and compression masters [Ade22]. A network device that is neither is instead a synchronization client. Time synchronization happens

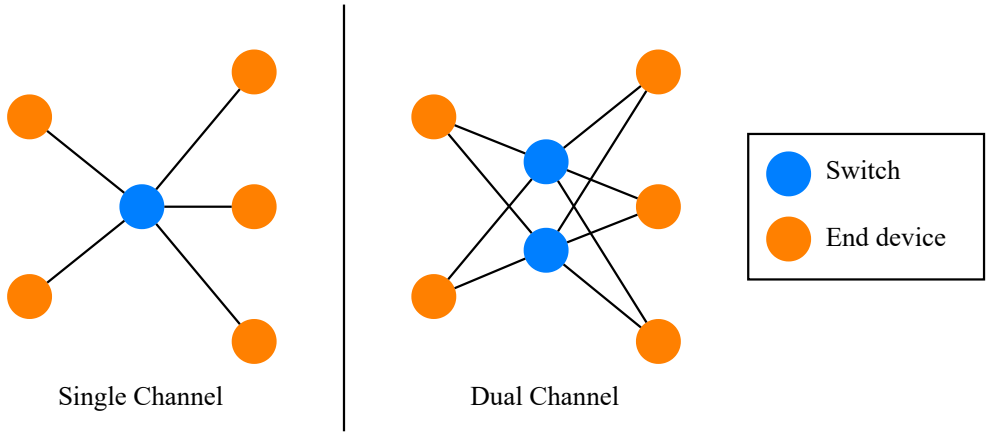

Figure 2.6: In a dual channel network, there is a second switch for each original switch, with a completely separate set of connections (derived from [Ade22]).

in two steps, also illustrated in Figure 2.5: In the first step, every synchronization master sends its current time to all compression masters. The compression masters then calculate the average of those times, which is now the new global time. In the second step, the compression masters distribute the new global time to every other device in the network which they then adopt, even the synchronization masters. If a synchronization master fails or provides implausible information, it is disregarded in the calculation of the new global time. If a compression master provides erroneous information, the network uses the majority consensus of the other compression masters instead.

2.3.5 Disruption handling

A network can experience disruptive failures, like a switch that sends frames slightly off schedule, hereby disrupting the switches directly connected to it, thus potentially compromising a large part of the network. To counteract this, TTE offers guardians, which are separate devices, each attached to a single switch [KAGS05]. The guardian receives a copy of each TT frame its switch sends and in case of erroneous behavior it can block the switches egress ports entirely, halting the propagation of an error state that could have otherwise spread throughout the network. Guardians only observe their switch's TT traffic and no other traffic.

For more redundancy, TTE offers multi-channel networks. In a dual channel network for example, there are twice as many switches and cables, essentially forming a second identical network on top of the original network, as seen in Figure 2.6. All the senders and receivers of the network are connected to both channels. Senders send their frames over both channels and receivers expect incoming frames on both channels [KAGS05]. If there is a fault in one channel, the receiver can use the frame from the other channel. If both channels work, the receiver drops the second identical frame and only passes the first one to its application. Note that BE traffic never makes use of redundancy and would e.g. only get passed through a single channel. As an alternative (or

for even more redundancy) a switch in a ring topology can forward a frame both clockwise and counterclockwise through the network. The switch receiving both frames then again drops the second one and only passes the first frame to the intended receiver.

2.4 Scheduling Fundamentals

We now introduce fundamental scheduling concepts and related terminology.

2.4.1 Networks and Flows

In the context of this thesis, a network is a collection of *end devices* and *switches*, connected together so that there is a path from any network device to any other network device. End devices want to send messages to other end devices in the network and switches only serve to forward those messages to help them reach their destination.

A *flow* (also often referred to as "stream") is a regular stream of messages that an end device wants to send to another end device. The messages to be sent are called *frames*. A flow always has the following static properties: one source end device, one destination end device, a frame sending period, and a frame size.

2.4.2 Time Management

There is a limit to how fast switches can process incoming frames. To formalize this physical limitation, we use the concept of a *macro tick*, a smallest possible time interval that all network devices can work with [COCS16]. Each time interval we use in our model must be a multiple of these macro ticks and is therefore discrete. For example, if we assume a macro tick of 1 µs, a time interval of 1.5 µs would not be allowed and would have to be rounded up to 2 µs to fit the macrotics.

A *schedule* is a finite set of instructions for all network devices. It is supposed to be repeated indefinitely until replaced by an updated schedule or until the network shuts down. A single loop of the network's global schedule is called a *hypercycle*. The duration of this schedule before it starts to loop is called the *hyper period*. The length of a hyper period is the least common multiple (lcm) of all flow periods. For example, if our schedule had flow periods of 80 µs, 120 µs, and 200 µs; then the hyper period would be 1,200 µs long.

We define the *subcycle period* as the greatest common divisor (gcd) of all flow periods. In our previous example, a subcycle period would be 40 µs long. The subcycle period determines the length of all subcycles. We assume that flow periods, the hyper period and the subcycle period are all in-phase, i.e., they all start at the same time. Figure 2.7 illustrates the special properties of hyper periods and subcycle periods by using our example scenario.

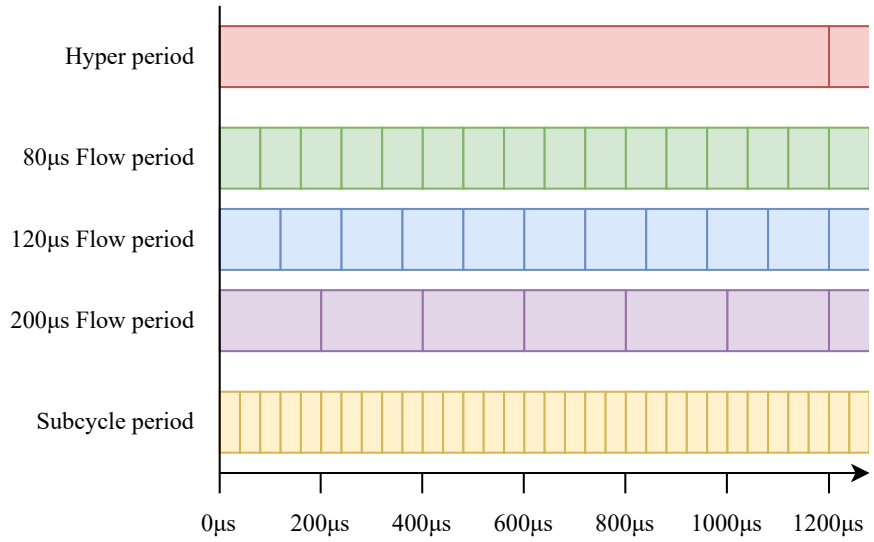

Figure 2.7: This diagram highlights the special properties of hyper periods and subcycle periods: When the hyper period ends, all flow periods end. When any flow period ends, the subcycle period ends.

2.4.3 Ports and Slots

The phrase "a flow being placed in the schedule" means that dedicated transmission slots are reserved for frames of this flow at every network link of their route, so that the schedule fully accommodates the stream of messages that the flow specifies. When the hyper period h is 2,000 µs and the flow period f is 500 µs, a frame has to travel from source to destination $h \div f = 4$ times. We call each of these sending instances a *strand*, so this flow has 4 strands per hypercycle. We always assume that hyper period and flow period are in-phase, i.e., whenever a hyper period ends, all flow periods end as well. This leads to the end of a hyper period always being a moment in which there are never any frames en route, which is an ideal time to transition to an updated schedule.

Each network node A that can send frames to a directly connected network node B has an *egress port* to B. When sending a strand, there may be different routes that the frame can take through the network. We call the route a strand takes its *configuration*. Concretely, a configuration is a list of egress ports that form a route from the flow's source to the flow's destination. In the context of this thesis, all strands of the same flow always use the same configuration.

When a flow is placed in the schedule, all egress ports that are part of its configuration must reserve time in which they send the flow's frame to the next node in the configuration. Each of these time reservations in an egress ports is called a *slot*. Slots may not overlap with other slots, which makes it increasingly difficult to place flows into highly utilized networks, i.e., networks whose global schedule accommodates a lot of flows already.

Incremental scheduling algorithms can modify schedules they have created at an earlier time. We count the times a scheduling algorithms is invoked; each of them is a *time step*. The first time we call our algorithm, is time step 0, the next time is time step 1 and so on. Before each time step,

one has to provide a list of flows that the schedule already accounts for but are no longer needed and/or a list of new flows to add to the schedule. All algorithms will remove unneeded flows first, by removing all slots from the schedule that are associated with those flows. Afterwards, when scheduling *defensively*, the remaining old slots from previous time steps must remain as they are and new slots can only be placed in unutilized gaps around them. When scheduling *offensively*, all remaining old slots may be moved as long as they still fulfill the requirements of their flow. No matter the scheduling strategy, all flows that the schedule already accounted for and that were not declared unneeded have to remain intact after a time step. Also, the algorithms may decide to only add a subset of the flows requested, though most algorithms described in Chapter 3 will never do this.

2.4.4 Delay and Jitter

There are different rules about placing individual slots into a schedule: When using *store-and-forward* switching a switch can only forward a frame as soon as it fully received it. When using *cut-through* switching a switch can forward a frame even when it only received and processed a the said frame's header (and not yet its body/payload).

In addition to that, there are delays to be considered when placing flows: The *transmission delay* is the time it takes for a network device to fully send a frame. The transmission delay depends on frame size and cable bandwidth; a slot is always as long as that frame's transmission delay. Then, the *propagation delay* is the time it takes for a signal to travel from one side of the cable to the other side. A switch has not received the full frame unless both transmission delay and propagation delay have passed. The *processing delay* is the time it takes for a switch to internally forward a message (or parts of a message it just received) from one of its ingress ports to one of its egress ports. When placing flows using store-and-forward switching, the slots of a message strand have to be at least as many ticks apart as the propagation delay plus the processing delay.

Jitter is the variance at which frames arrive at their destination. When there is no jitter, frames arrive at their destination in an unchanging interval. For example with a flow period of 1,000 µs the interval between frames being fully received would have to be always 1,000 µs. When there is jitter, the interval can vary. For example with a flow period of 1,000 µs the interval between frames being fully received could alternate between 500 µs and 1,500 µs without violating any of its flow's requirements. This kind of jitter which is caused by the schedule itself is called *reception jitter*. When practicing offensive scheduling, the switch from one schedule to a newer one can cause one-time *reconfiguration jitter*. Defensive scheduling avoids reconfiguration jitter altogether since it does not allow modifications of flows that have already been placed.

3 Related Work

In this thesis, we strive to find a fast running online scheduling algorithm that maximizes network throughput. While offline scheduling algorithms are used in advance before the network is in use, online scheduling algorithms must be able to manage a network that is already running: They have to be able to modify an existing schedule for when flows are added or removed. Also, new schedules, even for large networks, must be ready within seconds. Fulfilling these principles would make an algorithm suitable for plug-and-produce scenarios: It could incorporate new devices or flows into a running network within seconds and without disruption, reproducing the convenience of conventional, event-triggered networks.

Historically, online scheduling algorithms are preceded by offline scheduling algorithms. These assume that the flows will not change and schedules should be as optimal as possible, though the optimization criterion may vary. The deliberate tradeoff is a longer scheduling runtime: An offline scheduling algorithm may take hours or days to compose a schedule, which would not be acceptable for online scheduling algorithms.

Popular offline scheduling approaches include Integer Linear programming (ILP), Satisfiability Modulo Theories (SMT) and Constraint Programming (CP) [PRCS16] [SSN19] [VHT21]. Some approaches also consider that the shortest path through the network is not always the optimal one, so they optimize schedule and flow paths together [SDT+17]. Though even when only using shortest paths, scheduling is an NP-hard problem and therefore only suitable for small networks [DN16].

To also create schedules for large networks in a reasonable time frame, heuristics had to replace heavyweight optimizations. The tabu-search algorithm by Dürr and Nayak [DN16] for example can compose near-optimal schedules in less than 10 seconds, as long as there are no more than 50 flows to schedule. Sizeable scenarios with about 1500 flows need about 3.2 hours, which is an improvement compared to the algorithms discussed before, but still far too slow for plug-and-produce scenarios.

Online schedulers need to be even faster than that. This thesis primarily builds upon the research of Geppert et al., more precisely both their H2S and Cost-Efficient Lazy Forwarding (CELF) online scheduling algorithms [GDBR23]. H2S greedily schedules flows in one pass, placing flows individually in an order determined by two nested heuristics. The resulting schedules are not as optimized as offline scheduling algorithms would create them, but even schedules with tens of thousands of flows can be composed within a few seconds. We will discuss H2S in more detail in Section 5.1. CELF works similarly, the only difference being that flows are placed in an order determined by a score that not only considers the flow's properties, but also its potential route through the network. What sets these algorithms apart from most other online schedulers is their focus on aggregated network throughput as an optimization criterion as well as the option to reject individual flows that could not be scheduled. Most other algorithms are all-or-nothing: If they cannot incorporate a flow they either terminate with no result or escalate the problem to a (slower) offline scheduler.

A similar approach was taken by Raagaard et al., also greedily placing flows in one pass [RPGS17]. While H2S and CELF optimize aggregated network thoughput, Raagaard's goal was to have their TT schedule utilize the least amount of TSN queues to have as many queues as possible left for RC and BE traffic. Furthermore, their algorithm only considers a single route per flow and it would rather escalate to using an offline scheduler than reject unschedulable flows.

Vlk et al. contributed their so-called EPIC algorithm [VBHT22], which is similar to the algorithm of Raagaard et al., but it does not schedule in one pass: Instead of escalating when failing to schedule a flow, EPIC backjumps to a previous state, invalidating all placements in between and trying a different placement. Consequently, if there is a feasible scheduling solution, EPIC will find it eventually. Though as soon as the problem is no longer trivial, computation time will rise past what is acceptable for online schedulers.

Bujosa et al. proposed their HERMES algorithm, which also schedules greedily in one pass [BAP+22]. Their scheduler does not schedule flow by flow but egress port by egress port. Simply put: The other online schedulers first schedule one entire flow from source to destination, then the next entire flow from source to destination and so on. HERMES fully schedules everything for one egress port, then it schedules everything for the next egress port and so on. Egress ports at destination nodes are scheduled first, egress ports at source nodes last. Due to this ordering, HERMES can neither consider multiple routes for a flow, nor can it handle cyclic networks, which makes it unsuitable for many network topologies.

4 System Model and Problem Statement

We assume that our network is a connected, undirected graph consisting of switches and end devices. Every device can send and receive messages from a device it is directly connected to. All devices operate using the store-and-forward principle, though adapting the model to cut-through switching would be possible. The buffers of our switches are unbounded in size and allow for random access, i.e., buffered frames can be accessed in any order. It should be noted that while TTE allows random access, not all TTN standards do.

Each flow has the following properties:

- The *source* is the end device that is sending the frames.
- The *destination* is the end device that is supposed to receive the frames.
- The *period* describes the interval at which new frames become available for sending. E.g. if the period is 500 µs, then the source attempts to send 2,000 frames per second to the destination.
- The *frame size* is the size of each individual frame in bytes. All frames belonging to the same flow have the same size, though the source could simply use padding if the frame to be sent would otherwise be smaller.
- Finally, every flow has an unique numerical *id* to clearly identify it. This is needed since two flows with otherwise exactly identical properties may exist.

We define deadline of each strand to be equal to its period, i.e., when a new frame becomes available, the previous frame must have already arrived at the destination. We assume that flow periods are chosen in a way that the resulting hyper period is within reasonable limits, not more than a few milliseconds for example. All network devices must be able to store and swiftly access their own part of the schedule, which grows larger as the hyper period becomes longer. The resulting subcycle period should also be reasonable since very small subcycles have a negative impact on scheduling algorithm performance. Ideally, the chosen flow periods are harmonic, i.e., multiples of each other, though this is not required. Harmonic periods are widely used in several different industrial scenarios already [MNX+18].

In our model, both reception and reconfiguration jitter are unbounded, although the aforementioned deadline imposes a theoretical maximum on both kinds of jitter. While we do not attempt to optimize jitter in this thesis, we still keep track of reception jitter in our evaluations.

The scheduling algorithms we cover are incremental, they can either create a new schedule from scratch or modify an existing schedule by adding and/or removing flows from it. Each request to add and/or remove flows is a single time step and the scheduling algorithm may decide to only add a

subset of schedules (as described in Section 2.4.3) The algorithms typically have a runtime of only a few seconds and in a running network they could be run periodically or on demand, for example every time a new device is added to the network.

We assume that the schedules are generated and distributed by a central entity that has a global view on the network. When schedules change, all network devices execute the update at the exact same time, at the end of a predetermined hypercycle. How updated schedules are distributed throughout the network and how the network devices synchronize their clocks to be able to accurately follow these schedules is out-of-scope of this thesis. Possible practical solutions would be utilizing separate channels for this purpose, piggybacking control metadata on frames that are sent through the network anyway, or scheduling flows specifically for control frames. Network security and failure tolerance are also out-of-scope.

We primarily aim to optimize the aggregated network throughput of the computed schedules. The throughput of an individual flow is its frame size divided by its period. Aggregated network throughput is the total throughput of all flows our schedule currently accommodates. In contrast to related approaches, which optimize for the highest number of accommodated flows, this metric does not discriminate against flows with high frame sizes and/or low periods. Despite that, we still consider accommodated flows as a metric in our evaluations to allow comparisons with other algorithm evaluations that do not analyze aggregated throughput. Secondary metrics that we also take into account are algorithm runtime, reception jitter, and maximum buffer utilization.

Incremental scheduling algorithms are supposed to adapt schedules they created and are therefore incentivized to create schedules that already have future flexibility in mind. Since we cannot know how many times our scheduler would be actually invoked in practice, we need it to perform well at each invocation. Therefore, we do not only consider the aggregated network throughput after the *last* time step of our evaluation scenarios, but after *every* time step.

5 Algorithm Descriptions

We now introduce the H2S and CELF algorithms by Geppert et al. [GDBR23]. Both are incremental scheduling algorithms, meaning that they can modify a schedule they generated earlier to add and/or remove flows. We will later find improvements for these algorithms (see Chapter 6) and evaluate them (see Chapter 7).

Every time either algorithm is invoked (every *time step*), one has to provide parameters as to which flows are to be removed and what flows are to be added. Both algorithms will always remove flows first before placing new ones by simply clearing all slots in the schedule that are occupied by the to-be-removed flows. Then they will attempt to place as many of the new flows into the schedule as they are able to, without reorganizing any slots that were already there when the algorithm was invoked. The precise placement procedures are described in Sections 5.1 and 5.2. If either algorithm is not able to place all flows that were supposed to be added, they proceed as per their planning strategy:

If they are using the *defensive strategy*, they will never attempt to reorganize preexisting flows and simply submit the incomplete scheduling without all new flows. If they are using the *offensive strategy*, they take a so-called offensive step: They experimentally clear the entire schedule and then try to place all flows, both preexisting and new, into the schedule from scratch. If they are unable to place all preexisting flows into the schedule, then the offensive step fails and they fall back to their defensive solution, as if they had never taken an offensive step at all. Otherwise, the admitted flows of the offensive approach and the defensive approach are compared; whichever schedule achieves higher aggregated network throughput is submitted.

5.1 Hierarchical Heuristic Scheduling (H2S)

H2S is a greedy one pass scheduling algorithm that sorts all flows to be scheduled and then attempts to schedule them in that order. Flows are scheduled in an order determined by this hierarchical flow sorting heuristic:

1. Schedule low period flows first (low period streams are more likely to not be schedulable because they tend to require the most free slots; so we schedule them while there are still the most free slots available)
2. Schedule high frame size flows first (smaller frames are more likely to fit into gaps)
3. Schedule flows with the lowest ID first (definitive tiebreaker)

When scheduling a flow, there are often multiple routes that its frames can take through network to reach their destination. We call these candidate routes *configurations*. The algorithm that provides these configurations is outlined in Section 5.3. The configurations are then sorted in ascending order by their best case delay, which is the least amount of hops in networks with uniform propagation delay. The algorithm then attempts to place the flow into the schedule by using the first configuration in the list. It uses a balanced placement strategy for this, which is further discussed in Section 6.1. If placing the flow fails, it retries with the next configuration from the list. If placing the flow fails for all configurations, the algorithm gives up on that flow and proceeds trying to schedule the next flow in the list instead.

5.2 Cost-Efficient Lazy-Forwarding (CELF)

CELF is also a greedy one pass scheduling algorithm. Instead of sorting flows and configurations separately, CELF uses a combined configuration and flow sorting heuristic. When attempting to place a flow, H2S is willing to settle for very inefficient configurations (i.e., costly routes through the network). At the cost of a higher algorithm runtime, CELF attempts to avoid inefficient configurations, hoping to be able to schedule more flows with efficient configurations instead.

CELF starts by calculating configurations for each flow to be scheduled (using the algorithm described in Section 5.3) and ranks all configuration-flow pairs using the following comparison function:

1. Schedule pairs with low period flows first (same reasoning as before)
2. Sum the utilization percentage of all egress ports that the configuration uses and schedule pairs with the lowest total first (Configurations with less utilized egress ports are preferred. Short paths will have less percentages to sum and are therefore favored as well)
3. Schedule pairs with the lowest ID first (definitive tiebreaker)

Pairs are then scheduled in this order. Each flow is only placed once; when a flow is placed, other pairs that contain this flow are discarded. Again, the flow placement strategy in Section 6.1 is used when placing flows.

5.3 Generating Configurations

We generate up to 5 configurations per flow using a modified version of Dijkstra's algorithm by Geppert et al. [GDBR23]: Initially, all edges in the network graph have a weight of 1. We repeatedly use Dijkstra to calculate the shortest path through the network and add it to our set of configurations; duplicate configurations are discarded. Each time we generate a configuration, even if it is a duplicate, we add 2 to the weight of every edge that is part of the configuration. This repeats until either 5 unique configurations have been found or 10 discarded duplicates, whichever happens first.

In this context, Yen's k-shortest path algorithm [Yen71] is commonly used instead since there are no shorter paths than the ones it calculates. The main drawback however is that the generated paths can be very similar, often leading though network links that are already highly utilized [GDBR23]. The paths generated by our algorithm use the same network links less often, favoring completely alternate routes instead.

6 Proposed Modifications

Firstly, we are introducing the slot placement algorithm H2S and CELF are using, which were developed by Geppert et al. [GDBR23]. Then, we discuss the modifications we developed in the scope of this thesis, namely relaxed jitter, backward rifts, and gap closing.

6.1 Original Placement Algorithm

When H2S and CELF try to place a flow, they use a so-called balanced placement strategy. Let us start with a simple example, also illustrated in Figure 6.1: In this configuration, each frame of our flow needs to take three hops to get from its source to its destination. Our flow's period is 1,000 µs and the hyper period 2,000 µs. This means there are two flow periods within a single hyper period:

- The first flow period starts at the 0 µs mark and ends on the 1,000 µs mark.
- The second flow period starts at the 1,000 µs mark and ends on the 2,000 µs mark.

We have 500 µs subcycles, which means there are 2 subcycles per flow period, depicted in red and blue in Figure 6.1. The placement algorithm now tries to reserve a strand of slots at the start of each subcycle "as soon as possible" (as early as possible in the subcycle). The usual placement rules still apply: Slots may not overlap, all slots of a strand must stay within their respective flow period, and a switch cannot forward a frame through an egress port unless it fully received and processed that

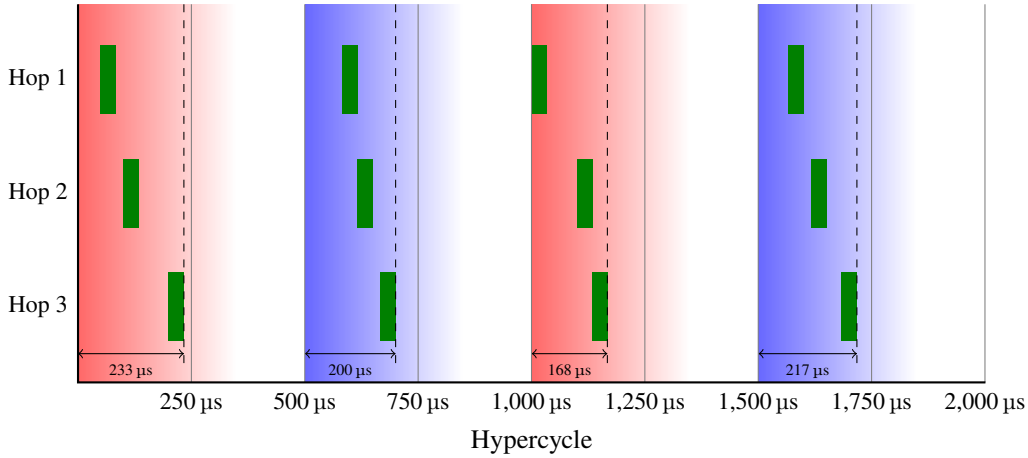

Figure 6.1: Balanced flow placement example.

Algorithm 6.1 Original placement algorithm

Input: h_l, s_l, c, f_p, f_v
$s_n \leftarrow f_p \div s_l$ // Subcycles per flow period
$f_n \leftarrow h_l \div f_p$ // Flow periods per hyper period
for $s_i = 0 \rightarrow s_n - 1$ **do**
    for $h_i = 0 \rightarrow f_n - 1$ **do**
        $o \leftarrow s_i \times s_l + h_i \times f_p$ // Offset for this subcycle
        $p' \leftarrow \text{STO}(c, o, f_p, f_v)$ // Possible scheduling for this strand ($\emptyset$ if no placement found)
        $p[s_i][h_i] \leftarrow p'$ // Store all schedulings
    end for
end for
for $s_i = 0 \rightarrow s_n - 1$ **do**
    if $p[s_i]$ contains at least one $\emptyset$ **then**
        $p_{delay}[s_i] \leftarrow \infty$
    else
        $p_{delay}[s_i] \leftarrow$ the maximum end-to-end delay of all strands in $p[s_i]$
    end if
end for
if $\forall k \in p_{delay} : k = \infty$ **then**
    return $\emptyset$ // No valid scheduling found
else
    Out of all $s_i \in [0, s_n - 1]$, choose s_i so that $p_{delay}[s_i]$ is minimal and **return** $p[s_i]$
end if

frame through one of its ingress ports. Considering this, the algorithm managed to find placements for all 4 subcycles, marked here in green. We now add up the total end-to-end delay of each frame strand.

- In the red subcycles, the first frame was received after 233 µs and the second frame after 168 µs, for a maximum of 233 µs.
- In the blue subcycles, the first frame was received after 200 µs and the second frame after 217 µs, for a maximum of 217 µs.

We prefer placements with the least end-to-end delay, so we schedule the flow into the blue subcycles, discarding the red placements. Note that if placing any of the slots in the red subcycles would have failed, red would be ineligible to be chosen and blue would be chosen instead, regardless of blue's total delay total.

The textual description of the original placement algorithm is as follows: To place a flow with a given configuration into a schedule, first divide the hypercycle into equal segments, each as long as the flow period. Then reserve a strand of slots as soon as possible (ASAP) at the start of every subcycle, following the usual slot placement rules (no slot overlaps, respect transmission and propagation delays). Now calculate the end-to-end delay of every strand, always starting at the beginning of each respective subcycle. Then calculate the maximum end-to-end delay of every n-th subcycle across all strands. The n-th subcycle with the lowest maximum end-to-end delay is chosen, all other slot reservations are discarded.

To express all this in a more formal manner, Algorithm 6.1 describes the pseudocode of the placement process. It either returns a valid scheduling for the flow or $\emptyset$ if it cannot find a valid scheduling. The parameters and functions that Algorithm 6.1 uses are as follows:

- h_l := length of a hypercycle (is 2,000 µs in previous example)
- s_l := length of a subcycle (is 500 µs in example)
- c := configuration to use, i.e., the exact path each frame takes through the network (example uses a path with three hops)
- f_p := the flow's period, i.e., the interval at which a new frame becomes available. Also doubles as the deadline for each frame; so as soon as a new frame becomes available, the previous frame expires. (f_p = 1,000 µs in example)
- f_v := the volume of each frame, i.e., their size in bytes
- STO(c, o, f_p, f_v) searches a valid scheduling for a single strand using configuration c, o as the tick where the subcycle starts, f_p as the flow period length, and f_v as the frame volume. Slots are placed ASAP.

6.2 Relaxed Jitter

When examining the original placement algorithm, one might ask themselves why we are always using the same subcycle for each flow period and not different subcycles. In other words, using our previous example: Why do we only accept placements that use only red subcycles or only blue subcycles? The overall end-to-end delay would be lower if we would use the blue subcycle for the first flow period and the red subcycle for the red flow period.

The drawback is higher reception jitter: Even though there is a new frame available at the source node every 1,000 µs, the interval at which the destination node receives those frames would alternate between 468 µs and 1,532 µs. Despite not violating any deadlines, this might still be an undesirable property. But since reduced reception jitter is of lower importance, we can easily adapt the algorithm to allow for using separate subcycles as we described.

6.3 Backward Rifts

Our main goal is to increase aggregated network throughput. Figure 6.2 illustrates the main problem: In the upper diagram, a vertical line always denotes a transition between subcycles. There is a lot of unutilized space right at the end of most subcycles, which we now want to utilize as well. Scheduling low period flows is a lot more restrictive than scheduling high period flows; we can use the space more effectively by placing frames of high period flows in these spaces where low period flows cannot be placed. The bottom diagram of Figure 6.2 illustrates this idea well: There we can see an intense concentration of high period flows around most subcycle transitions.

One idea to accomplish this is the use of forward and backward rifts. "Forward rift" is a new term for the subcycles in the original placement algorithm: Each beginning of a subcycle has a forward rift; slots are placed ASAP on its right side (into the future). Backward rifts follow the same idea,

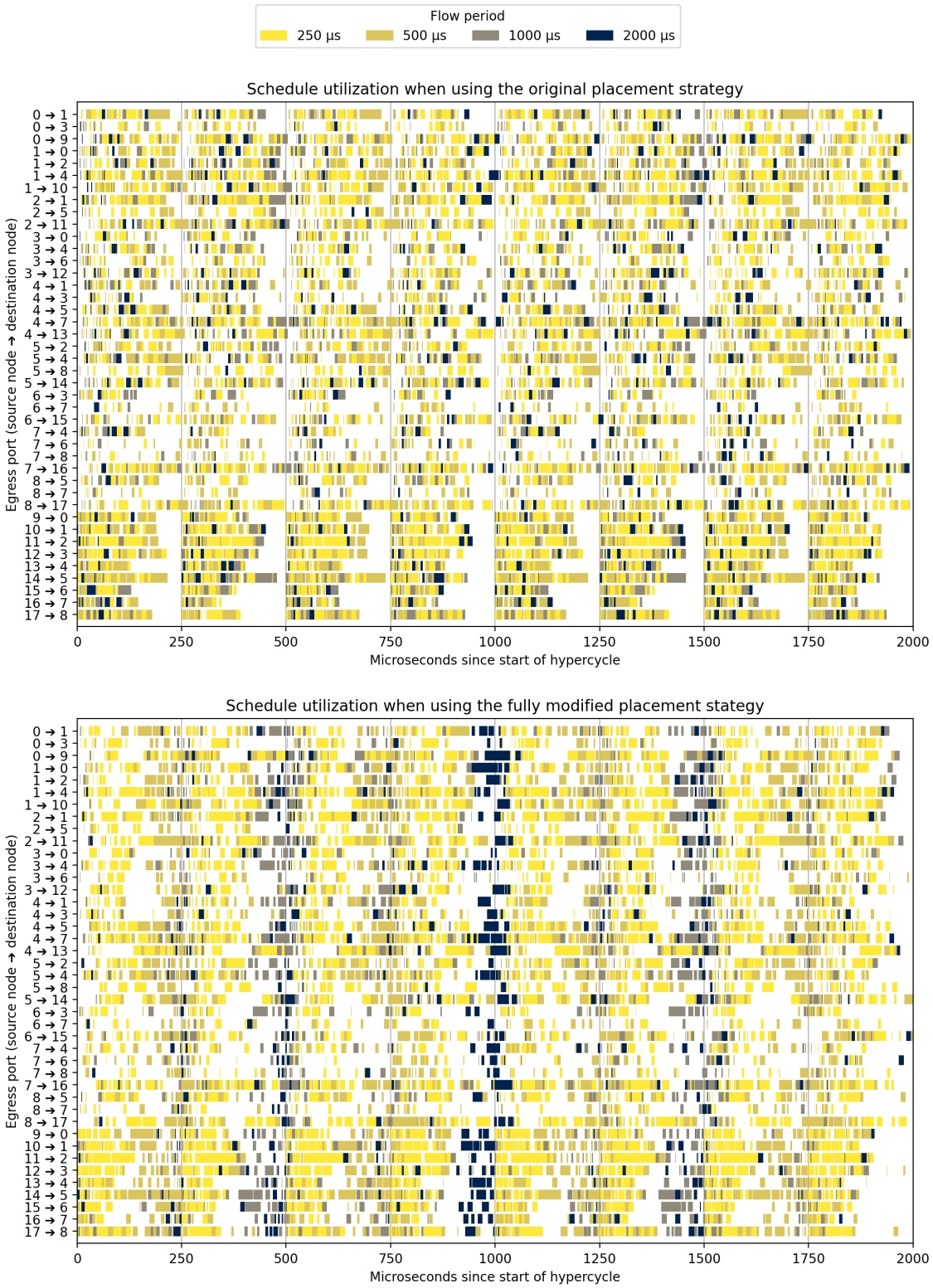

Figure 6.2: Schedule utilization on a small 3×3 grid network. Node ids below 9 are for switches, all other ids are for end devices. The bottom diagram was generated using the "between, one jump, 1.1, use gapclosing" parameter set from Section 7.2

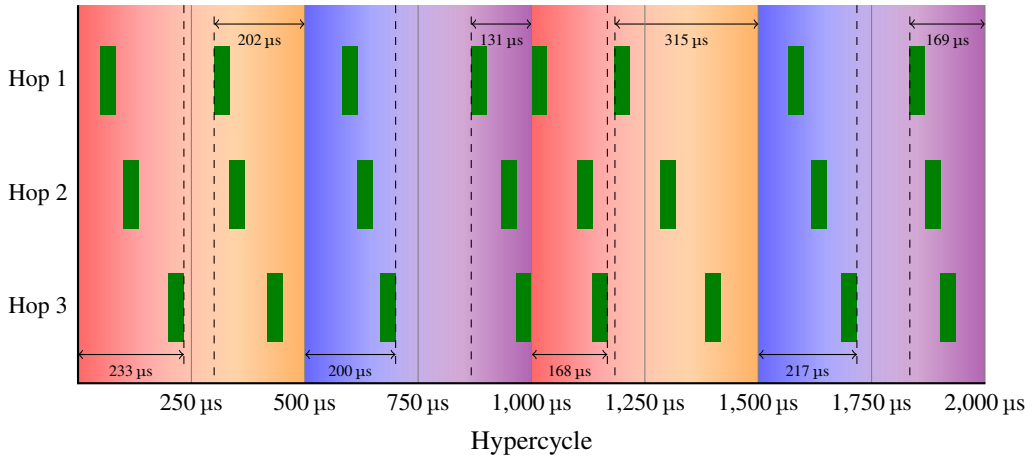

Figure 6.3: Placement example with backward rifts.

but in reverse: They are situated at the end of each subcycle; slots are placed as late as possible (ALAP) on its left side (into the past). To calculate the end-to-end delay in this case, we use the time period between the backward rift and the start time at which this strand's first frame is sent.

Another example should illustrate this idea, as seen in figure Figure 6.3. Same as before, the hyper period is 2,000 µs long, the subcycles are 500 µs long, and the flow period is 1,000 µs long. The red and blue subcycle markings are still here, these are now called forward rifts. But in addition to that, we now also have orange and violet markings at the end of each subcycle, these are backward rifts. Forward and backward rift placements can freely bleed into each other, slot placements are only illegal if they exceed the limits of their flow period, not the subcycle limits. At forward rifts, slots are placed ASAP, at backward rifts slots are placed ALAP. We then calculate the transmission delay for each rift. Jitter does not matter as much here, so we simply choose the rift with minimal transmission delay in each flow period. We end up choosing the violet rift in flow period 1 and the red rift in flow period 2.

6.3.1 Placement

We thought of different strategies for placing backward rifts:

- **Everywhere:** Same as in the example: Backward rifts are placed at the end of every subcycle.
- **Between:** Same as *everywhere*, but do not place backward rifts at the final subcycle of a flow period. This means that lower flow periods also have less backward rifts. The hoped-for advantage is that higher period flows are more likely to make use of backward rifts and hence favor utilizing space that low period flows cannot utilize due to their tighter deadlines.
- **Between & Final:** Same as *between*, but always place a backward rift at the end of the hypercycle. The regular *between* strategy would not utilize the end of the hypercycle, hence we make an exception for it.

Refer to Figure 6.4 for an example.

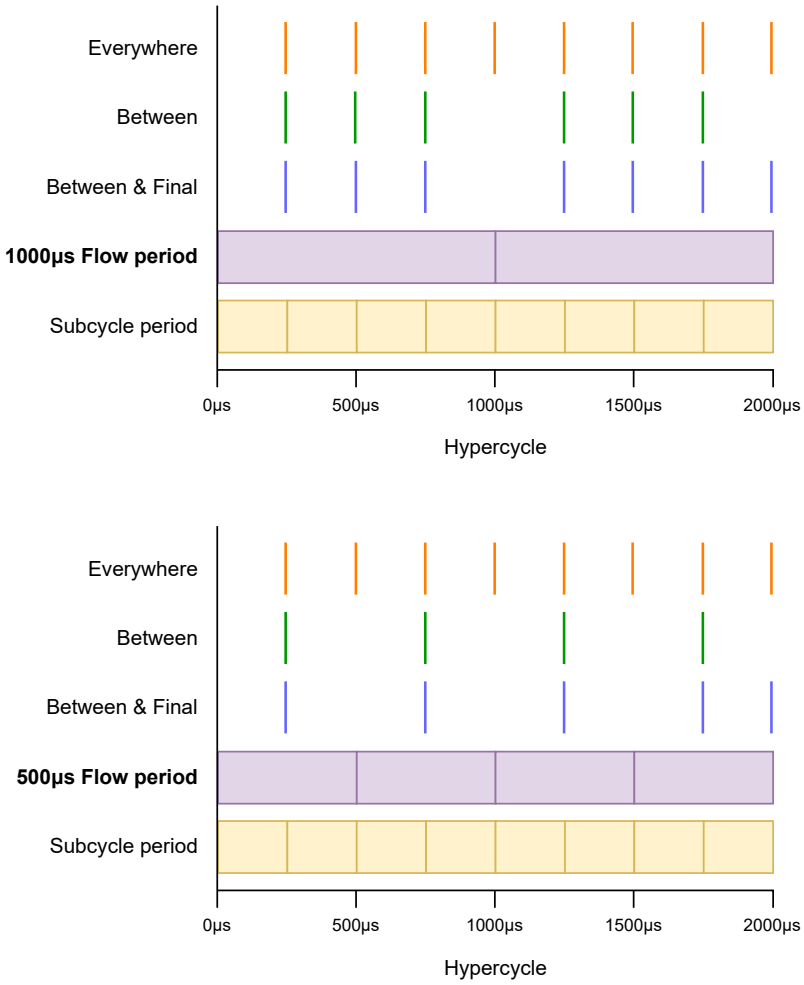

Figure 6.4: Example of where the different placement strategies would place their backward rifts, with the subcycle period being 250 µs in this case. *Everywhere* always places as many backward rifts as there are subcycles per hypercycle. *Between* and *between & final* place less backward rifts when the flow period is lower.

6.3.2 Shifting

Our goal with backward rifts was to utilize the space that the original placement strategy could not utilize: the left and right side of the subcycle borders. When placing the backward rift *at* the subcycle border, the ALAP placement only covers the left side of the border but not the right side. Therefore, as long as we do not shift backward rifts beyond deadline limits, utilization should increase. Here are different shifting strategies we came up with:

- **Border:** The backward rift simply remains at the subcycle border, where the backward rift placement strategy originally placed it.

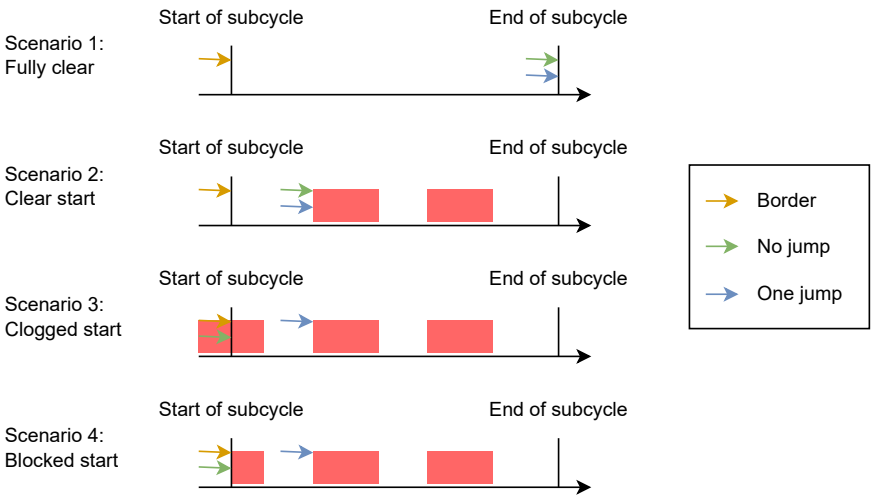

Figure 6.5: Where the different shifting strategies would place their backward rift. The red rectangles are slots that are already reserved on the egress port which lead to the flow's destination node.

- **No jump:** If not at the end of a flow period, shift the backward rift to the right until it collides with an already reserved slot at the egress port of the last hop. Also stop shifting when colliding with the next subcycle border.
- **One jump:** Same as *no jump* but if the tick immediately after the subcycle border is already reserved (i.e., if the backward rift would not shift at all), then shift to the first free tick and then shift further to the first reserved tick after that. Similarly to *no jump*, always stop shifting when colliding with the next subcycle border.

Figure 6.5 demonstrates the what the different strategies do in several scenarios while Algorithm 6.2 describes the shifting process more formally with the following parameters:

- f_p := The flow period of the flow to be placed
- c := The configuration to place the flow with
- s_1 := The first tick of the subcycle at which to place the backward rift
- s_2 := The first tick of the subcycle that comes right after the subcycle of s_1

6.3.3 Cost Multiplier

We can also apply a multiplier to the end-to-end delays of backward rifts to make them more or less attractive compared to forward rifts. Experimenting with this new parameter may have desirable results since we predict that backward rifts perform better if other slots were placed in forward rifts already.

Algorithm 6.2 Backward rift shifting algorithm

```
1:  Input: f_p, c, s_1, s_2
2:  if using border strategy or ∃n ∈ ℕ^0 : n × f_p = s_1 then
3:      return s_1
4:  end if
5:  for every unutilized space u in the last hop egress port of c do
6:      if Using no jump strategy then
7:          if the tick right after u is ≥ s_1 then
8:              if the tick at the start of u is ≤ s_1 then
9:                  return min (s_2, the tick right after u)
10:             else
11:                 return s_1
12:             end if
13:         end if
14:     else                                          // Using one jump strategy
15:         if the tick right after u is > s_1 then
16:             return min (s_2, the tick right after u)
17:         end if
18:     end if
19: end for
20: return s_1
```

6.4 Gap Closing

Our final improvement idea is based on the concept of defragmentation. Though instead of rearranging already placed slots offensively, we attempt to close gaps between slots preventively: After the placement algorithm has made a decision on which slots are going to be reserved for a flow, an additional algorithm looks at the result once more to close gaps between the newly placed slots and slots that were already reserved beforehand. It iterates through each string of slots from destination to source, shifting them to the right (into the future) if it can place them next to another slot without violating any deadlines or rules. It will not shift a slot if it is already touching another slot or the hypercycle border.

Figure 6.6 provides examples how gap closing works and Algorithm 6.3 describes the process more formally; parameter r is the slots the placement algorithm has decided to reserve to place the flow into the schedule.

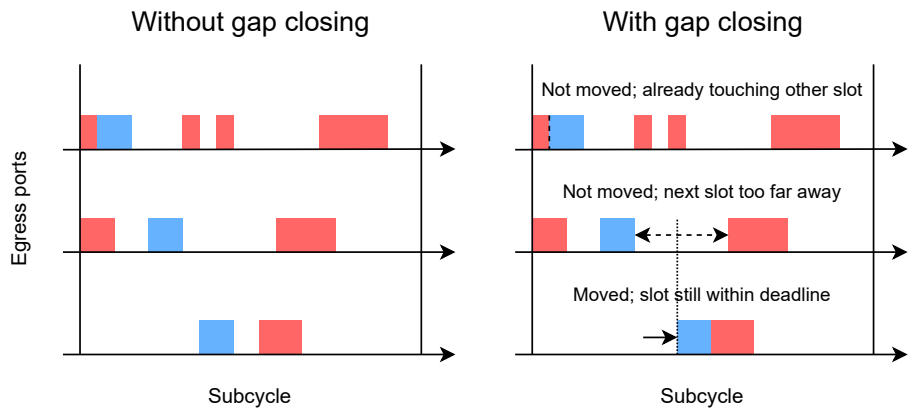

Figure 6.6: Example of gap closing. Blue are the slots to be placed, red slots are already reserved. Note that the algorithm is applied from bottom to top! On the second egress port, moving the slot past the dotted line would mean that the next switch has to send the frame before receiving it.

Algorithm 6.3 Gap closing algorithm

```
1:  Input: r
2:  for every strand t in r do
3:      d′ ← deadline of t minus propagation delay                    // Running deadline
4:      for every slot in s in t, from destination to source do
5:          if s touches another slot on its egress port or the hypercycle border then
6:              pass
7:          else
8:              if there is free space to the right of s and moving s to the very right of that free
    space would not violate d′ then
9:                  Move s to the very right of the free space
10:             end if
11:         end if
12:         d′ ← starting tick of s minus propagation and processing delay
13:     end for
14: end for
15: return r
```

Algorithm 6.4 Placement algorithm with all proposed modifications

Input: h_l, s_l, c, f_p, f_v
$s_n \leftarrow f_p \div s_l$ // Subcycles per flow period
$f_n \leftarrow h_l \div f_p$ // Flow periods per hyper period
if using *everywhere* backward rift placement **then**
    $r_n \leftarrow 2 * s_n$ // Number of rifts per flow period
else
    $r_n \leftarrow 2 * s_n - 1$ // Number of rifts per flow period
end if
for $h_i = 0 \rightarrow f_n - 1$ **do** // Iterate over all flow periods
    $p \leftarrow \emptyset$
    $p_{delay} \leftarrow \infty$
    if $h_i = f_n - 1$ and using *between & final* backward rift placement **then**
        $r_n \leftarrow r_n + 1$ // Use one more rift on the final flow period
    end if
    for $s_i = 0 \rightarrow (r_n - 1)$ **do** // Iterate over all rifts in flow period
        if $s_i \equiv 0 \mod 2$ **then** // *Forward rift:*
            $o \leftarrow (s_i \div 2) \times s_l + h_i \times f_p$ // Offset for this forward rift
            $p' \leftarrow \text{STO}(c, o, f_p, f_v)$ // Schedule ASAP ($\emptyset$ if no placement found)
            $m \leftarrow 1$
        else // *Backward rift:*
            $o \leftarrow ((s_i + 1) \div 2) \times s_l + h_i \times f_p$ // End of the subcycle
            $o' \leftarrow \text{SHIFT}(f, c, o, o + s_l)$ // Offset for this backward rift
            $a \leftarrow h_i \times f_p$ // Earliest availability time
            $p' \leftarrow \text{STOR}(c, o', a, f_v)$ // Schedule ALAP ($\emptyset$ if no placement found)
            $m \leftarrow$ Backward rift cost multiplier
        end if
        if $p' = \emptyset$ **then**
            continue
        end if
        $p'_{delay} \leftarrow$ transmission delay of p'
        $p'_{delay} \leftarrow m \times p'_{delay}$ // Apply cost multiplier
        if $p'_{delay} < p_{delay}$ **then** // Replace the old placement if this one is better
            $p_{delay} \leftarrow p'_{delay}$
            $p \leftarrow p'$
        end if
    end for
    if $p_{delay} = \infty$ **then**
        return $\emptyset$ // No placement found for this flow period
    else
        $r[h_i] = p$ // Store this flow period's best placement
    end if
end for
$r \leftarrow \text{GAPCLOSE}(r)$
return r // Return the best placements of each period

6.5 Fully Modified Placement Algorithm

Algorithm 6.4 is the complete placement algorithm which incorporates all modifications mentioned above. It either returns a valid scheduling for the flow or $\emptyset$ if it cannot find a valid scheduling. The parameters and functions that Algorithm 6.1 uses are as follows:

- h_l := length of a hypercycle
- s_l := length of a subcycle
- c := configuration to use, i.e., the exact path each frame takes through the network (example uses a path with three hops)
- f_p := the flow's period, i.e., the interval at which a new frame becomes available. Also doubles as the deadline for each frame; so as soon as a new frame becomes available, the previous frame expires.
- f_v := the volume of each frame, i.e., their size in bytes
- $\textsc{sto}(c, o, f_p, f_v)$ searches a valid scheduling for a single strand using configuration c, o as the tick where the subcycle starts, f_p as the flow period length, and f_v as the frame volume. Slots are placed ASAP.
- $\textsc{stor}(c, o, a, f_v)$ searches a valid scheduling for a single strand using configuration c, o as the backward rift's tick, a as the tick this strand becomes available at, and f_v as the frame volume. Slots are placed ALAP.
- $\textsc{shift}(f_p, c, s_1, s_2)$ is the backward rift shifting algorithm: Algorithm 6.2
- $\textsc{gapclose}(r)$ is the gap closing algorithm: Algorithm 6.3

To briefly summarize Algorithm 6.1: The loop on line 9 iterates over all flow periods and the nested loop on line 15 iterates over all rifts within each flow period. Inside the inner loop an if-else block (starting on lines 16 and 20) handles forward rifts and backward rifts respectively. After a placement has been found it is evaluated, the multiplier is applied, and the placement is only kept if it has a better end-to-end delay than all placements before it (line 32). After finding the best placements for every flow period, the gap closing algorithm has a chance to adjust the final result (line 43) before it is submitted.

7 Evaluation

In this section, we evaluate the performance and effectiveness our proposed modifications. Firstly, we describe our simulation setup. Then, we conduct experiments to determine the most effective parameter set for our modifications. Finally, we evaluate the resulting algorithms in a variety of scenarios.

7.1 Setup Description

In our simulations, a macro tick is always 1 µs long like in [COCS16]. The switches in our simulated network all use the store-and-forward principle and have a processing delay of 4 µs [DK+14]. Every link between two connected network devices is bidirectional and both directions have a bandwidth of 1 Gbit s^{-1} with a propagation delay of 1 µs [FGD+22].

Our scenarios are generated randomly; each flow's period is an evenly random value in 250 µs $\times$ $\{1, 2, 4, 8\}$, making them harmonic. Harmonic periods are commonly used in several industrial scenarios [MNX+18]. Each flow's frame size is an evenly random value in 125 bit$\times\{1, 2, 4, 6, 8, 12\}$, making every transmission delay perfectly fit an integer number of macro tics (e.g. with our 1 Gbit s^{-1} bandwith, a 125 bit frame takes 1 µs to transmit). A flow's source and destination end device are always random, though source and destination are never identical.

Additionally, unless mentioned otherwise, an evaluation scenario has these default properties: The network is a 5×5 grid of switches, with a single end device connected to every switch. The scenario has 50 timesteps, ranging from 0 to 49. In timestep 0, 1,500 random flows are added to the schedule. Every subsequent time step removes 100 random flows and adds 200 random flows. Each one of our evaluations uses 10 different scenarios that were randomly generated as described. When using randomly generated topologies, we generate 5 of them and run each scenario on each topology, resulting in 50 topology-scenario combinations per algorithm. Grid topologies however are predefined: There is only 1 grid topology for every size, so evaluations involving grids contain only 10 topology-scenario combinations per algorithm. The metrics we gather during our simulations are always averaged over all topology-scenario combinations.

We ran our simulations on an Ubuntu 20.04 machine with two AMD EPYC 7401 24-Core processors and 256 GB of primary memory. The algorithms and simulation environment were implemented in C++20 and ran on a single thread.

7.2 Parameter Experiments

Before evaluating our modified placement algorithm, we need to decide on the set of parameters our modified algorithms are going to have. We already introduced these parameters in Chapter 6, but here is a brief summary of them:

- **Backward rift placement** which determines where and how many backward rifts are placed. Can be set to *everywhere*, *between*, or *between & final*. Their explanations are in Section 6.3.1.
- **Backward rift shifting** which determines how each backward rift is shifted to the right after it has been placed. Can be set to *border*, *no jump*, or *one jump*. Their explanations are in Section 6.3.2
- **Backward rift cost multiplier** which determines how strongly forward rifts are favored over backward rifts. Can be set to *any non-negative floating point value*. This parameter was explained in Section 6.3.3. After some explorative testing, we decided to keep the value around 1.0 to avoid exaggerated effects; concretely we will set the multiplier to one of these values: $\{0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 2.0\}$
- **Gap closing** which performs preventive, defensive defragmentation. Can be turned *on* or *off*. This modification was explained in Section 6.4

We tested all 216 possible parameter combinations with the H2S algorithm, plus an unmodified variant of H2S. Figure 7.1 shows the results of this experiment, ranking all of them by their total aggregated throughput across all timesteps and emphasizing the top 9 ones. The entire top 9 already achieves substantially more throughput than the unmodified H2S algorithm. They are all using the "between, one jump, use gapclosing" parameter set, with only the multiplier parameter varying, though 1.1 is the best.

Noticeably, from timestep 37 up until the very end a wholly other set of parameters seems to achieve a higher throughput. Hence, we created an alternate selection of 9 algorithms, same as before, but ranking them only by the throughput achieved from timestep 37 onwards. Figure 7.2 shows the results. These alternate parameter sets all use *between & final* backward rift placement instead of just *between*. They perform substantially worse in the first 20 timesteps than our top 9. Instead they seem to be most useful in "cramming" scenarios, where they need to admit flows to a schedule that is already highly utilized. However, we do not expect cramming to be a common use case since a highly utilized network will reject a lot of new flows; even if the number of rejected flows is slightly lower, many applications require a whole set of admitted flows to properly function and even one rejected flow can render the whole application unusable.

Hence we only consider the top 9 as candidates and choose their best, "between, one jump, 1.1, use gapclosing", as the parameter set for our modified algorithms in the upcoming evaluation.

7.2.1 Single-Timestep Scenarios

We first and foremost want to develop a good incremental scheduling algorithm. Therefore, most of our evaluation scenarios use multiple timesteps in order to test the algorithms for their ability to schedule in a future-proof manner. Though of course, one can also provide all flows to be scheduled

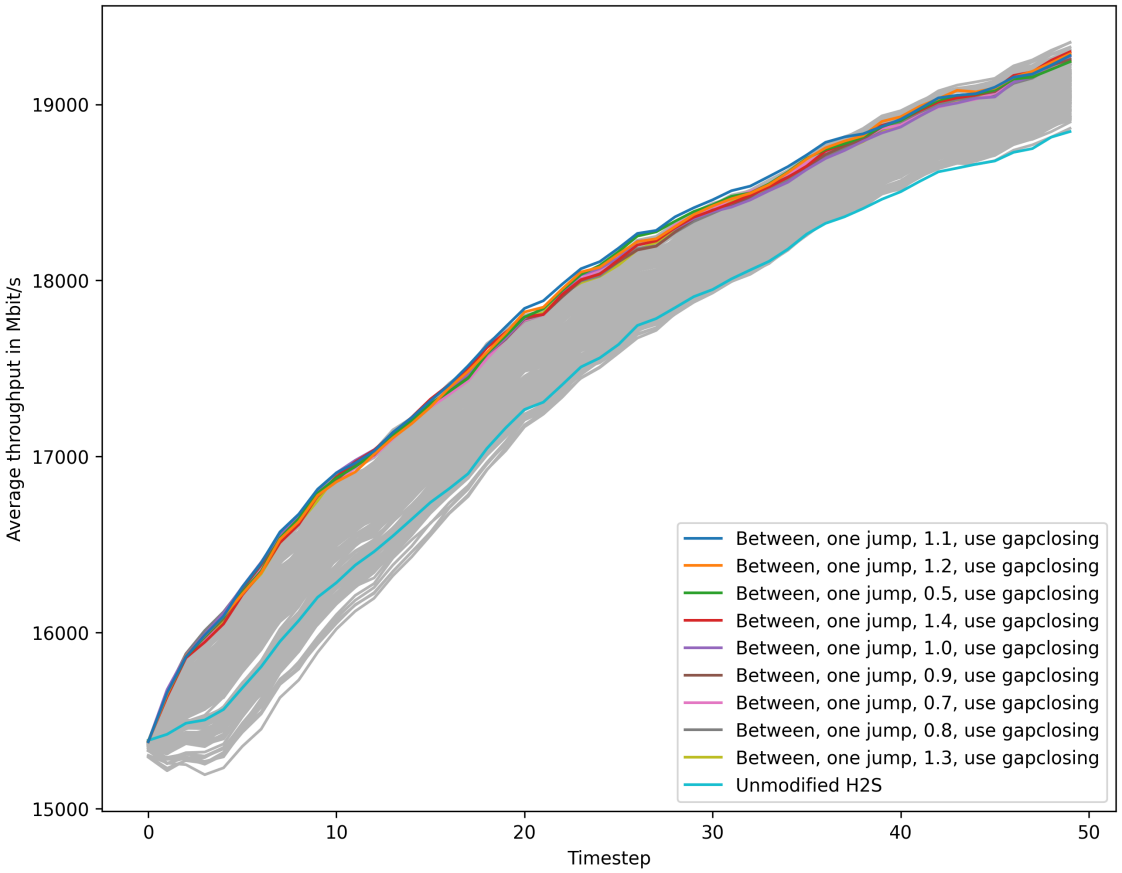

Figure 7.1: Result of the parameter experiments. Out of 217 parameter sets, only the top 9 that had the highest throughput across all timesteps are colored and ranked in the legend from best to worst. The unmodified H2S algorithm is colored in cyan, even though it was not among the top 9.

in a single timestep, giving the algorithm perfect knowledge of the scenario. While this is not a use case we focus on, the parameter set favored here is substantially different, which makes the data interesting nevertheless.

Same as before, we tested all 216 parameter combinations plus an unmodified version of H2S; the results are shown in Figure 7.3. Compared to unmodified H2S, the modified variants achieved a substantially lower improvement factor than in the multi-timestep scenarios. Also, whereas the multi-timestep scenarios favor the *between* and *use gapclosing* parameter values, the 1-timestep scenarios favor *everywhere* and *no gapclosing*. Overall the results are not surprising, given our modifications were specifically intended for future-proof scheduling, but we still achieved a small improvement. “Everywhere, border, 1.0, no gapclosing” would be the optimal parameter set for this specific use case.

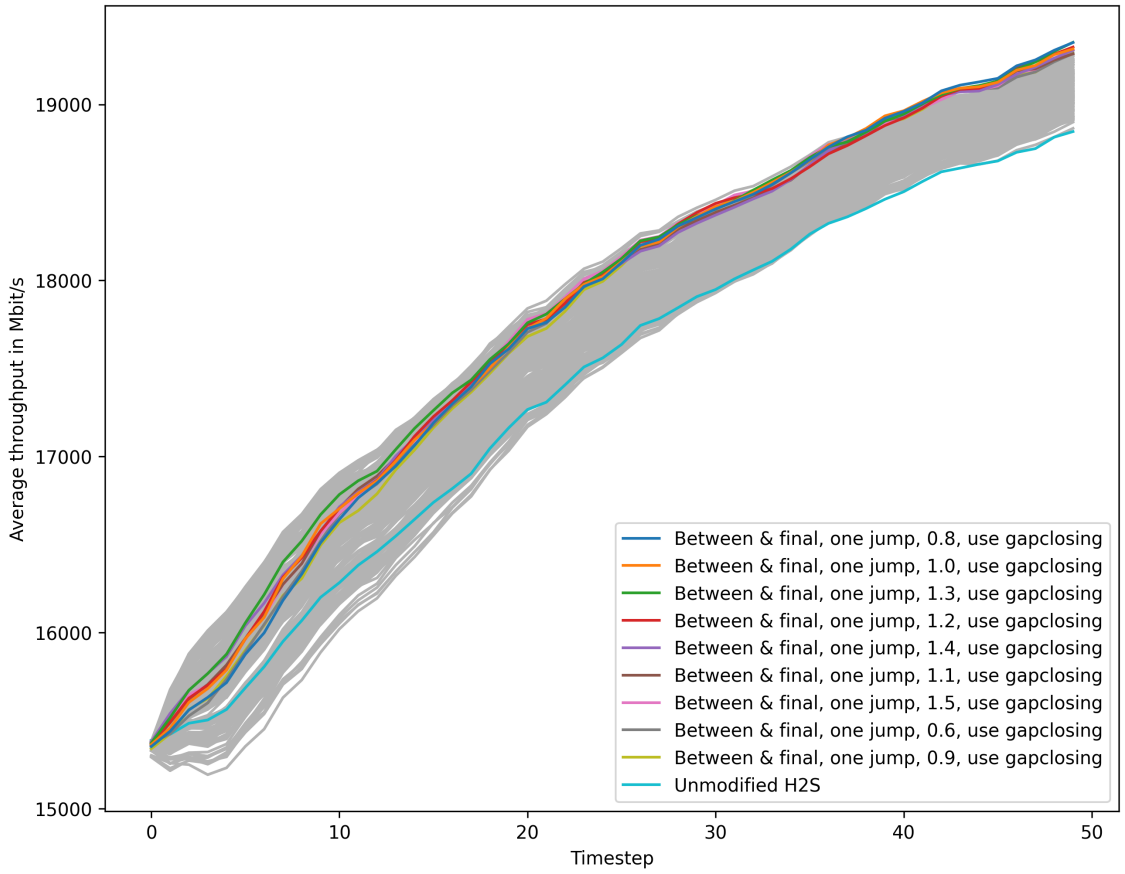

Figure 7.2: Result of the same parameter experiments illustrated in Figure 7.1, but this time the alternate selection that achieved the highest throughput *across all timesteps 37 and higher* are colored and ranked in the legend from best to worst. Again, unmodified H2S is included even though it was not part of the alternate selection.

		Brift mult.	0.5	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4	1.5	2.0
Gapclosing	Brift shift	Brift plcm.												
Use gapclosing	Border	Everywhere	19,928.45	19,902.50	19,932.70	19,880.35	19,848.50	19,938.95	19,970.50	19,970.65	20,007.85	19,991.30	19,992.00	20,032.95
		Between	19,933.35	19,933.35	19,933.35	19,933.35	19,933.35	19,933.90	19,933.90	19,933.15	19,929.75	19,931.60	19,933.20	19,933.45
		Betw. & fin.	19,854.20	19,845.30	19,784.30	19,760.60	19,747.00	19,834.45	19,881.15	19,873.50	19,963.10	19,905.40	19,939.80	19,945.05
	No jump	Everywhere	19,965.20	19,917.35	19,933.75	19,907.65	19,846.05	19,939.10	19,995.05	19,989.55	20,022.85	20,007.70	20,016.80	20,082.05
		Between	19,950.25	19,950.25	19,950.25	19,950.30	19,948.25	19,950.35	19,954.70	19,939.80	19,929.35	19,939.25	19,943.45	19,961.25
		Betw. & fin.	19,907.25	19,862.90	19,835.35	19,815.90	19,783.95	19,866.20	19,881.75	19,919.35	19,975.95	19,943.50	19,958.45	19,992.15
	One jump	Everywhere	19,958.40	19,910.45	19,929.65	19,908.20	19,835.55	19,936.00	19,978.85	19,995.60	20,036.15	20,003.10	20,018.85	20,063.80
		Between	19,950.10	19,949.60	19,949.50	19,952.20	19,950.75	19,950.40	19,951.90	19,946.05	19,939.40	19,934.30	19,942.60	19,949.95
		Betw. & fin.	19,897.10	19,861.35	19,843.65	19,821.00	19,774.00	19,865.35	19,883.00	19,921.90	19,968.60	19,933.55	19,952.70	19,983.10
No gapclosing	Border	Everywhere	20,140.05	20,118.50	20,118.55	20,135.80	20,139.60	20,164.55	20,135.15	20,132.15	20,129.05	20,114.75	20,111.80	20,113.20
		Between	20,026.60	20,026.60	20,026.60	20,026.60	20,026.60	20,026.60	20,026.60	20,026.70	20,025.60	20,025.60	20,025.60	20,031.55
		Betw. & fin.	19,975.30	20,013.30	20,006.45	20,050.55	20,056.15	20,065.75	20,059.30	20,051.50	20,052.15	20,071.50	20,060.45	20,025.55
	No jump	Everywhere	20,106.75	20,136.30	20,115.30	20,137.30	20,128.45	20,142.20	20,122.90	20,114.25	20,152.75	20,113.50	20,131.00	20,148.00
		Between	20,037.10	20,037.10	20,037.10	20,037.10	20,037.00	20,036.65	20,035.90	20,040.45	20,043.85	20,045.20	20,047.25	20,031.35
		Betw. & fin.	19,967.55	20,037.15	20,023.25	20,058.00	20,084.05	20,068.85	20,066.75	20,089.20	20,103.05	20,064.25	20,083.45	20,085.40
	One jump	Everywhere	20,108.30	20,128.15	20,120.50	20,140.60	20,129.40	20,145.25	20,121.65	20,115.30	20,152.10	20,127.65	20,142.75	20,141.80
		Between	20,041.40	20,041.20	20,040.10	20,040.50	20,042.15	20,039.65	20,040.25	20,039.60	20,042.10	20,042.40	20,042.60	20,015.10
		Betw. & fin.	19,976.90	20,038.55	20,030.85	20,059.90	20,084.60	20,074.55	20,059.05	20,078.40	20,101.80	20,073.15	20,080.50	20,080.00
		Unmodified H2S	20,032.40											

Figure 7.3: Average aggregated throughput in Mbit s^{-1} of single-timestep scenarios with all 217 parameter combinations. There is a color gradient applied to the table, with red being the worst throughput reading in the whole data set and dark green being the best.

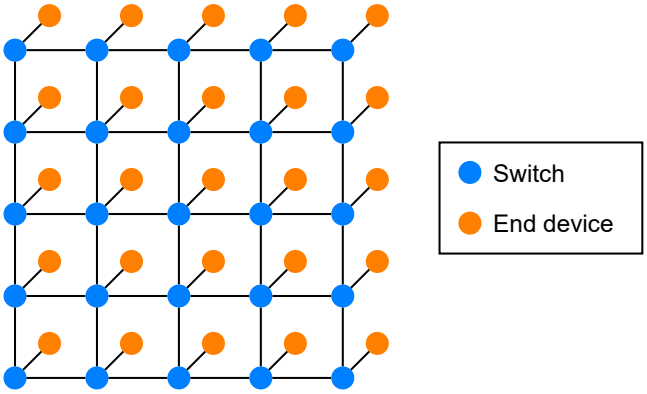

Figure 7.4: 5×5 grid topology

7.3 Prototype Evaluation

The four algorithm variants that we are going to evaluate are the unmodified versions of the H2S and CELF algorithms as introduced in Chapter 5 as well as H2S and CELF with our modified placement strategy and parameter set "between, one jump, 1.1, use gapclosing" as described in Section 7.2. The four algorithms will be tested on network topologies of different kinds and sizes. We used the Python NetworkX library [HSS08] to generate the topologies we use in our evaluations.

The evaluation features delta (Δ) metrics, which measure the difference on a regular metric that a modified algorithm achieves compared to its unmodified counterpart.

There is a solving time spike at timestep 0 for all algorithms on topologies with 100 switches or less. It is caused by the disproportionately high amount of flows added at timestep 0, compared to the way lower number of flows added in each subsequent timestep.

7.3.1 Various Topologies

Grid

Grids (also known as meshes) are our default topology; every evaluation in this thesis uses grids as their topology unless mentioned otherwise. In a grid topology, the switches can be arranged in such a way that their edges form a grid structure, as illustrated in Figure 7.4. Furthermore, every switch has exactly one end device connected to it, so there are always as many end devices as there are switches. We are using 5×5 as the default size, resulting in a network with 25 switches, 25 end devices, and a total of 65 two-way links between them.

Grid topologies are always planar and removing any individual link between switches will not disconnect the network. There are a lot of possible paths between any two end devices compared to our other topologies. If inner links are highly utilized, traffic can be routed through the outer links, mitigating utilization imbalances and hence major bottlenecks.

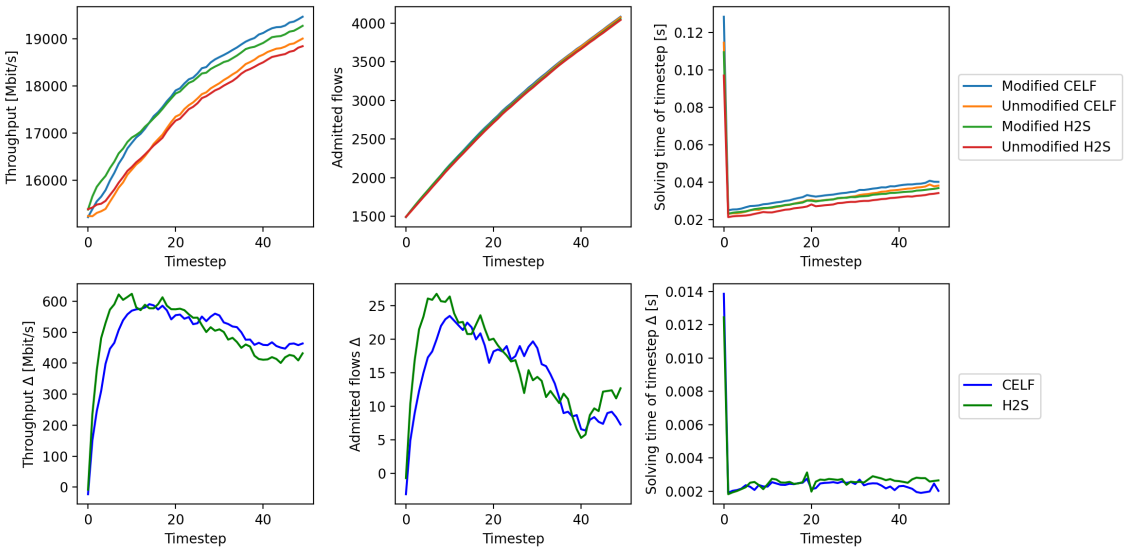

Figure 7.5: Evaluation results on a 5×5 grid topology

Figure 7.5 shows the performance of our four algorithms on a 5×5 grid topology. Unmodified H2S and CELF show similar performances, with H2S achieving more throughput in the initial timesteps and CELF achieving more throughput in the later timesteps. CELF consistently requires a few milliseconds more solving time per timestep than H2S, though the solving time of all variants consistently increases by 15 ms over time, due to the increased difficulties of placing flows in a highly utilized network.

The modified algorithms perform worse than the unmodified algorithms on timestep 0: The solving time is about 13.5 ms higher but the throughput and admitted flows delta remains at around 0. On subsequent timesteps the modified variants achieve a consistent performance increase of around 550 Mbit s^{-1} in throughput and around 22 in flows admitted while increasing solving time by a mere 2.5 ms per timestep. Both of our modified algorithms perform very similarly on both CELF and H2S, in throughput, admitted flows and solving time. The throughput delta diminishes slightly and the admitted flows delta diminishes substantially as the network becomes more utilized. This is due to the consistent influx of random flows which, over many timesteps, are able to utilize free space in the schedule that our modified algorithms were able to utilize earlier already.

Random

Random topologies are randomly generated using this procedure: First, the desired number of switches n is added to the network and they are then linked together like a $G_{n,p}$ random graph [Gil59]: Every pair of switches has probability p of being connected, independent from all other connections. If this generates a graph that is not connected, the result is discarded and the process repeated until a connected graph is generated. Finally, every switch is connected to exactly one end device, making the number of switches and end devices equal. By default, our number of switches is 25 and the pair connection probability $p = 0.128$.

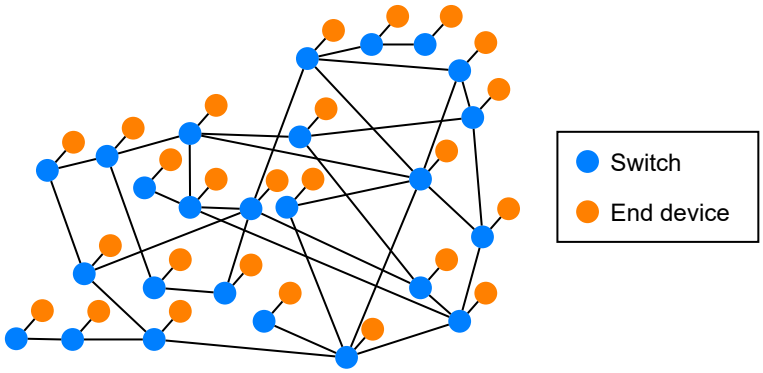

Figure 7.6: Example random topology with 25 switches and $p = 0.128$

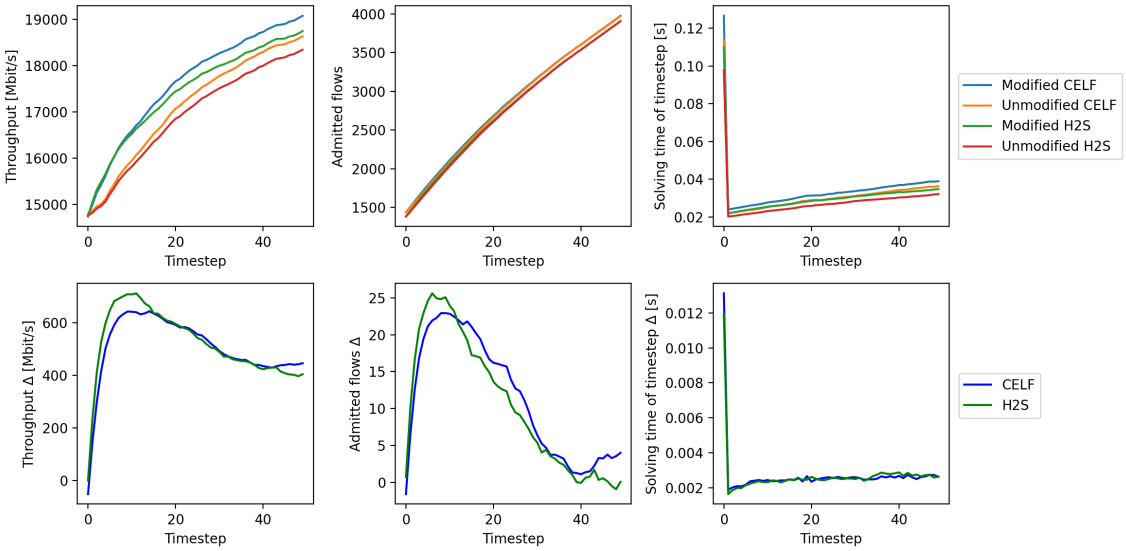

Figure 7.7: Evaluation results on random topologies

Random topologies are varied, featuring switches with both high and low numbers of connections. Since all switches have the same probability of being connected, the network's diameter usually scales less with increasing network size compared to grids.

While CELF and H2S had about the same performance on the grid topology, CELF performs better on the random topologies, both in throughput and admitted flows. The throughput, admitted flows and solving time deltas are very similar to those of the grid topology, albeit the graphs look smoother since they represent the averaged performance over 5 random topologies instead of a single grid topology. Also, the admitted flows deltas dip a lot deeper in the later timesteps than they did on the grid topology.

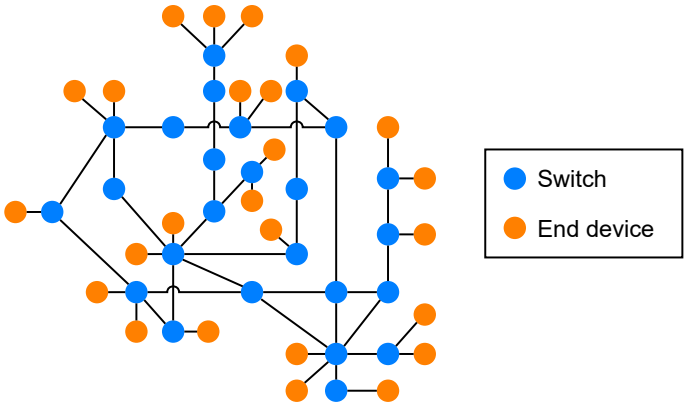

Figure 7.8: Example waxman topology with 25 switches and 25 end devices

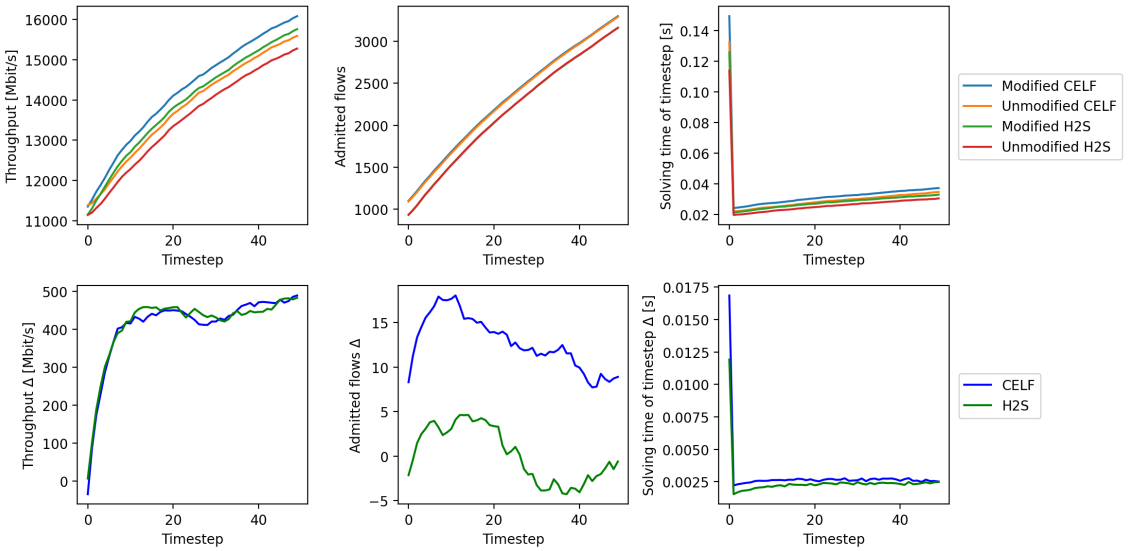

Figure 7.9: Evaluation results on waxman topologies

Waxman

Waxman topologies [DC02] are randomly generated using this procedure: First, all switches are randomly placed in an 1×1 rectangle. Then the switches are randomly connected, whereas switches close to each other are more likely to be connected. Concretely, each pair of switches has a probability of $p = 0.4 \times \exp \frac{-d}{0.25 \times L}$ of being connected where d is the distance between the two nodes and L is the distance between the two nodes in the network that are furthest apart. Should this procedure result in a disconnected graph, then the result is discarded and the procedure repeated until the result is a connected graph. Finally, the end devices are placed: Switches with a degree of 1 are connected to an end device. Then, end devices are connected to switches at random until the desired number of end devices is reached. Figure 7.8 shows an example waxman topology.

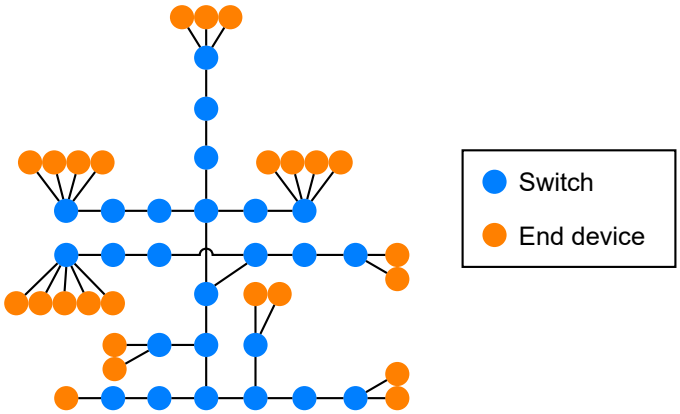

Figure 7.10: Example tree topology with 25 switches and 25 end devices

For our evaluations we use waxman topologies with 25 switches and 25 end devices. Waxman topologies attempt to mimic real world networks in the sense that switches in close physical proximity are more likely to be connected by a cable. Due to them being randomly generated, they often feature cycles of various sizes as well as small segments of the network that are only connected to the rest through one or two links. Since they have less links and paths are generally not shorter than in the grid topology, the overall achievable throughput and flows admitted is lower.

Figure 7.9 shows the performance of our four algorithms averaged over 5 different waxman topologies. The results are similar to those of the grid topology: No notable performance increase on timestep 0, but about 400 Mbit s^{-1} more throughput in subsequent timesteps. Notable is that the CELF variants perform substantially better on waxman topologies than the H2S variants, both in throughput and admitted flows. This is due to CELF being able to mitigate bottlenecks, which are more common in a waxman topology than in a grid topology.

Unlike with the grid topology, the throughput delta does not diminish with increasing network utilization, in fact it increases by about 80 Mbit s^{-1} over time. The admitted flow delta is what differentiates the waxman performance most from the grid performance: Instead of the delta starting at 0 before skyrocketing and then diminishing, the H2S delta takes the rough shape of a sine wave, being positive in the first half of timesteps and negative in the second half before approaching 0 again at the final timestep. The CELF admitted flow delta takes the same shape as the one of H2S, at least in the first half of timesteps, but modified CELF can admit about 12 flows more on average and is never worse than 0.

Tree

Tree topologies are randomly generated using this procedure: The switches are generated using the `random_tree` function of the Python NetworkX library [HSS08]. Then, all switches with a degree of 1 are connected to one or more end devices, depending on the desired total number of end devices. Figure 7.10 shows an example tree topology.

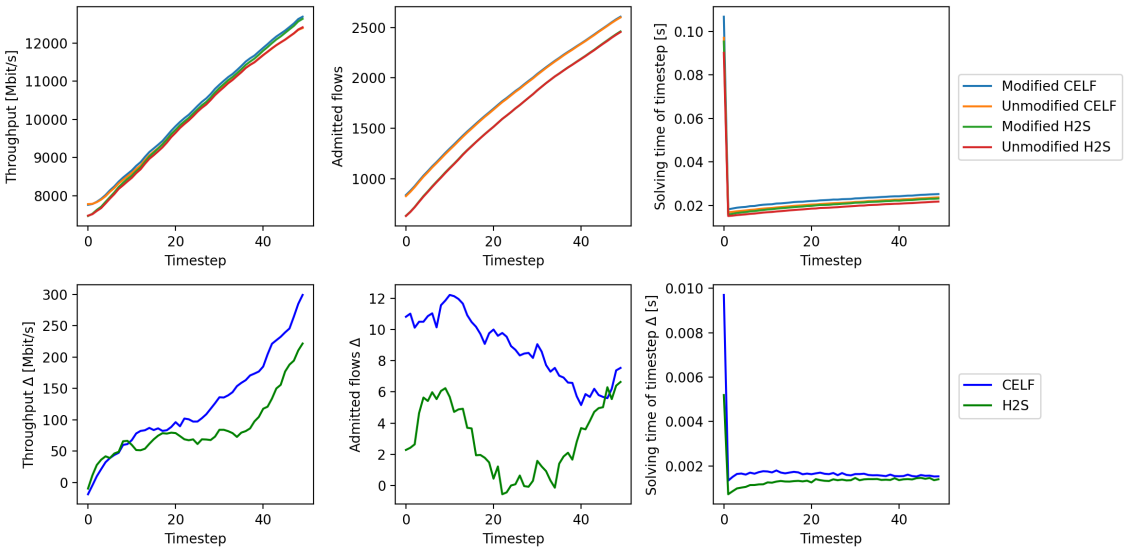

Figure 7.11: Evaluation results on tree topologies

The special properties of trees is that there is only one path between any two end devices. It is very likely that a large bottleneck forms in the center of the tree, reducing the maximum achievable throughput.

Whereas our modified algorithms were able to achieve a high throughput delta of hundrets of $\mathrm{Mbit\,s^{-1}}$ in the first few timesteps on the other topologies, they only achieve about $70\,\mathrm{Mbit\,s^{-1}}$ on the tree topology initially. Past 20 timesteps the throughput delta starts to increase substantially, though even then it is still lower compared to the other topologies. Therefore it seems that, with a tree topology, the future-proof aspects of our modifications, especially gap closing, are more relevant than the more immediate benefits that backward rifts introduce.

Whereas in the first few timesteps the CELF algorithms achieve the highest throughput, in the last few timesteps the modified algorithms are favored instead. The CELF algorithms will favor flows that do not need to pass through the highly utilized center of the tree which initially boosts throughput. In later timesteps however, other random flows that have their paths in the outer parts of the tree would fill these underutilized links anyway, nullifying the benefits of using CELF instead of H2S.

7.3.2 Scaling

So far, our algorithm variants were evaluated on small networks with only 25 switches. Now we will evaluate how they will perform on bigger networks, on 5×5, 7×7, 10×10, and 32×32 grid topologies with 25, 49, 100 and 1,024 switches respectively.

Larger networks naturally allow for a higher aggregated throughput and a higher rate of admitted flows. We need to make sure that our algorithms never manage to easily schedule all flows in order to be able to evaluate the performance among them. Hence we have adjusted the scenarios for larger

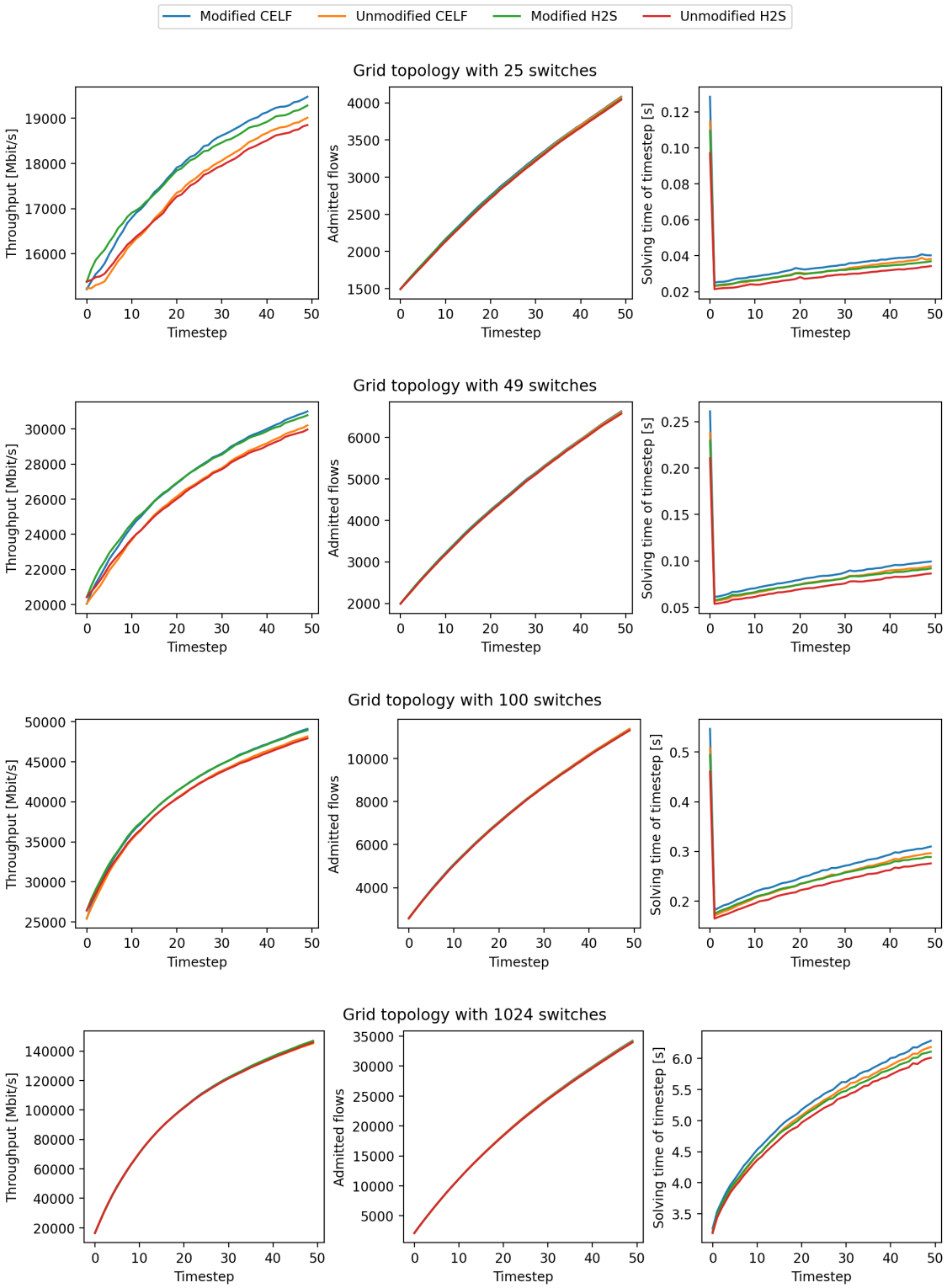

Figure 7.12: Evaluation results on grid topologies of various sizes (regular metrics)

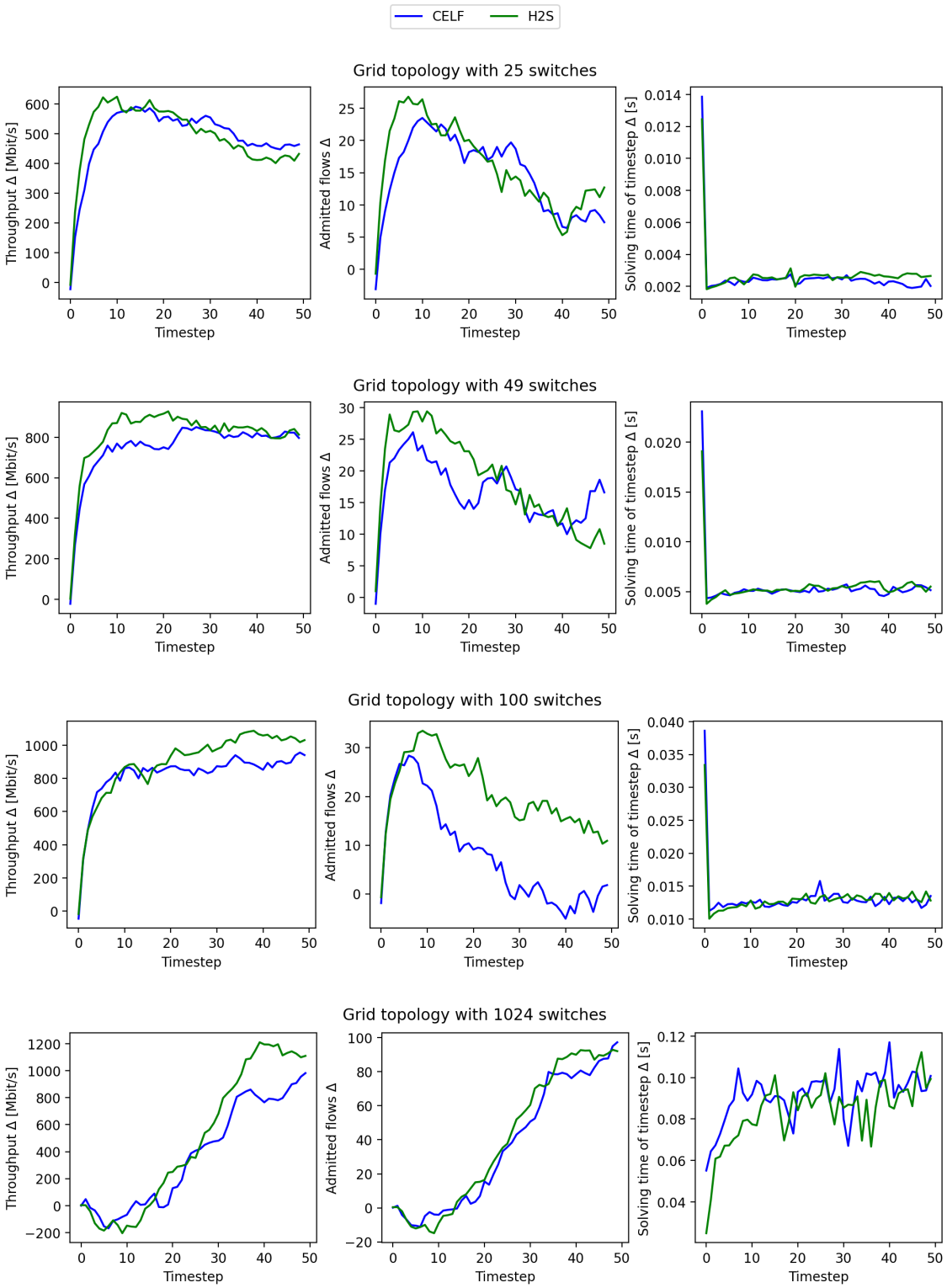

Figure 7.13: Evaluation results on grid topologies of various sizes (delta metrics)

Network size in number of switches	Flows added on timestep 0	Flows added on timesteps > 0	Flows removed on timesteps > 0
25	1500	400	200
49	2000	350	175
100	2600	700	350
1024	2400	2400	1200

Table 7.1: The scenario parameters used for different network sizes.

networks, as specified in Table 7.1. Counter intuitively, grid networks with 100 switches are able to schedule slightly more flows on timestep 0 than networks with 1,024 switches, since the average path lengths between end devices are substantially longer on a 32×32 grid.

Figure 7.12 and Figure 7.13 list the resulting performance diagrams for all aforementioned network sizes. Our modified variants achieve no notable performance increase in terms of admitted flows across all sizes. In terms of throughput, they consistently achieve a higher throughput than their unmodified counterparts, scaling from $500\,\mathrm{Mbit\,s^{-1}}$ to $900\,\mathrm{Mbit\,s^{-1}}$ at a size of 100 switches. However, the solving time increase scales from 2.5 ms to around 13 ms in the same size range, making the drawbacks scale faster than the benefits.

In networks with 1024 switches, the performances behave abnormally compared to the three smaller networks, especially the performance deltas among the modified and unmodified algorithm variants, listed on Figure 7.13. Most notably, in 1024 switch networks, instead of the solving time delta spiking on timestep 0 due to the large influx of flows compared to subsequent timesteps, solving times are actually lower on timestep 0 compared to subsequent timesteps. The main reasons of this are the base solving time per timestep no longer being below one second and the scenario adding proportionally less flows on timestep 0 compared to subsequent timesteps. Furthermore, while the throughput delta increases greatly after timestep 0 in smaller networks and then quickly reaches a plateau, 1024 switch networks instead have negative throughput deltas in the first 15 timesteps, reaching the aforementioned plateau much more slowly.

Overall, the benefits of our proposed modifications diminish as the networks grow larger. However, they still achieve throughput improvements in the long term regardless of network size. While the throughput improvements do not scale well compared to the solving time, the solving time increase remains at reasonable levels, making our improvements still viable even in large networks where high throughput is desired.

7.3.3 Random topologies with 1024 switches

Since the delta metrics of large 1024 switch networks behave so differently than the other sizes we looked at, we decided to also present large *random* topologies with 1024 switches, instead of just large grid topologies. To keep the amount of links somewhat comparable to those in the grid topology, we use $p = 0.00477$ as the link connection probability, which should result in an expected value of 2500 links between switches in total while the grid topology has 1984 links between switches.

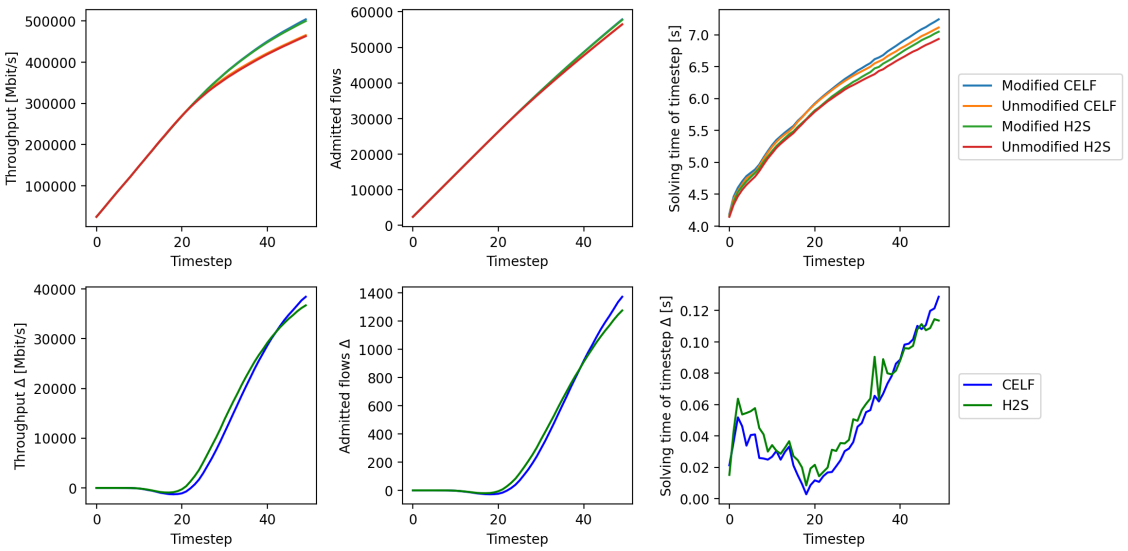

Figure 7.14: Evaluation results on large random topologies with 1024 switches

As show in Figure 7.14, there is not much of a performance difference between the H2S and CELF algorithms, but a notable difference between the modified and unmodified variants. While the difference in throughput among the algorithms on a grid with 1024 switches was so small that all four graphs merged into a a single overlapping line, here the modified algorithms suddenly achieve a substantially higher throughput than the unmodified algorithms past timestep 20. Over all previous evaluations, the highest throughput delta achieved was 1,200 Mbit s^{-1} and now we are seeing a steady increase to up to 40,000 Mbit s^{-1}. The same is true for the admitted flows delta, capping at about 100 on the large grid with 1024 switches but now achieving a steady increase up to 1,400 Mbit s^{-1}. Finally, the solving time deltas are unusual too, spiking after timestep 0, then gradually decreasing up until timestep 20 and then steadily increasing to 120 ms.

In summary, on random topologies with 2014 switches, our modified algorithms achieve next to no increased performance before timestep 20 and then a performance increase of up to 5% after timestep 20.

7.3.4 Offensive planning

All previous evaluations solely used defensive scheduling, meaning that slots that have been placed in the schedule cannot be modified unless their flow is removed during a timestep. Offensive scheduling allows for reorganizing already placed flows as part of an offensive step. This has been described in more detail in Chapter 5.

The most unusual results are the solving times: The solving time of both CELF algorithms increases with each timestep, but in three instances the solving time is abruptly lowered before consistently increasing again. These saw teeth are solely caused by offensive steps, which are almost always attempted since the defensive solution is rarely able to incorporate all flows. We know that the saw teeth are not caused by either an offensive or defensive solution being chosen at those respective timesteps due to the likelihood of an offensive solution becoming lower which each time step, to

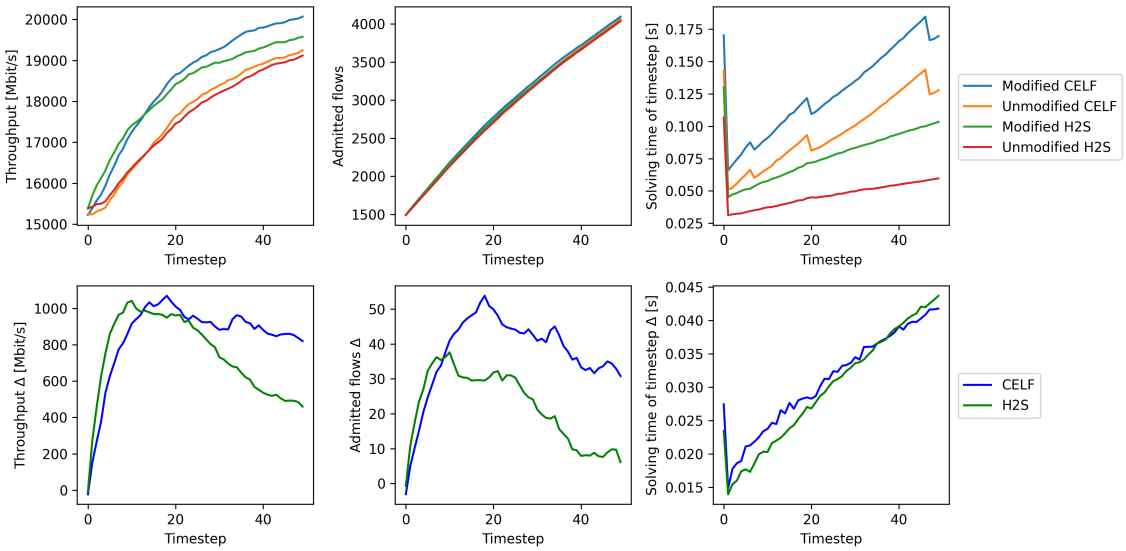

Figure 7.15: Performances when using offensive scheduling on a grid network with 25 switches

the point that only defensive solutions are chosen past timestep 40. This is due to the fact that it becomes increasingly harder to schedule all preexisting flows during an offensive step when the network is already highly utilized. Despite this, offensive steps are still attempted across all timesteps, even though they are very likely to repeatedly fail.

Our current best theory on how the saw teeth come to be is that key links in the network become more and more utilized over time until passing a threshold where the CELF algorithms suddenly consider flow configurations through other links more viable, where finding slot placements for them is a lot easier. This would explain why the H2S algorithms do not show the same behavior, since they do not consider link utilization at all. Either way, the saw teeth phenomenon is caused by unmodified CELF and our modified version merely inherits the same property that causes this phenomenon.

The solving times increase steadily increase as network utilization increases since the offensive steps attempt to reschedule all currently scheduled flows, which become larger in number with each timestep. For each one of those flows the code segments with our improvements are invoked as well, also affecting the solving time deltas.

The throughput and admitted flows deltas our modified algorithms achieve are quite high for a small network. These evaluations were made on a network with 25 switches and the deltas surpass those of a network with 49 switches when using defensive strategies only (cf. Figure 7.13). However, this only applies to the CELF variant; the modified H2S algorithm achieves substantially less throughput and admitted flow deltas than the CELF algorithm.

Overall, our modified version of CELF is a substantial improvement over unmodified CELF when using offensive planning. However, there is also a substantially increased solving time, hence we expect modified CELF to scale poorly when the topology is a lot larger.

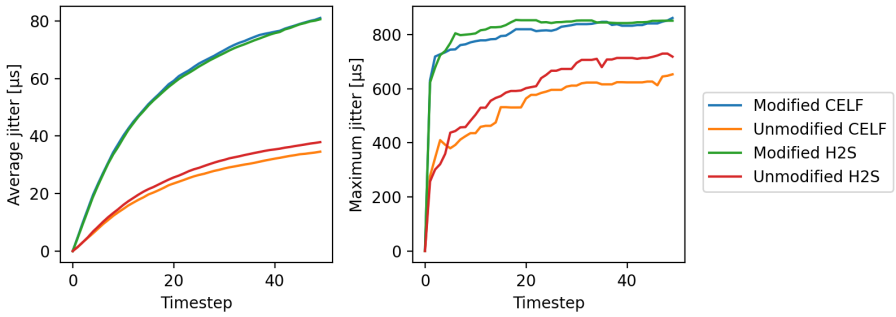

Figure 7.16: Reception jitter on a grid topology with 25 switches

7.3.5 Jitter

Finally, we examine the reception jitter that our modifications cause. As mentioned in Section 6.2, we consider higher throughput in exchange for higher reception jitter a favorable tradeoff. Figure 7.16 illustrates the reception jitter our four algorithms cause. Our modified variants quickly climb to a maximum jitter of 800 µs whereas the unmodified variants take a bit longer to settle at about 600 µs maximum jitter. All it takes for maximum jitter to become high is a single unfavorable flow, which is inevitable when there are hundreds of flows scheduled. The unmodified algorithms limit the combination of subcycles that a flow can use in order to reduce jitter. The modified algorithms do not do this, hence maximum jitter is higher.

The average jitter behaves similarly, with the exception that it slowly increases with more and more timesteps. On timestep 0, both maximum and average jitter are zero because flows are scheduled in asceding order by their period, leads to their slots being placed in an ordered fashion. But as soon as a lower period flow is placed after a high period flow, jitter can accumulate, which can easily happen on timestep 1.

H2S creates slightly more jitter than its CELF counterpart since CELF favors configurations with less utilization, where the schedule is more likely to still be ordered.

8 Conclusion and Outlook

Time-triggered networks use a schedule do ensure that real time messages reach their destination on time. Online scheduling algorithms can not only generate TTN schedules quickly, but also update existing schedules, making their network adaptable and extendable without the need for interrupting the network while it is running. In this thesis, we discussed ways of improving two online scheduling algorithms for time triggered networks: H2S and CELF. We proposed three modifications to these algorithms: "Relaxed jitter" removes the jitter limitation strategies used by H2S and CELF, reducing the number of flows being rejected. "Backward rifts" fill those empty spaces in the schedule that are hard to utilize effectively by using slots that have the least placement restrictions. And finally, "gap closing" performs preventive, defensive defragmentation to reduce the number of empty spaces that can only be filled by slots that are small enough. All these modifications were combined into an improved placement algorithm and tested with different parameters. We determined that "between, one jump, 1.1, use gapclosing" is the best parameter combination for scenarios where more and more flows are introduced to the network in several timesteps.

We then evaluated how well our modifications perform in different scenarios. Our modifications are generally better at improving aggregated network throughput rather that flows admitted. The algorithms achieve throughput improvements with grid, random, waxman, and tree topologies, though the grid topology was the most ideal with a consistent performance increase of about $550\,\mathrm{Mbit\,s^{-1}}$ in throughput while only increasing solving time by 2.5 ms in a network with 25 switches. Different network sizes were tested as well, with the result that our algorithm generally works best in small networks with around 25 switches. Our modifications achieved throughput improvements in larger networks as well, though our they scale a lot better on random topologies, rather than grid topologies. At the same time, the additional solving time our modifications require are still small, about 100 ms in large networks with 1,024 switches. Finally, our modifications work really well with offensive scheduling strategies, achieving up to $1{,}000\,\mathrm{Mbit\,s^{-1}}$ more throughput in networks with 25 switches. However, the solving time with offensive strategies does not scale well, which could become problematic in networks much larger.

All this comes at a cost of higher reception jitter, though our algorithm could be adapted to limit jitter at the cost of some throughput.

Outlook

If low jitter is of importance, our "relaxed jitter" modification could be reverted to once again limit the combination of subcycles that a flow can place slots in. With lower jitter also comes a lower buffering requirement, which is relevant when adapting H2S and CELF to TTN, another possible topic for the future.

We also played with the idea of splitting large frames up, so that the smaller frame fragments can fit into free space in the schedule that the large frame could not. This could optimize throughput and admitted flows even further.

Finally, our implementations of the algorithms were single-threaded so far. It could be viable to explore multithreading for reduced solving times.

Bibliography

[18] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600. DOI: 10.1109/IEEESTD.2018.8457469 (cit. on pp. 19, 21).

[Ade22] A. Ademaj. *Time-Triggered Ethernet – A Powerful Network Solution for Multiple Purpose*. 2022. URL: https://www.tttech.com/sites/default/files/documents/TTEthernet_Technical-Whitepaper_TTTech.pdf (visited on 05/21/2024) (cit. on pp. 20, 22, 23).

[BAP+22] D. Bujosa, M. Ashjaei, A. V. Papadopoulos, T. Nolte, J. Proenza. "HERMES: Heuristic Multi-queue Scheduler for TSN Time-Triggered Traffic with Zero Reception Jitter Capabilities". In: *RTNS 2022: The 30th International Conference on Real-Time Networks and Systems, Paris, France, June 7 - 8, 2022*. Ed. by Y. Abdeddaim, L. Cucu-Grosjean, G. Nelissen, L. Pautet. ACM, 2022, pp. 70–80. DOI: 10.1145/3534879.3534906 (cit. on p. 28).

[COCS16] S. S. Craciunas, R. S. Oliver, M. Chmelík, W. Steiner. "Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Brest, France: Association for Computing Machinery, Oct. 19, 2016, pp. 183–192. ISBN: 9781450347877. DOI: 10.1145/2997465.2997470. URL: https://doi.org/10.1145/2997465.2997470 (cit. on pp. 24, 47).

[DC02] J. Dall, M. Christensen. "Random geometric graphs". In: *Physical review E* 66.1 (2002), p. 016121 (cit. on p. 54).

[DK+14] F. Dürr, T. Kohler, et al. "Comparing the forwarding latency of openflow hardware and software switches". In: *Fakultät Informatik, Elektrotechnik Informationstechnik, Univ. Stuttgart, Stuttgart, Germany, Tech. Rep. TR* 4 (2014), p. 2014. URL: https://www2.informatik.uni-stuttgart.de/bibliothek/ftp/ncstrl.ustuttgart_fi/TR-2014-04/TR-2014-04.pdf (cit. on p. 47).

[DN16] F. Dürr, N. G. Nayak. "No-wait Packet Scheduling for IEEE Time-sensitive Networks (TSN)". In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Brest, France: Association for Computing Machinery, Oct. 19, 2016, pp. 203–212. ISBN: 9781450347877. DOI: 10.1145/2997465.2997494 (cit. on p. 27).

[Edd22] W. Eddy. *Rfc 9293: Transmission control protocol (tcp)*. 2022 (cit. on p. 15).

[Fan] G. Fanuc. *TTEthernet–a powerful network solution for advanced integrated systems* (cit. on p. 20).

[FGD+22] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. “Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows in the Real-Time Data Plane”. In: *IEEE Trans. Netw. Serv. Manag.* 19.2 (2022), pp. 1807–1825. DOI: 10.1109/TNSM.2022.3150664 (cit. on p. 47).

[GDBR23] H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. “Just a Second - Scheduling Thousands of Time-Triggered Streams in Large-Scale Networks”. In: *CoRR* abs/2306.07710 (2023). DOI: 10.48550/ARXIV.2306.07710. arXiv: 2306.07710 (cit. on pp. 27, 31–33, 35).

[Gil59] E. N. Gilbert. “Random graphs”. In: *The Annals of Mathematical Statistics* 30.4 (1959), pp. 1141–1144 (cit. on p. 52).

[HSS08] A. A. Hagberg, D. A. Schult, P. J. Swart. “Exploring Network Structure, Dynamics, and Function using NetworkX”. In: *Proceedings of the 7th Python in Science Conference*. SciPy. SciPy, June 2008, pp. 11–15. DOI: 10.25080/tcwv9851 (cit. on pp. 51, 55).

[KAGS05] H. Kopetz, A. Ademaj, P. Grillinger, K. Steinhammer. “The Time-Triggered Ethernet (TTE) Design”. In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2005), 18-20 May 2005, Seattle, WA, USA*. IEEE Computer Society, 2005, pp. 22–33. DOI: 10.1109/ISORC.2005.56 (cit. on pp. 15, 20, 22, 23).

[KDD14] S. Kumar, S. Dalal, V. Dixit. “The osi model: overview on the seven layers of computer networks”. In: *International Journal of Computer Science and Information Technology Research* 2.3 (2014), pp. 461–466 (cit. on p. 17).

[KLR94] M. H. Klein, J. P. Lehoczky, R. Rajkumar. “Rate-monotonic analysis for real-time industrial computing”. In: *Computer* 27.1 (1994), pp. 24–33 (cit. on p. 15).

[MNX+18] M. Mohaqeqi, M. Nasri, Y. Xu, A. Cervin, K. Årzén. “Optimal harmonic period assignment: complexity results and approximation algorithms”. In: *Real Time Syst.* 54.4 (2018), pp. 830–860. DOI: 10.1007/S11241-018-9304-0 (cit. on pp. 29, 47).

[PRCS16] P. Pop, M. L. Raagaard, S. S. Craciunas, W. Steiner. “Design optimisation of cyber-physical distributed systems using IEEE time-sensitive networks”. In: *IET Cyper-Phys. Syst.: Theory & Appl.* 1.1 (2016), pp. 86–94. DOI: 10.1049/IET-CPS.2016.0021 (cit. on p. 27).

[RAK10] S. Rahamatkar, A. Agarwal, N. Kumar. “Analysis and comparative study of clock synchronization schemes in wireless sensor networks”. In: *Analysys* 2.3 (2010), pp. 536–541 (cit. on p. 15).

[RPGS17] M. L. Raagaard, P. Pop, M. Gutiérrez, W. Steiner. “Runtime reconfiguration of time-sensitive networking (TSN) schedules for Fog Computing”. In: Santa Clara, CA, USA. Santa Clara, CA, USA: IEEE, 2017, pp. 1–6. ISBN: 978-1-5386-3667-1. DOI: 10.1109/FWC.2017.8368523 (cit. on p. 28).

[SDT+17] E. Schweissguth, P. Danielis, D. Timmermann, H. Parzyjegla, G. Mühl. “ILP-based joint routing and scheduling for time-triggered networks”. In: *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. RTNS ’17. ACM, Oct. 2017. DOI: 10.1145/3139258.3139289 (cit. on p. 27).

[SSN19] A. C. T. dos Santos, B. Schneider, V. Nigam. “TSNSCHED: Automated Schedule Generation for Time Sensitive Networking”. In: San Jose, CA, USA (Oct. 2019). San Jose, CA, USA: IEEE, Oct. 2019, pp. 69–77. ISBN: 978-1-7281-4089-6. DOI: 10.23919/FMCAD.2019.8894249 (cit. on p. 27).

[TPS12] D. Tamas-Selicean, P. Pop, W. Steiner. “Synthesis of communication schedules for TTEthernet-based mixed-criticality systems”. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS ’12. Tampere, Finland: Association for Computing Machinery, Oct. 7, 2012, pp. 473–482. ISBN: 9781450314268. DOI: 10.1145/2380445.2380518. URL: https://doi.org/10.1145/2380445.2380518 (cit. on p. 22).

[VBHT22] M. Vlk, K. Brejchová, Z. Hanzálek, S. Tang. “Large-scale periodic scheduling in time-sensitive networks”. In: *Comput. Oper. Res.* 137 (2022), p. 105512. DOI: 10.1016/J.COR.2021.105512 (cit. on p. 28).

[VHT21] M. Vlk, Z. Hanzálek, S. Tang. “Constraint programming approaches to joint routing and scheduling in time-sensitive networks”. In: *Comput. Ind. Eng.* 157 (2021), p. 107317. DOI: 10.1016/J.CIE.2021.107317 (cit. on p. 27).

[WAA+15] S. T. Watt, S. Achanta, H. Abubakari, E. Sagen, Z. Korkmaz, H. Ahmed. “Understanding and applying precision time protocol”. In: Jeddah, Saudi Arabia. Jeddah, Saudi Arabia: IEEE, Dec. 2015, pp. 1–7. ISBN: 978-1-4673-9454-3. DOI: 10.1109/SASG.2015.7449285 (cit. on p. 15).

[Yen71] J. Y. Yen. “Finding the K Shortest Loopless Paths in a Network”. In: *Management Science* 17.11 (July 1971), pp. 712–716. ISSN: 1526-5501. DOI: 10.1287/mnsc.17.11.712 (cit. on p. 33).

All links were last followed on 01.10.2024

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