How fast can it wake up? Putting Link Sleeping into Practice

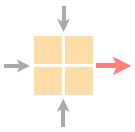
Master Thesis

Author(s):
Röllin, Lukas

Publication date:
2023-10

Permanent link:
https://doi.org/10.3929/ethz-b-000637984

ETH zürich



Networked Systems
ETH Zürich — seit 2015

How Fast Can It Wake Up?
Putting Link Sleeping into Practice

Master Thesis

Author: Lukas Röllin

Tutor: Romain Jacob

Supervisor: Prof. Dr. Laurent Vanbever

May 2023 to October 2023

Abstract

We investigate link sleeping as a way to save energy in computer networks and explore the topic from the perspective of finding the bottlenecks and problems that arise when implementing it on current systems. With this work, we revise the previous assumption that network interfaces can wake up within milliseconds and show that today's transceivers take seconds to wake up. We show that link sleeping is nevertheless possible today and explore how it behaves in different conditions. Our experiments identify the interface wake-up time as the biggest limitation for the performance of link sleeping. This work is a stepping stone for future research aiming to improve energy efficiency in networks through link sleeping.

Contents

Chapter 1

Introduction

Previous research on making networks more energy efficient falls into two categories: making the individual parts more efficient or turning parts that are not needed off. This work focuses on the second category, specifically turning off individual links.

In wired networks, turning off links is promising for two reasons. One is the observation that many network links are underutilized [7] (<30%) because network operators typically plan for worst-case scenarios. The other reason is that transceivers waste a lot of energy when idle. For example, we measure 5W idle power vs. 6.2W under full load for a 100G LR4 optical transceiver. In other words, 80% of its total power is spent for no useful work. Additional power measurements and the measurement methodology can be found here [11].

Previous research on link sleeping made at least one of those two assumptions:

- The Interfaces can wake up quickly (within milliseconds)
- The controller has access to up-to-date or future network state information

The first assumption is that interfaces can wake up within a few milliseconds; in fact, we observe that they take three orders of magnitude longer (Fig. 5.1). This makes many previous ideas, *e.g.*, waking up for each incoming packet or buffer and burst traffic [12], infeasible.

The second assumption is that the required network state information is instantaneously available to the controller. Some works even assume reliable predictions of future network demands, which is very challenging—at best. Some of the works that assume future network demands are [8] and [5]. However, collecting and distributing network information takes some time and might be inconsistent due to different delays from the nodes to the controller.

Our work aims to identify the limits of today's networks and design a realistic link sleeping system. In summary:

- We present a first prototype of a *power plane* responsible for optimizing the network energy usage, which is seamlessly integrated between the standard data and control planes (§ 3.1).
- We evaluate the performance of this prototype for different network and traffic scenarios (§ 5.2).
- We measure interface wake-up delays (§ 5.1) and show that milliseconds wake-up times remain impossible today.
- Nonetheless, we show it is possible to implement link sleeping in hardware if one considers larger time scales (§ 5.3).

This work helps estimate how often we can set links to sleep and how long it takes for the network to react to traffic changes. We show in what kind of network conditions link sleeping performs well and what the costs are if it does not. Sadly, we cannot translate sleeping times into energy savings yet as this requires power models that are currently unavailable. Establishing such models is part of parallel research efforts.

The first few weeks of this work were spent trying to get a better understanding of what the bottlenecks are for waking up networking hardware. While this work is focusing on turning off individual links, additional insights about line card sleeping can be found in Appx. A.

Chapter 2

Background and Related Work

2.1 Background

This section gives some relevant background on the steps necessary to wake up a link if we detect congestion in the network. We present some rough timescales for those steps as present in our experiment setup. In the second part, we introduce the OSPF TE metrics extension which we use to distribute network link load information in our system.

2.1.1 Link Wake-up Steps

When using link sleeping techniques, being able to turn on a network interface fast is important. Links should only be asleep if they are not essential to the operation of the network but should be available as soon as there is more demand than can be handled by the sleeping network. Being able to react to such changes requires a few steps that need to happen before the link is available to forward traffic again.

Steps	Scale	Limitation
Detection	∼1s	frequency of load measurement
Flooding	∼0.1s	network diameter
Wake-up	∼1-10s	interface wake up delay (Fig. 5.1)
Convergence	∼0.1s	OSPF parameters & computation

Table 2.1: Our measurements show that the link wake-up delay is the slowest part of the wake-up process.

Table 2.1 summarizes the major steps and shows which factors limit the speed of each step. The first step is to detect that there is more demand than can be handled. For example, this can be done by monitoring the link load and is mainly limited by how frequently this information can be measured.

The second step is to forward the congestion signal to all the nodes that must react. In the worst case, the information must be distributed to all nodes, which scales with the network diameter.

The next step is physically turning on the interface. Surprisingly we found this to be the slowest part in the order of a few seconds to more than 10 seconds for some of the tested transceivers. A more detailed analysis can be found in § 5.1.

Finally, the routing state must converge to use the newly woken-up interface; that is, detecting that the interface is up, computing the new best paths and installing those paths in the data plane. Without tuning, this can take tens of seconds but can be reduced by tuning the routing protocol parameters. With tuning, we can get OSPF to converge within a few 100 milliseconds. The exact parameters can be found in § 4.2.

2.1.2 OSPF TE Metrics Extension

OSPF is an intra-domain routing protocol that uses link information to calculate the shortest path within the network. To distribute the link state information inside the network the OSPF protocol uses link state advertisement (LSA) messages. There are a few types of LSA messages one being opaque LSAs which can be used to add functionality to OSPF which not all devices inside the network have to support. If a router does not have support for the specific feature, it will just ignore the content but still distribute the LSA to its neighbors. The feature we are interested in is the OSPF Traffic Engineering (TE) Metric Extensions [6] which we use to distribute link utilization information within the network without the need for a custom protocol.

2.2 Related Work

There is already a lot of research regarding link sleeping to make networks more efficient, but most of it is based on assumptions that need to be revised in practice. Some research like [16], [12] or [14] assume that we can turn on interfaces within milliseconds, which we show to be around three orders of magnitude slower with today's hardware. It is possible that this assumption might have been correct in the past but we show that it is definitely not universally true today.

Another line of work does not consider the wake-up time but focuses on the algorithms to decide which links can be put to sleep like in [3], [5], [13] or [9]. Our work focuses on implementing link sleeping in today's systems and researching the problems and limitations that exist. GreenTE [16] formulated as being a TE problem suffers from a slower reaction time compared to our solution, due to the bigger complexity of collecting traffic matrices and running a more complex algorithm. While a more complex sleeping decision could also benefit our approach, it needs to be weighed against the slower reaction time. Thanks to our split approach by using a decentralized wake-up approach we could combine the benefits of fast reaction time in the case of congestion with a more precise sleeping decision as proposed in GreenTE.

The other unrealistic assumption in previous research is that they assume the controller can access things like up-to-date traffic matrices or even future traffic demands. One of the works that gets the wake-up delay of interfaces right but assumes access to future traffic demands is ElasticTree [8]. Assuming access to future traffic demands creates a favorable scenario for the controller that hides away a lot of problems that would occur in practice. For example, a sudden increase in traffic demand could overwhelm the network before the controller is able to react since he has no prior knowledge of the increase.

Even more research on link sleeping can be found in this survey [4] on green routing protocols using sleep scheduling in wired networks.

Chapter 3

Design

We need a controller to orchestrate the link sleeping, it is called the power plane and works next to the control and data plane. Its job is to turn links off, if possible, to save on transceiver power while taking care of minimizing network disruptions. This chapter goes into the design of the power plane and what requirements it has to fulfill. In addition, we present how we measure the link load in the network as well as how we remotely turn on and off links from the controller.

3.1 Power Plane Design

The power plane's overall mission is to put as many links to sleep as possible with minimal disruption to the network. In other words, the power plane has three objectives.

1. It must *not disconnect the network* by putting links to sleep.
2. It must *decide which links to put to sleep* while minimizing traffic redirection and congestion.
3. If congestion happens, it must *react quickly* and turn links back on to resolve the congestion.

While we aim to set as many links to sleep and keep them asleep for as long as possible, avoiding congestion takes priority.

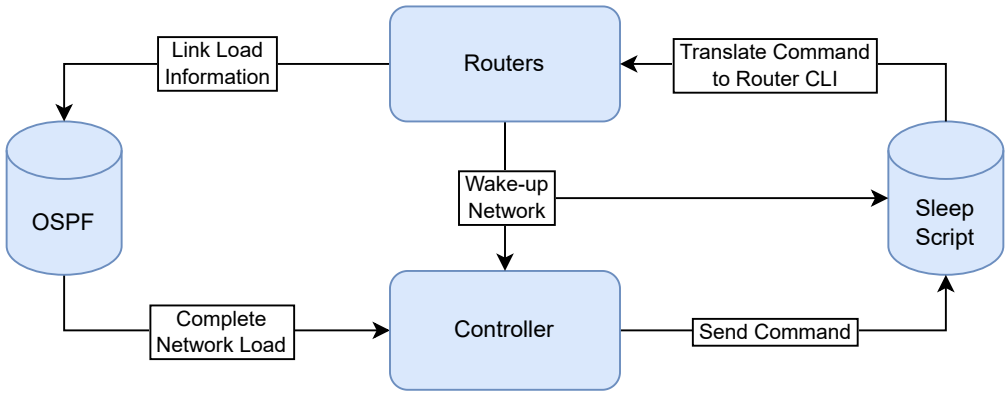


Figure 3.1: The interactions between the different components of the simple power plane, including the shortcut of the decentralized wake-up.

We consider a network running OSPF with a centralized power plane controller deciding which links to put to sleep and a decentralized wake-up mechanism allowing any node to initiate a network-wide wake-up. We use OSPF because it is a widely used intra-domain routing protocol that is capable of distributing network link loads through the OSPF TE Metric Extensions (RFC 7471); this gives the power plane a complete view of the link loads in the network, albeit with some delay.

Sleep Listing 3.1 summarizes the controller logic for putting links to sleep with Listing 3.2 showing the inner workings of the *check_reroute_capacity* function. The two parameters *max_util* and *safety_margin* can be used to tune the controller, valid values are between zero and one. A higher *safety_margin* can decrease the amount the network can sleep but helps with not disrupting the network. It helps the controller to have more headroom in case of traffic demand changes. A smaller *max_util* also decreases the amount the network can sleep but helps prevent the controller from going into a loop. If we reroute less traffic, there is less chance of rerouted traffic causing congestion somewhere else in the network, which would force us to wake up the network again.

A centralized controller easily avoids accidental network partitions and synchronizes the two sides of a link; the controller always knows which links are asleep and which must stay up to keep the network connected. Guaranteeing connectivity is non-trivial with a decentralized sleeping mechanism. The same is true for synchronizing the two sides of a link, since for a link to be up both transceivers must be on. Because the two transceivers are on two different devices, they need a way to synchronize the state of both transceivers.

Listing 3.1: Sleep Decision Pseudocode

```
# Power plane parameters
max_util = 0.2 # maximal utilization for putting links to sleep
safety_margin = 0.2 # minimum available utilization before wake-up

# 1. Calculate link loads from OSPF TE metrics
network_state = read_ospf_database()

# 2. Check link utilization and available bandwidth to reroute
for link in network_state.links:
    # Minimize rerouting: do not sleep links with a utilization higher than max_util
    if link.utilization > max_util: continue
    # Avoid congestion: set links to sleep only if there is enough capacity to reroute
    minimum_available = check_reroute_capacity(link, max_util)
    if minimum_available > link.utilization + safety_margin:
        mark_link_to_sleep(link)

# 3. Sort the links by utilization and avoid disconnecting the network
sleep_links = sort_by_utilization(links_marked_to_sleep)
for sleep_link in sleep_links:
    if disconnects_network(sleep_link): continue
    else: send_sleep_command(sleep_link)
```

Wake-Up The network wake-up is decentralized to speed up the reaction to congestion. Instead of waiting for OSPF to distribute the link load information in the network, each router can start a wake-up event independently. If a router detects high utilization or a link failure on its interfaces, it starts broadcasting wake-up commands within the network. High utilization in this case means

Listing 3.2: Reroute Capacity Pseudocode

```
def check_reroute_capacity(link, max_util)
    min_avail_list = []
    # Check both ends of the link
    for endpoint in link.endpoints:
        # Find smallest available link load toward other neighbors
        for neighbor, neighbor_link in endpoint.all_links:
            if neighbor in link.endpoints: continue
            margin = max_util * neighbor_link.bandwidth
            min_avail_list.append(max(0,neighbor_link.available - margin))
    return min(min_avail_list)
```

that the remaining capacity of the link is smaller than the *safety_margin*. The centralized sleeping controller also receives the wake-up message such that it does not try to turn off links simultaneously. This could happen because the controller has an outdated view of the network link loads. The sleep and wake commands are sent via TCP to a script running on the routers. The script receives those commands and translates them into router CLI commands turning the requested links on or off. Fig. 3.1 shows the interactions that happen between the different systems.

The distributed wake-up also helps in the case of a link failure, since each node knows if a link is asleep or not it can also detect if a link actually fails. In this case, it also starts broadcasting wake-up commands. So if the network should get disconnected due to a link failure, the network would reconnect if possible as soon as all links are up again.

To avoid the power plane from changing between waking up and sleeping too much, we pause the controller from setting links to sleep after we receive a congestion message. The parameter for how long the controller waits is set to 4 seconds in our case.

3.2 Link Load Measurement

We use a weighted average similar to the one used in the TCP bandwidth estimation algorithm to make the link utilization measurements a bit more robust. We not only use the measured link load at this moment but average it with the previously measured value using this formula:

$$linkload_t = 0.8 * linkload_{measured} + 0.2 * linkload_{t-1} \tag{3.1}$$

3.3 Sleep Script

For the controller to be able to turn off and on links in the network, we use a straightforward protocol where each router is listening on a port for commands from the controller. By sending one of three possible command types, the controller can turn on individual links, turn them off or turn all links on at once. This allows the controller to remotely change the link state on each router in the network.

Sleep Trick When turning links off, we use an additional trick to stop the router from sending more packets on a link that will turn off soon. We set the OSPF cost to the highest possible value first to discourage any traffic over that link before we physically turn the link off. This allows us to avoid a black hole where packets will get lost between the time it takes OSPF to detect that a link is down and the time it calculates a new path and installs that new route in the data plane.

Chapter 4

Experiment Setup

This section goes into the finer details of how the different evaluation setups are implemented. We have 3 different setups:

- The interface measurement setup to measure interface wake-up times
- The mini-internet [10] setup to emulate network topologies
- The hardware implementation to validate the network emulation

4.1 Interface Wake-Up Delay Measurement

We use two routers to conduct the interface wake-up delay measurements by sending pings over the interface until we receive an answer. The routers are connected to a server that orchestrates the measurements, see Fig. 4.1. The two routers used for this experiment are Cisco Nexus 9300 (C93108TC-FX) running NXOS 10.3.

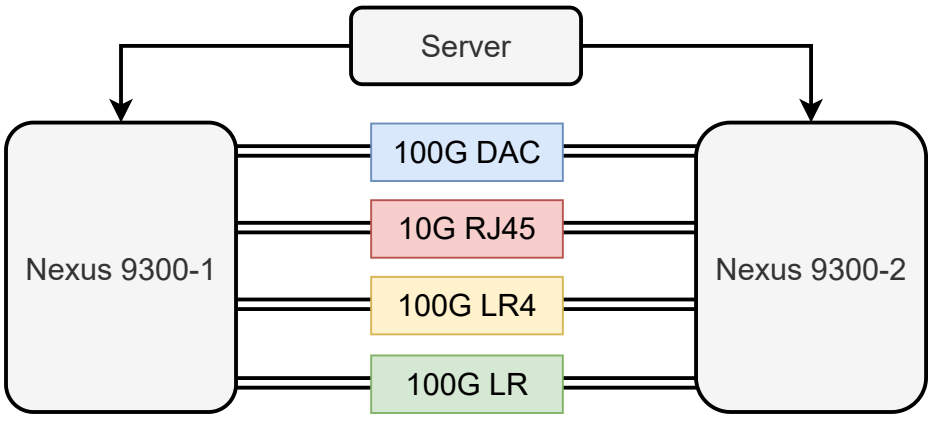


Figure 4.1: A script running on the first router turns on and off the interfaces and measures the wake-up time by sending pings to the other router until it responds.

Measurement The server starts a measurement run by establishing an SSH connection to one of the routers and runs a Python script on the Cisco router. This Python script will run multiple scenarios in one run, varying the time between shutting down a link and turning it back on. The idea is to see if the sleep time has any influence on the wake-up delay. The sleep times measured are between 0 seconds and 5 minutes. To ensure the router can not learn any patterns, we randomized the order in which the different sleep times are measured.

The script runs through the following steps to get one wake-up delay measurement.

1. It turns down the interface we want to test.
2. It waits for the specific sleep time we want to measure.
3. It starts sending ping messages every 100ms to the router on the other side of the link.
4. It turns up the interface again and measures the time it takes from giving the wake-up signal until the first ping message is returned over the interface.

Outliers Initially, we had some problems with the measurements being sometimes inconsistent and having significant outliers that we could not explain. We then tried to measure the wake-up delay with the auto-negotiation feature disabled. We suspected this to be the problem that caused the outliers, but it was not a problem with auto-negotiation. It turns out that there is a safety mechanism on our Cisco router that will count the number of times the link changed its state for a set period. If it happens more than a predefined limit, the router no longer turns on the link. By changing this setting to be more forgiving, we eliminated the largest outliers.

4.2 Network Emulation

To emulate a big network topology without needing access to a lot of hardware, we use a tool called mini-internet [10] that virtualizes network nodes as docker containers and connects them using virtual interfaces. The mini-internet is a tool developed at ETH Zürich that allows users to emulate a network.

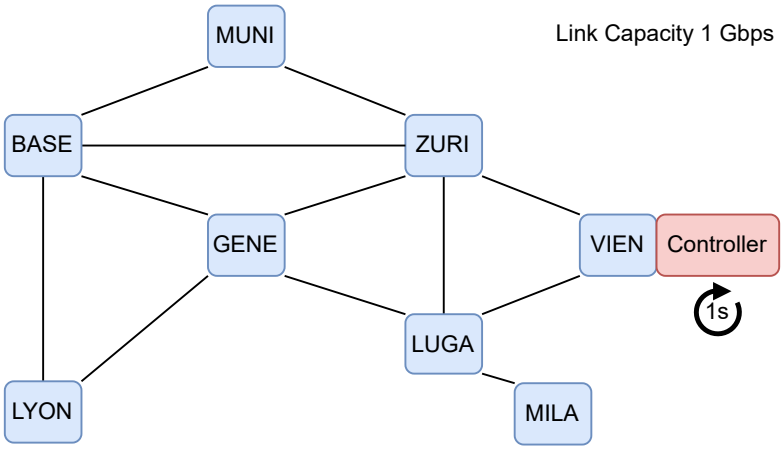


Figure 4.2: We use this topology since it is the default one used in the mini-internet. The controller is running in VIEN and makes decisions every second.

Setup Unlike network simulators, it uses the standard Linux network stack and can run open-source routing suites like FRR [2]. This is important for our application since we want our results to be as close to running on actual systems as possible because we are trying to find the limitations and problems that arise when running those link sleeping techniques in practice. The other nice thing that it allows us to do is to emulate link delays and different link buffer sizes thanks to Linux's included traffic control (tc) tools. They enable us to play with buffer sizes, while the traffic control network emulator (tc-netem) can emulate different link characteristics like delay and loss rate. For most of our experiments, we use the default topology used in the mini-internet, which is shown in Fig. 4.2. The links are 1 Gbps links with a delay of 10 ms and a buffer that is capable of buffering 100 ms of traffic. While we use only 1 Gbps links in our testing, this is only a technical limitation of the emulation, the target are links with a capacity of 100 to 800 Gbps. The controller is run on the VIEN node but this is arbitrary and could be changed easily to any other node in the network. In theory, one could also imagine having a primary and a backup controller to make the setup more resilient. We run the emulated network on a VM with 32 cores and 16 GB of RAM, the host system is equipped with an AMD EPYC Rome CPU. The VM handles all the docker containers for the nodes as well as traffic generation and everything else we need for the link sleeping system.

Additional Tools To test out link sleeping techniques on a network, we need to build some additional tools that help us distribute network utilization information to the controller and a way for our controller to send sleep and wake commands to the routers. Since we are trying to keep everything as practical as possible, we are also trying to use as many of the existing protocols and features as possible. This is why we decided to use OSPFv2 and the OSPF extension TE metrics [6] to distribute bandwidth utilization information in the network. Not only does OSPF take care of distributing information within the network for us, but it is also a widely used intra-domain protocol. The FRR routing suite has experimental support for the OSPF TE metrics extension, but the link load information is not automatically read from the links. This is why we use a simple script that reads out the link utilization periodically and then inserts this information into the OSPF extension to distribute it in the network. Since the FRR support for the OSPF TE metrics is only experimental and because of some strange behavior where it would not update the link load when waking up the link, we inject our own opaque LSAs into the network. The distribution of those opaque LSAs is still taken care of by OSPF but the initial creation of the LSAs is done by our link load measurement script directly.

OSPF Parameters To speed up the convergence process, we optimize some OSPF parameters. We set the hello interval to 100 ms, the dead interval to 1 second and set all the links to type point-to-point. The throttle timers for LSAs and shortest path calculations are set to an initial value of 10 ms increasing to a maximum of 100 ms. Those values are not set in stone and can be increased if necessary without much impact since the slowest part in the setup is still the wake-up time of the interfaces. Measuring the OSPF traffic volume with those adapted parameters reveals that the average load is 32 kbps per link. So even in our example with only 1 Gbps links it is just 0.0031% of the link capacity.

Delay We do not go into any delay numbers in this work because they highly depend on the topology, buffer sizes and the traffic pattern. Note that some of the flows will see an increase in end-to-end delay as the shortest paths tend to get longer if we set links to sleep.

4.3 Hardware Setup

To validate that our network emulation works similarly to a hardware-based implementation, we also implement one of the scenarios in hardware with the help of two Cisco Nexus 9300 routers. Since the scenario we want to compare contains eight routers, we need a way to have more than one node per physical router. We do this using multiple VRFs (virtual routing and forwarding instances) on our devices. Each VRF is equivalent to one node in the network and is also running its own OSPF instance.

VRF Setup Each VRF is independent of the other, so if we want to connect one VRF to another, we have to attach a physical cable to the router that is connected to one port that is assigned to the first VRF and to a second port on the same device that is assigned to the second VRF. To connect two VRFs between the two devices we have to plug in a cable between the two routers again making sure that the ports are assigned to the VRFs we want to connect. With this method, we can build the same network topology shown in Fig. 4.2 using only our two routers, which behave like eight independent routers connected through physical links. Fig. 4.3 shows how the nodes are split up between the two physical routers.

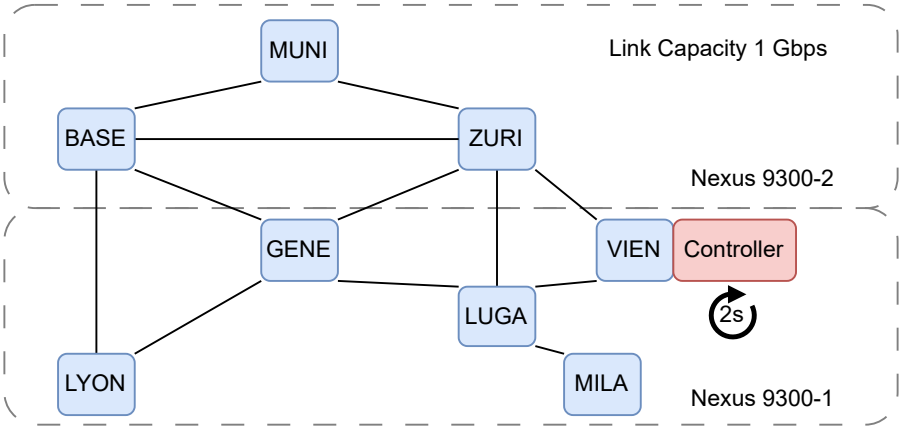


Figure 4.3: Each router runs multiple virtual routing instances with the help of VRFs.

TE Metrics Since our Cisco Nexus 9300 routers do not support the OSPF TE metrics extension, we must write a script to generate OSPF opaque LSA messages. We do this by running a script on each virtual router that will periodically check the link load and then craft an opaque LSA message to send to all its neighbors. Since the routers support opaque LSAs, they will distribute the LSA within the network in the same way as if the routers supported the TE metrics extension.

Sleep Script We again have a script running in each virtual router that accepts commands from the controller and can turn on and off links according to those commands. The only thing we changed to make it run on the Cisco routers is to exchange the CLI commands from the FRR syntax to the Cisco one without needing to touch the rest of the code.

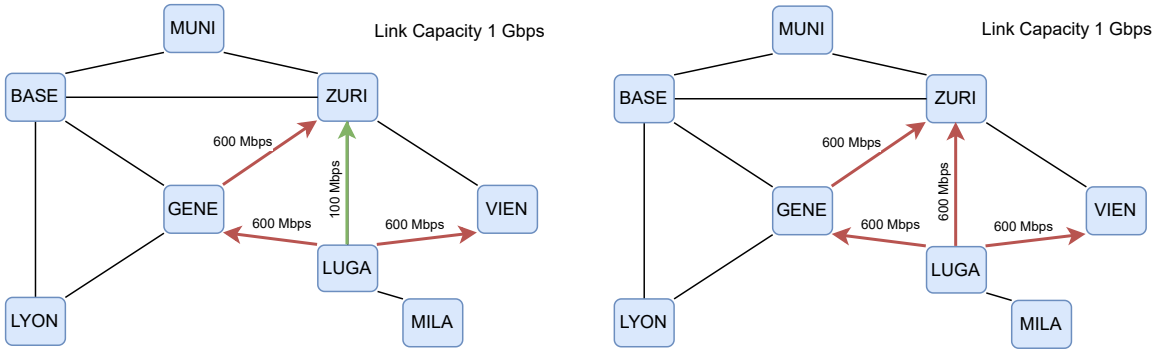
Controller Because the hardware implementation is a bit slower the controllers run every two seconds instead of every second. The reason for this is mostly the slow access time of the Cisco CLI. Just like the mini-internet version the controller again runs on the VIEN router.

4.4 Traffic Generation

We use Iperf3 to generate the necessary traffic for our experiments. A script sets up pairs of senders and receivers on hosts that are directly connected to the routers. We do this because running Iperf3 directly on the routers causes performance problems. The same is true if the generated packets are too small which is why we increase the packet size to 4000 bytes for the emulated setup. Without doing this, the Iperf processes have a tendency to slow down when congestion occurs even for UDP flows, due to the system getting overwhelmed. We use UDP traffic for most of the experiments because it is better at showing the available network capacity since it does not throttle down in case of congestion.

4.4.1 Handpicked Traffic Scenario

At this place, we introduce a specific traffic scenario we use to explore how the controller and network react to sending links to sleep. The reason to use this handpicked traffic is to make it easier to understand what is happening inside the network. It can get complicated quickly due to the interaction between shutting down links and rerouting traffic. The traffic scenario is designed to create three different conditions in the network, as illustrated in Fig. 4.4.



(a) Reduced traffic demand at 0-30s and 90-120s. (b) Increased traffic demand between 30-90s.

Figure 4.4

The first is a traffic load (Fig. 4.4a) that is low enough to allow the controller to shut down links. The second is an increased traffic demand (Fig. 4.4b) that causes the controller to react and wake up the network. The third condition is a reduction in traffic load (Fig. 4.4a) back to the beginning where the controller can shut down additional links.

The scenario has three traffic flows that stay constant and one flow that will vary to create those three network conditions. All the links shown here have a capacity of 1 Gbps. We have three flows of 600 Mbps traffic between LUGA → VIEN, LUGA → GENE and GENE → ZURI. The one flow that varies is between LUGA → ZURI. It starts with 30 seconds of 100 Mbps traffic that can be safely redirected over other links. Then, after 30 seconds, the flow increases to 600 Mbps for 60 seconds, which congests the network if the link LUGA → ZURI is asleep but causes no problem if the network is completely awake. After those 60 seconds, the flow will return to the original 100 Mbps and remain there for the rest of the scenario. The complete traffic scenario lasts for a total of 120 seconds. The controller gets started after 10 seconds to give the network some time to stabilize the traffic scenario and measure the link load accurately.

4.4.2 Random Traffic Scenario

In case we are running an experiment with randomly generated traffic, we use the following method to generate it. We first set the number of flows that we want to generate, together with the minimum and maximum duration that the flows should have. We then iterate and uniformly pick a source and destination node and resample until the destination is different from the source. Then we uniformly sample a start time within the experiment window and also uniformly sample a duration within the minimum and maximum set above. If the flow lasts longer than the whole experiment timeframe, we shorten the duration so that it ends at the end of the experiment run. The bandwidth of the flow is determined by sampling a number between 1 and 10 and then multiplying it with a factor that we can set to scale up or down the amount of traffic. So to get flows between 10 and 100 Mbps, we would set a factor of 10. We scale the traffic this way to ensure that the flows still have the same start time, duration, source and destination. This eliminates most of the variance between the scenarios when we only try to scale the traffic load. For TCP instead of setting a duration, we set the amount of traffic to be sent to $duration * bandwidth$.

Chapter 5

Evaluation

In this chapter, we go through the results of the link sleeping implementations. The evaluation is split into three parts; the first shows the transceiver wake-up delay measurements. The second and biggest part is based on the mini-internet implementation and explores different controllers, traffic loads, the influence of TCP, traffic demand changes, scaling to bigger network topologies and interface wake-up delays. The third part validates that the mini-internet implementation behaves the same as the hardware implementation at least for one specific scenario.

We present a selection of the results here:

- The transceiver wake-up delay is in the order of seconds and is independent of the sleep time (§ 5.1).
- TCPs congestion control mechanism can help avoid congestion during the network wake-up (§ 5.2.3) with minimal effect on the flow completion time (§ 5.2.4).
- The rate at which traffic changes, directly influences the link sleeping performance (§ 5.2.6).
- We can scale the link sleeping system to bigger topologies but might need to rethink some design decisions to improve the effectiveness (§ 5.2.8).
- Lowering the transceiver wake-up time is a major way to improve the performance (§ 5.2.9).

5.1 Interface Wake-Up Delay

Setup For this experiment, we use the setup described in § 4.1.

Results Previous work assumes interface wake-up times of a few milliseconds, but our measurements show that it is on the order of seconds. We measure different types of interfaces and find that 100G optical transceivers take roughly 8-15 seconds to wake up, while copper-based ones are a bit faster, taking around 2-8s (Fig. 5.1).

We also confirm that wake-up delay is independent of the sleep time; *i.e.*, even if sleeping for just 50 ms, it takes seconds to wake up the transceiver. This means that all the state is lost on turning off the link and it takes the transceiver the same time to go through the start-up procedures.

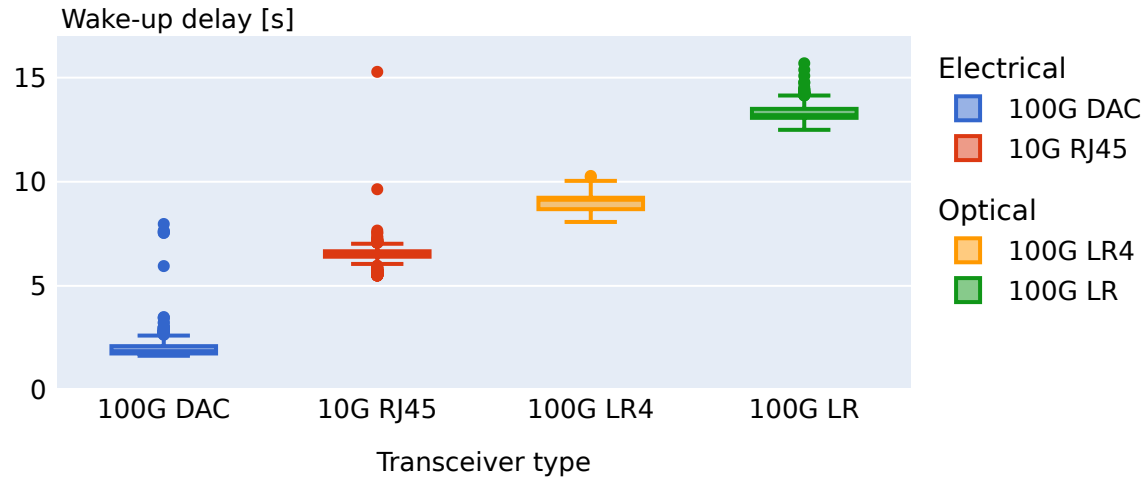


Figure 5.1: The wake-up time is not a few milliseconds as assumed in previous research but in the order of seconds. The box spans from Q1 to Q3 with the line marking Q2. The whiskers correspond to the box edges +/- 1.5 times the interquartile range.

Conclusion In summary, one should be careful when turning links off, as reverting the decision takes a long time. This might not be necessary if we had a way to tell the interface to go into a sleep state instead of completely turning off the power. It could allow the transceiver to keep some parts alive and reuse the state on wake-up, potentially saving time. But it is currently unclear how much energy savings that would enable.

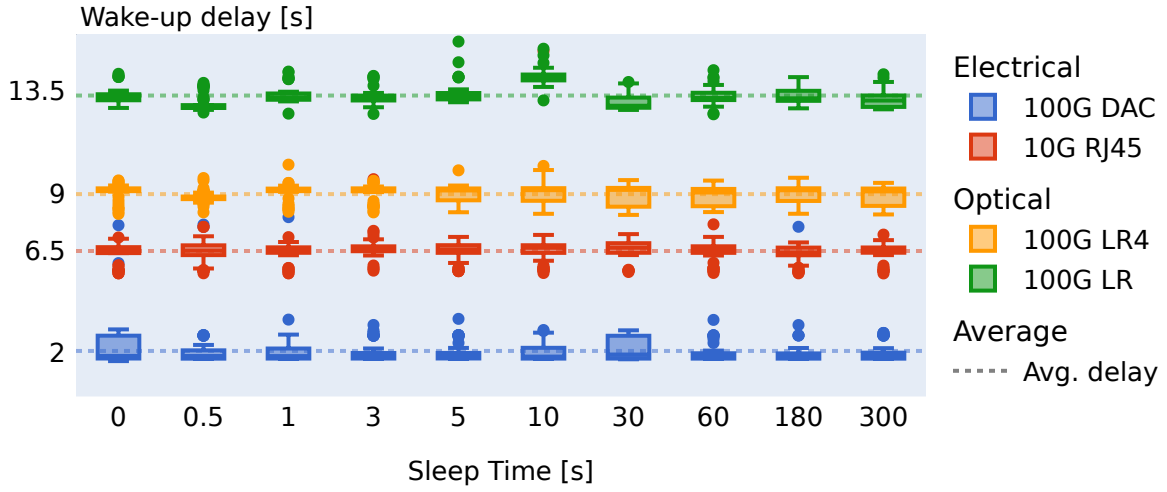


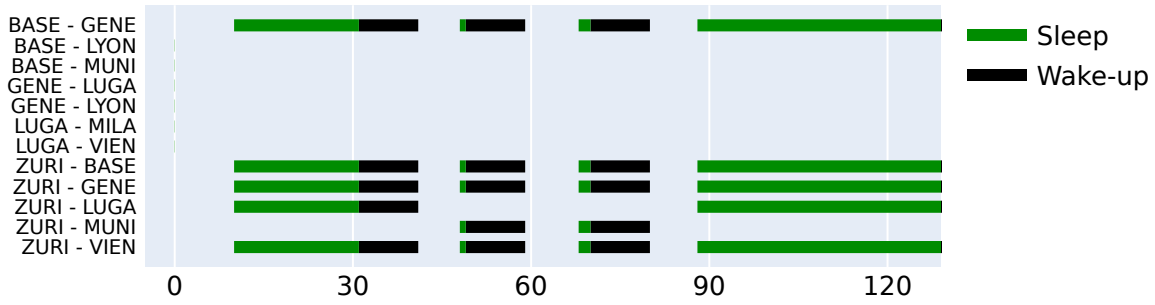
Figure 5.2: The wake-up time is independent of how long the transceiver is asleep.

The long wake-up time also strongly discourages sleeping and waking up on a per-packet basis. Even buffering packets while the interface wakes up, as suggested by [12], is not feasible with current hardware. Buffering packets for 10 seconds overflows even the biggest buffers. Secondly, congestion control like TCP would already assume the packet is lost, due to the timeout being reached, even if we had big enough buffers.

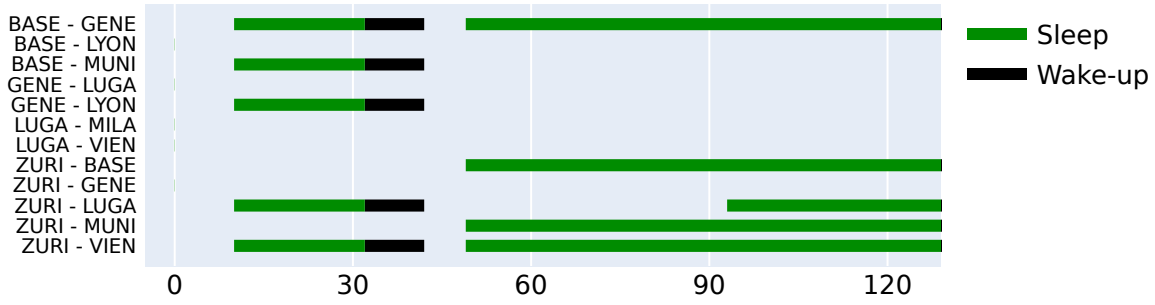
5.2 Mini-Internet

5.2.1 Controller Optimization

Setup In this evaluation, we show that we need to be careful about which links we send to sleep to not end in a loop where we turn off links, cause congestion and have to wake up repeatedly. In short, we prioritize links that carry little traffic (see § 3.1). This heuristic appears efficient enough in our evaluation. Ideally, the controller would detect such loops and discourage turning down those specific links at least in the short term while the traffic demand is still the same.



(a) Baseline controller goes into a loop because it chooses bad links to set to sleep.



(b) Improved controller finds a better set and can keep them off until the end

Figure 5.3

The baseline is a controller that only checks if a link can be shut down without optimizing which links to prioritize. The controller detects that a link can be shut down by checking the other interfaces of the two routers connected by the link in question. If the smallest available bandwidth on the other interfaces is able to handle the traffic over the link, it is considered a sleep candidate. The difference between the two controllers is that the improved controller sorts them by their link utilization and discards any that are already utilized more than 40% even if rerouting would be possible. This is a parameter that can be tuned to reduce the risk of turning off links that congest the network with their rerouted traffic. Since all the traffic of a link needs to be rerouted onto another path, this rerouted traffic could cause congestion somewhere else in the network. This would cause the network to wake up again, returning to the original state in which case the controller would again turn off the link if there is no traffic demand change in the meantime. By decreasing the amount of traffic that needs to be rerouted, we can reduce the risk of congestion somewhere else in the network. If we had a more detailed view of the network like

knowing the source, destination and traffic amount for each flow, the controller could in theory calculate on which path the traffic gets rerouted. Since we do not have access to this information in our setup, the controller cannot predict if the rerouted traffic causes a problem somewhere else.

Results The difference between the two controllers can be seen quite well in the sleeping pattern with the handpicked traffic scenario described above in § 4.4.1 The improved controller can adapt to the increased traffic demand by waking up the network and adjusting the sleeping links accordingly (Fig. 5.3b). Meanwhile, in Fig. 5.3a we can see that the baseline controller wakes up the network and does not pick good links to sleep, which congests the network shortly afterward and keeps waking up until the traffic demand is low again. While this could also happen to the improved controller, it is far less likely because the improved controller aims to limit the total volume of rerouted traffic.

Conclusion A safer but much slower approach would be to shut down links individually and wait for the link load updates before turning the next link to sleep. This is something we might explore in future research. Ideally, we would have detailed load information with both source and destination node information since this would allow us to calculate precisely what route the rerouted traffic would take after a link is shut down. This would eliminate the problem of rerouting traffic that causes congestion somewhere else. Such a system would be similar to a traffic engineering solution where we collect the traffic matrix at regular intervals. Due to the increased complexity, this would probably mean that the controller will have a more accurate but also more delayed view of the network. The benefits and drawbacks of such a controller need to be investigated in further detail.

5.2.2 Traffic Load

Setup In this section, we vary the traffic load to see how it affects the network's ability to sleep and how it impacts network congestion. For each traffic amount, we run the experiment and look at the results over a span of 120 seconds. The traffic in this case is created randomly which is also why at higher average traffic loads even the completely awake network struggles with the amount of traffic. Since we generate the traffic randomly, we are not checking if the network can handle the amount even without shutting links down. This is why we plot not only the loss but also the loss without a power plane to show the difference. See § 4.4.2 for how the random UDP traffic is generated.

Results We see in Fig. 5.4 that even for higher traffic loads, the additional loss due to sleeping is minimal. One of the reasons is the fact that there is no traffic step like in the handpicked case and so the controller has some time to react to the increase in traffic and can therefore wake up the network before loss occurs. We can also clearly see that the amount of time we can sleep decreases with an increase in traffic demand which is expected. This is actually a positive thing since a good controller should try to follow the demand pattern with the number of links that are awake. We can also see that with the topology we are testing, even without any traffic, we cannot have the network operating below 60% of network links awake. The reason is that we need the network to be connected so there is a lower bound on the number of links that can be asleep.

Conclusion The controller is capable of following the traffic demand and scales the amount of time spent sleeping according to the traffic load present. There is an increase in loss due to the network being asleep when we increase the traffic demand. We believe this to be caused, at least in

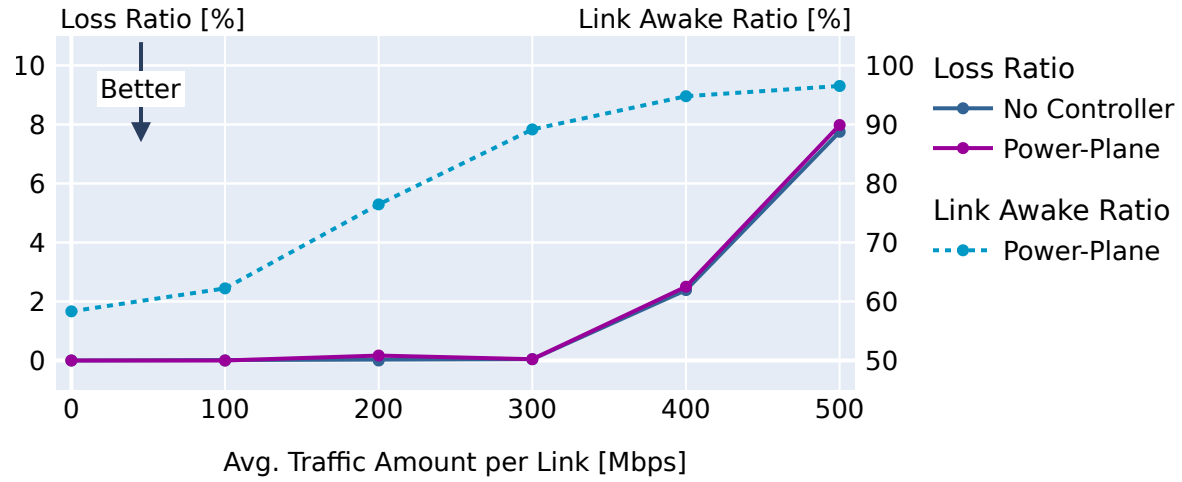


Figure 5.4: Increased traffic demand will decrease the amount of time the network is asleep. The difference in loss between the power plane and the fully awake network is small.

part, by the way we scale the traffic demand. Since we scale the bandwidth of the flows, this also means that there are bigger jumps in traffic demand, making it harder for the controller to react to the increase.

5.2.3 TCP and UDP Traffic

Setup A mix of UDP and TCP traffic helps us avoid loss thanks to congestion control but comes at the cost of reduced bandwidth for the TCP flows during the wake-up phase. The TCP flows can recover some of the lost bandwidth as soon as the network is awake again by increasing the sending speed when the congestion is over. We again use the handpicked traffic scenario and keep the traffic amount the same but vary the ratio of TCP and UDP.

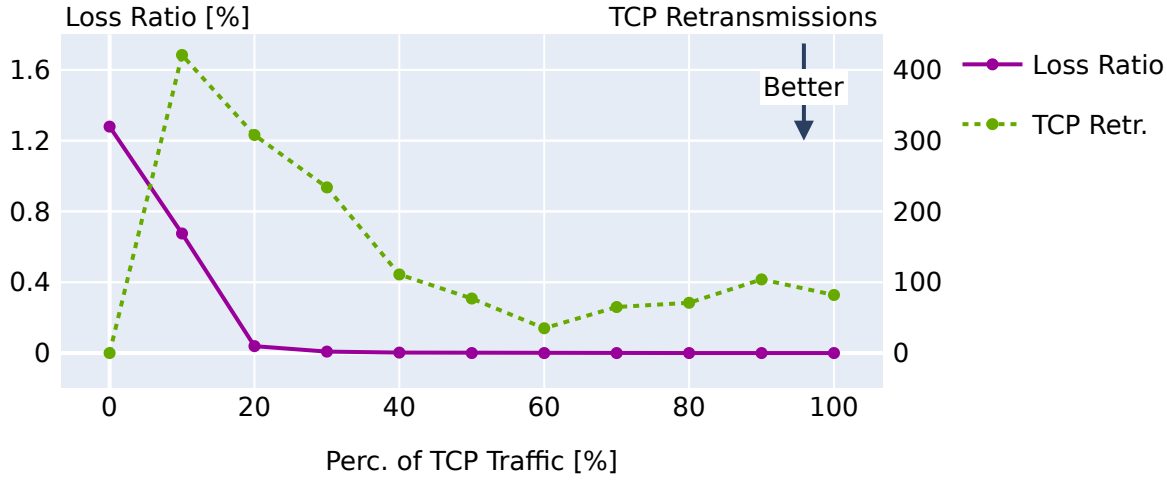


Figure 5.5: Loss in UDP flows disappears above 20% due to TCPs congestion control working

Results Our measurements show that the loss reduces basically to 0 as soon as more than about 20% of the traffic is TCP (Fig. 5.5). The reason for this is that during congestion we overuse the links by roughly 200 Mbps when using the handpicked traffic scenario. So to limit congestion on those links, we need to have at least 200 Mbps of this traffic to be TCP since UDP will not throttle when loss occurs. Calculating 200 Mbps / 1200 Mbps we find that the amount of TCP necessary is about 16%. So above this value, we expect to see less congestion and therefore also fewer UDP packets getting dropped. This is not a hard border where above this threshold we automatically see 0% loss, but we can see that the border between loss and barely any loss happens somewhere between 10 and 20 percent as predicted. The same trend can be seen in TCP retransmissions, where if we have more TCP traffic the congestion can be resolved faster resulting in fewer dropped packets and therefore fewer retransmissions. The TCP retransmissions plateau above 50% because for TCP to detect the congestion packets need to be dropped first.

Conclusion Since a big portion of the internet traffic is based on TCP, shutting down links might not be as costly as expected. The question to answer will be if it is acceptable to have a reduction in bandwidth for a few 10s of seconds while the network is waking up. Keep in mind that the reduction in bandwidth depends on the demand change and your safety margin. Even in our handpicked case where we have big traffic demand changes the bandwidth is only reduced by 10-20% while the network wakes up.

5.2.4 TCP Flow Completion Time

Setup 1 While the TCP congestion control helps with packet loss by throttling down to avoid congestion, we expect the flow completion time to increase because of that. For this experiment, we again use randomly generated traffic as in § 5.2.2 but using TCP flows instead. To see how we generate the random traffic look at § 4.4.2.

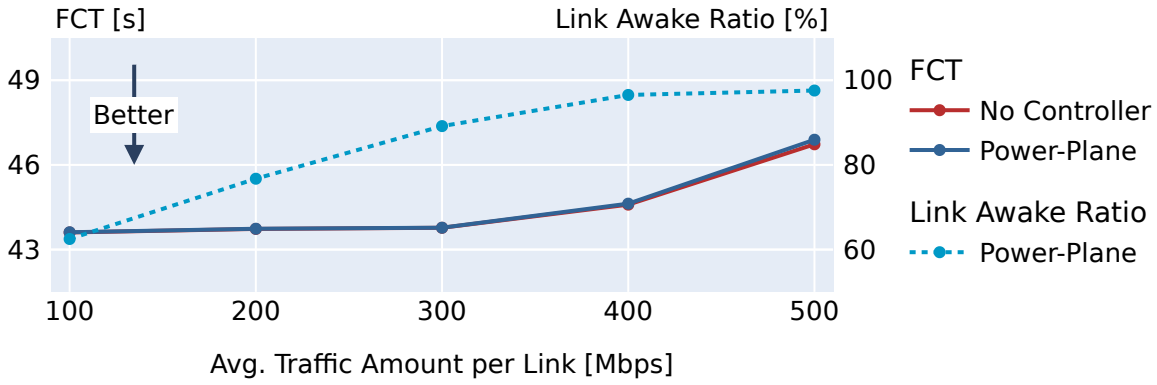


Figure 5.6: The increase in FCT looks similar to the loss increase in Fig. 5.4. The biggest increase in average FCT is under 200 ms compared to the fully awake network.

Result 1 We find that inducing such a scenario where the FCT (flow completion time) increases by a significant amount is hard. Because if there is not much traffic load, there is no loss where TCP would need to throttle down. At the same time, if the traffic load is constantly high, the network is always awake so there is also barely any increase in FCT. This is shown by using the same traffic as in § 5.2.2 but changing it to TCP. The results in Fig. 5.6 are similar to the UDP

case but the relative increase is ten times smaller. While UDP packet losses are final, in the TCP case some of the flows can recover lost bandwidth after the network is awake again. They can do that by sending at a higher rate after the full network capacity has been restored.

Setup 2 To observe an increase in FCT we had to increase the size of the flows to be between 50 and 500Mbps and make them shorter. We use 150 flows during a time of 120s with each flow lasting between 1 and 20 seconds. The reason for using bigger flows is that we want to overload the network during wake-up as much as possible so we need bigger jumps in traffic demand. The reason for shorter flows is to limit the flow's ability to recover the lost bandwidth when the network is awake again.

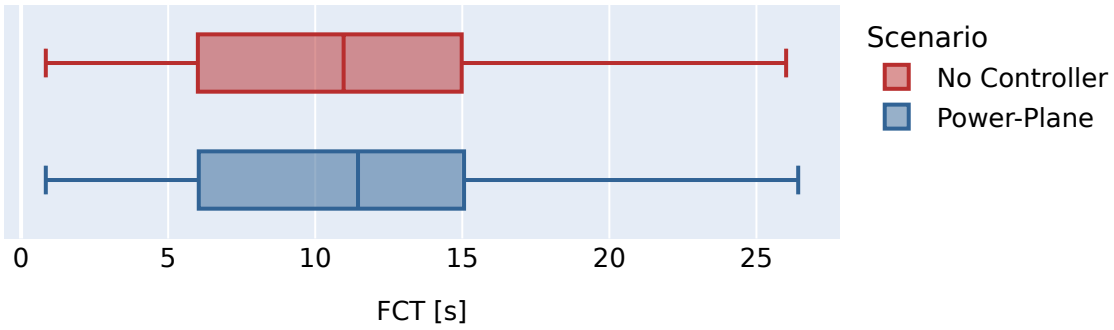


Figure 5.7: The FCT only increases by a small amount even though one would naively expect the FCT to increase by the wake-up time of the network. The box spans from Q1 to Q3 with the line marking Q2. The whiskers correspond to the box edges +/- 1.5 times the interquartile range.

Result 2 Only by crafting fast-changing traffic loads, we can see an increase in FCT. As we show in Fig. 5.7 the median FCT only shifts by around half a second or an increase of about 5%. The reason for this is that while the TCP flows have to throttle to avoid congestion, they do not have to stop sending until the network is fully awake but just throttle enough to avoid congestion. So if one link is overloaded by 10%, the flows only need to throttle by 10% until more capacity is available again. The increases in FCT would be around $100/90 = 11\%$ since the flows can still send at 90% of their bandwidth. While one may expect the FCT increase to be in the same order as the network wake-up time, this is clearly not the case.

To see how the individual flows are affected Fig. 5.8 shows the flow completion time in green for the case of a fully awake network. The red markings show the additional time the flows take with the power-plane enabled. The blue elements show flows that finish faster with the power-plane active. This can happen in pairs where one flow will get slower while another one gets faster. This is due to one flow getting assigned more bandwidth by chance in the case of congestion; *i.e.*, there are more dropped packets in one flow compared to the other. The dark grey bars mark the times at which the network is not completely awake. This includes the times the network is asleep and the roughly ten seconds it takes to wake it up.

Conclusion It takes a very specific traffic scenario to observe FCT increases. While we can induce scenarios where the FCT increases for TCP flows we also show that the increase is not in the order of the wake-up time as one might expect. Because while the network is waking up, the

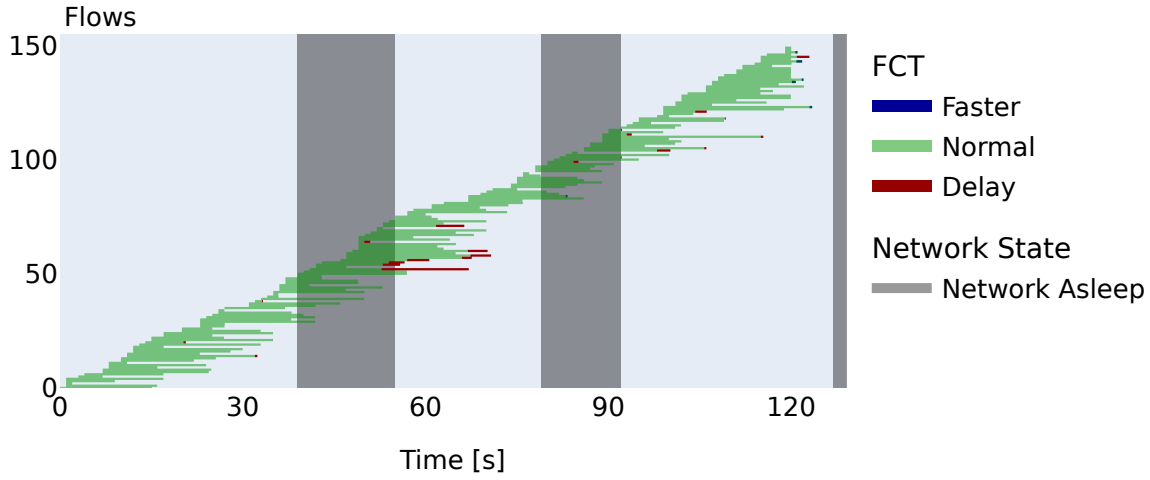


Figure 5.8: Flows that run while the network is not fully awake can see an increase in FCT that occurs due to less network capacity being available while the network is waking up.

flows can still send traffic just at a reduced rate. This increases the FCT only by the amount that the flows have to slow down until more capacity becomes available through waking up the network.

5.2.5 Traffic Demand Oscillation

Setup In this experiment, we show what happens if we encounter an oscillating traffic demand with different frequencies. To show the effects different traffic demand change patterns can have we adapt the hand-picked traffic scenario to incorporate those patterns with one changing flow. The traffic profile for the flow from LUGA→ZURI is shown in Fig. 5.9, including the ramp profiles for § 5.2.6. The other three flows stay the same as in the handpicked case. We use only UDP flows for this experiment.

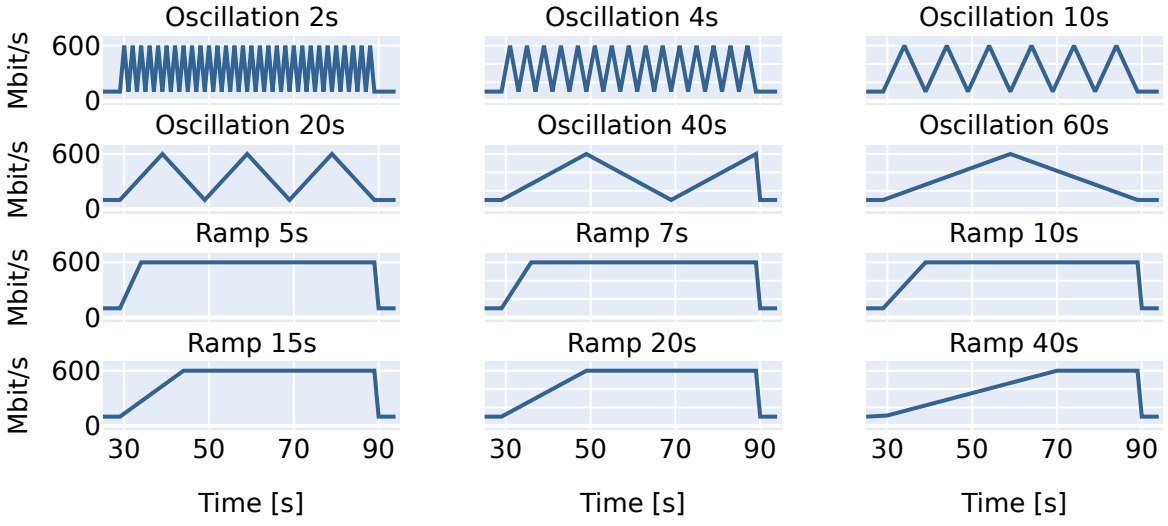


Figure 5.9: Selection of tested traffic patterns, both oscillations and ramps

Results Looking at Fig. 5.10 we can spot two interesting maxima, one for the loss ratio at an oscillation period of 20 seconds and one maxima at an oscillation period of 10 seconds for the network's ability to sleep. It is no coincidence that those two relate to the time it takes the network to wake up.

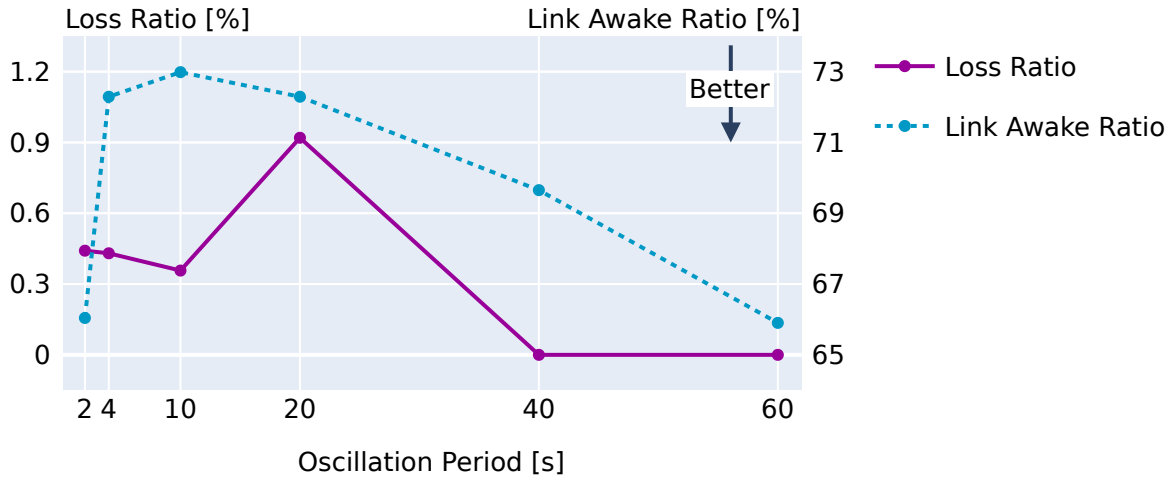


Figure 5.10: Having traffic oscillations with periods close to the interface wake-up delay creates unwanted interactions with the controller.

The first maxima can be explained by the fact that this period is double the time it takes to wake up the network. So in the first half of the period, the traffic is high while we are waking up the network, incurring losses during this time. In the second half of the period, the traffic demand is low meaning we turn the network off again. This means the network is always sleeping at the beginning of a new period, such that we incur congestion losses every period, explaining the high loss ratio. The reason for the lower losses for lower frequency oscillations is that the controller can react to the changing traffic demand condition and wake the network up fast enough so that we do not see any loss for slow oscillations. For higher frequencies, the network does not really react to the changes anymore. But because the frequency increases, it can buffer more of the burst and smooth out the demand change resulting in lower losses. So similar to a control system the worst-case scenario is oscillations roughly on the time scale of the control loop.

The reason for the second maxima is that in this case, we turn on the network for 5 seconds and keep the network up a bit longer because we do not want to turn off the network too fast again. It happens that in this case we sometimes turn the network off for only 2 seconds and then turn it on again. So the network is basically always on, explaining the high average awake ratio.

Conclusion In network conditions where we have traffic demand oscillations that are on the same timescale as the network wake-up delay, we expect the controller to perform worse. It is similar in nature to control systems where ideally we want the control loop to be faster than whatever it is we are trying to control.

5.2.6 Traffic Demand Ramp

Setup In this part, we investigate how different traffic demand ramps influence the controller's ability to avoid congestion. The worst-case scenario for the controller is a traffic demand change in the form of a step function because there is no prior signal that would allow the controller to react

to this increase in traffic demand before it happens. Luckily, this is rather unlikely to happen in networks carrying lots of aggregated traffic because, in practice, changing the load of a 400 Gbit link by multiple 10s of percent needs multiple hosts to increase their traffic in a coordinated manner. The probability of multiple nodes needing a multiple of their normal traffic demand simultaneously becomes rather unlikely.

What is a more realistic change in traffic demand is a ramp of traffic where the traffic demand slowly grows over a certain time *e.g.*, during the morning when more and more people start to work. The different ramp profiles used for this experiment can be found in Fig. 5.9. We again use only UDP flows for this experiment.

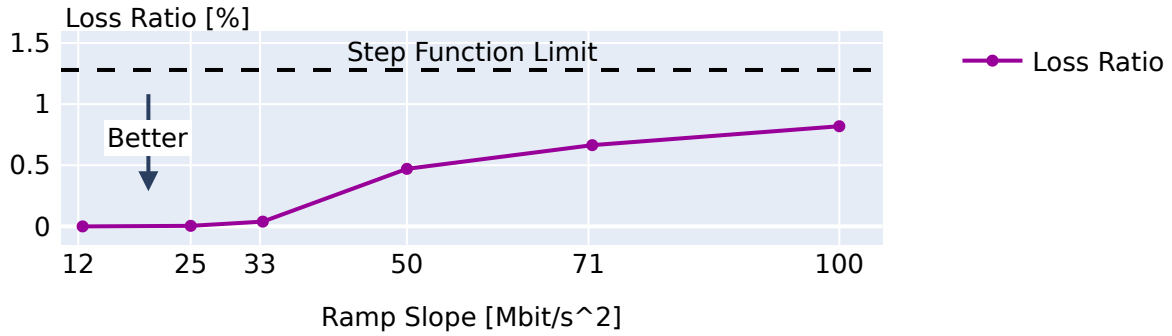


Figure 5.11: Steeper ramp slopes increase the loss since the controller cannot react fast enough.

Results Assuming traffic demands in practice do not change like a step function but more like a ramp, we can see that the controller can react and wake up the network if a link starts to see high utilization and can avoid the congestion occurring if the traffic increase is slower than the wake-up time Fig. 5.11. The step function limit line marks the loss if we have a ramp with slope infinity or in other words a step function.

Conclusion We can put this into a mathematical formula where m is the gradient of the traffic demand ramp, T is the total wake-up delay of the network, S is the safety margin of the link and B is the link bandwidth.

If Eq. (5.1) holds the controller can react fast enough to the change in traffic to avoid congestion.

$$S * B \geq T * m \tag{5.1}$$

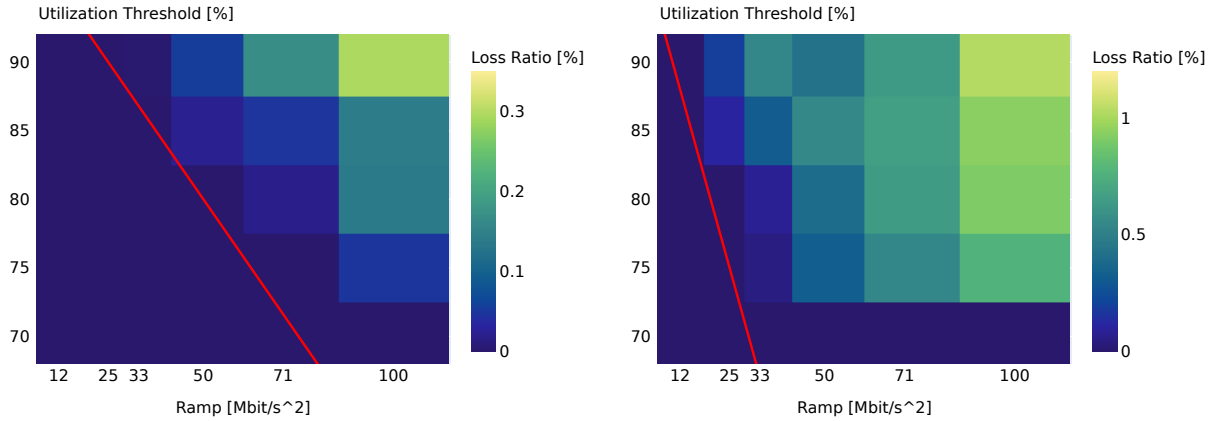
where:

S is the Utilization threshold $\in [0, 1]$
B is the Link Capacity in Mbps
T is the network wake-up time in seconds
m is the Slope of the traffic ramp in Mbps/s

Attention this does not apply to cases where the controller shuts down links that will cause congestion due to rerouted traffic. If we turn off links, their traffic will be rerouted by OSPF. Since this happens pretty much instantly there is going to be a jump in load on the other links which will look like a step function. If this jump in load is higher than the available capacity of the links, it will cause congestion.

5.2.7 Ramp Equation

Setup We confirm the validity of Eq. (5.1) by running an experiment where we vary the ramp slope, utilization threshold and interface wake-up delay. The red line in Fig. 5.12 is the border at which the condition holds, described by $S * B = T * m$. The x-axis in this case is m, the y-axis is S and the link capacity B is 1 Gbit/s. There are two plots, one for T=4s and one for T=10s. We again use the handpicked UDP traffic scenario described in § 4.4.1.



(a) 4s Interface Wake-up Delay

(b) 10s Interface Wake-up Delay

Figure 5.12: Heatmap confirms that the above equation (red line) approximately describes the border where no traffic disruptions occur, the further away we get from the line towards the upper right the more loss occurs

Result Fig. 5.12 shows that Eq. (5.1) approximately describes the border condition between traffic loss and no traffic loss. Due to the discrete ramp slopes and utilization thresholds, the line does not match up exactly. Another important point is that there is no loss for the utilization threshold set to 70% since at that point the link between LUGA and ZURI stays awake.

Conclusion We experimentally validate this equation by changing the different parameters and measuring the amount of loss that occurs in the network. We can see that the condition marks the border at which point loss starts to appear. We can also see that going further away from this line will increase the loss.

5.2.8 Topology Scaling

Setup To show that we can also increase the topology from only eight nodes to more we use the géant network [1]. We use the topology that consists of their fiber and spectrum links (Fig. 5.13). We configure those links to have a capacity of 1 Gbps and a delay of 10 ms. This network topology is about 4 times the size both in number of nodes and links compared to the topology used in the other experiments. For the traffic load experiment, we again use randomly generated UDP traffic as described by § 4.4.2.

To measure the controller computation time we let the network run without any traffic for 120s and measure the average execution time of the controller. We run this experiment without any traffic to not influence the computation time since the traffic generation could take away computing

resources from the controller. The controller still goes through the same computation steps even if there is no traffic.

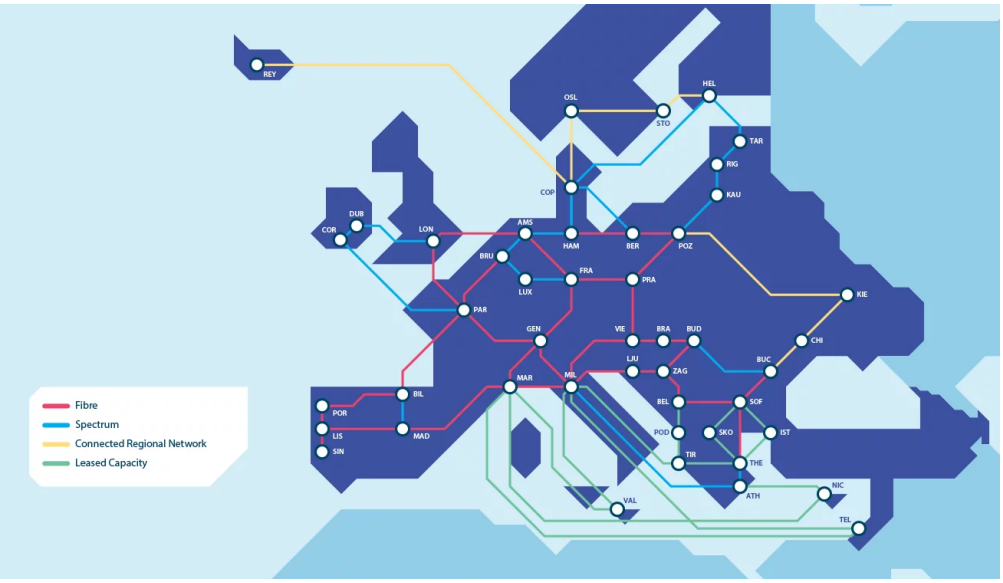


Figure 5.13: The géant network topology as found here [1]. We use the topology that is marked by the red (Fibre) and blue (Spectrum) links.

The OSPF convergence time is calculated by measuring the time it takes for a ping to change to the new path. We go through the network and measure the convergence time for each link and average the results. For each link, we run a ping for all pairs of nodes where the shortest path includes this link. The link is tested by:

1. turning the link off
2. starting a ping with an interval of 10 ms
3. turning the link back on
4. measuring the time it takes for the ping to switch paths

We then measure the convergence time of that link by taking the maximum out of all the measured pings. We expect pings originating further away from the tested link to take longer to switch paths. Since we are interested in the total convergence time of the network, we want to measure the time until the last path is switched.

Topology	Nodes	Links	Controller Computation Time	OSPF Convergence Time
Base	8	12	81.03 ms	384.6 ms
Géant	35	48	92.16 ms	556.7 ms

Table 5.1: The OSPF convergence time is still below 1s even for the bigger topology and the controller can still be executed faster than 100 ms. While the OSPF convergence time is acceptable for today's interfaces, if the wake-up delays get closer to 0 s, we would need to speed up this process.

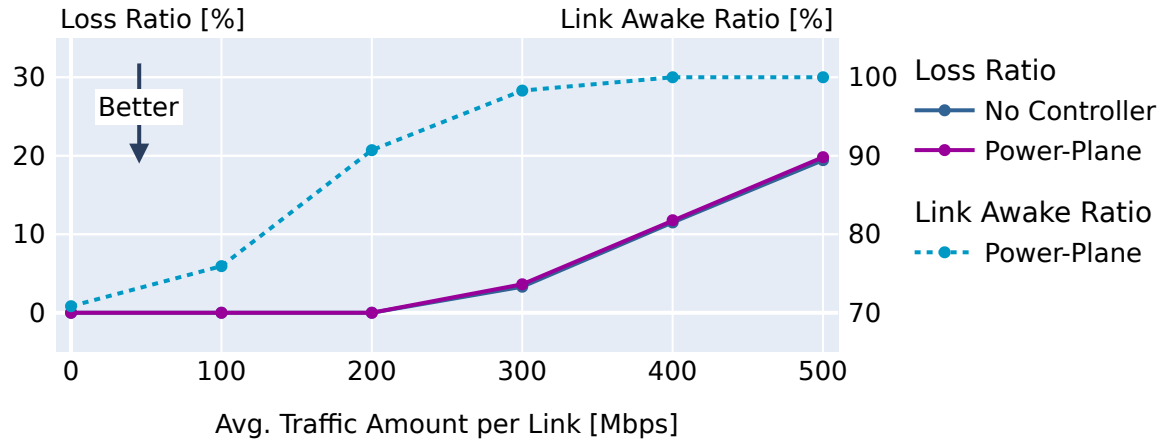


Figure 5.14: Traffic load experiment for the simplified géant topology. The difference in loss between the power plane and the fully awake network is again similar to the smaller topology in Fig. 5.4.

Result We show in Table 5.1 that increasing the number of nodes from 8 to 35 increases the worst-case OSPF convergence time by about 200 ms. The controller computation time increases from 81 ms to 92 ms. Fig. 5.14 shows that even for a bigger topology the power plane does not cause much more additional loss.

Conclusion The increase in convergence time comes partly from the larger network diameter growing from 40 ms to 100 ms and partly due to the computation of the shortest path taking longer in the bigger network. This convergence time is acceptable today with the interfaces taking an order of magnitude longer to wake-up. If interfaces with faster wake-up delays become available in the future, this might change. One way to speed up this process would be to adapt OSPF with an additional state that indicates that a link is sleeping. In the case of a wake-up event OSPF could then already have a pre-computed routing state for the fully awake network ready.

The additional loss due to the power plane is again small and matches the experiment for the smaller topology in § 5.2.2. The bigger difference can be seen in the link awake ratio. They are not directly comparable since the network topology plays a big role in how many links can actually sleep. For example, even without any traffic, the bigger topology can only reduce the links that are awake to 70% due to the fact that we want to keep the network connected at all times. But even then we can see that the link awake ratio increases faster and more steeply than in the experiment with the smaller topology. The reason for this is that no matter where there is a problem of congestion in the network, our simple power plane will wake up the entire network. So the controller has to keep the network awake more often since the chance of having congestion somewhere in the network increases with the size. One might have to rethink the decision of waking up the whole network in case of congestion when scaling the system to bigger topologies. One possibility would be to separate the network into multiple regions and then only wake up the region where there is congestion happening. This could also be more dynamic, such that each node will wake up only the nodes that are no more than a certain amount of hops away.

In conclusion, we find that it is possible to scale our system to bigger topologies. While the performance is still good for smaller traffic loads, one might have to revisit some previous design decisions to optimize the performance for larger topologies; *e.g.*, waking up the whole network when congestion is detected.

5.2.9 Future Interfaces

Setup Since we are working with virtual interfaces, we can reduce the wake-up delay to 0s and investigate the improvement we would see if we had transceivers that could wake up nearly instantly. We are using the same handpicked UDP traffic scenario again.

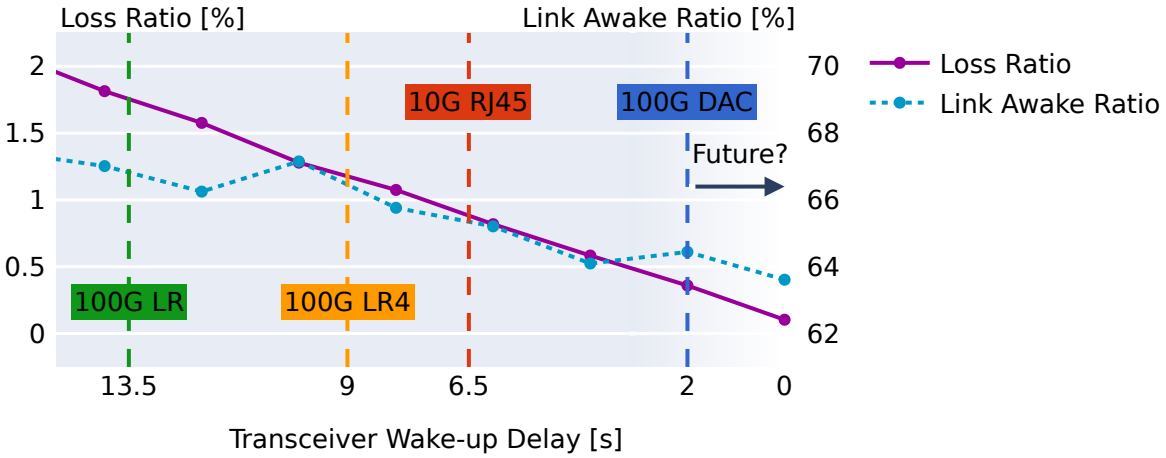


Figure 5.15: Faster waking interfaces create less disruption to the network and lets the controller turn links off faster again.

Results We can see that the network reacts much faster to the change in traffic. It is so fast that barely any traffic gets lost due to congestion because the wake-up of the whole network is so fast that the congestion only occurs for about 1s. But even improving the wake-up delay by a few seconds can already be a big help. We see a linear decrease in loss by changing the wake-up delay. This is because the loss we see stems from the congestion and the faster the link can wake up the faster this congestion is resolved. This goes back to the formula presented above where decreasing the link wake-up delay brings us closer to achieving the condition, thus reducing the amount of network congestion.

Conclusion Because the congestion is resolved faster, there is also an improvement in the amount the network can sleep. Because as soon as the congestion is dealt with the controller can already start with turning off links. So we show that if we had access to interfaces that can wake up faster, we could dramatically increase the performance of link sleeping.

5.3 Hardware Verification

Setup We implement our controller on hardware to validate the results from the mini-internet setup. The hardware setup is described in § 4.3. The scenario contains the same base UDP traffic scenario as described above in § 4.4.1. We also adapt the parameters to be the same on both implementations. For one the controller runs only every 2 seconds instead of every second due to the hardware implementation being slower. Another change is that we had to set the OSPF hello interval to 1 second since our hardware implementation does not support fast OSPF hello packets (<1s hello interval).

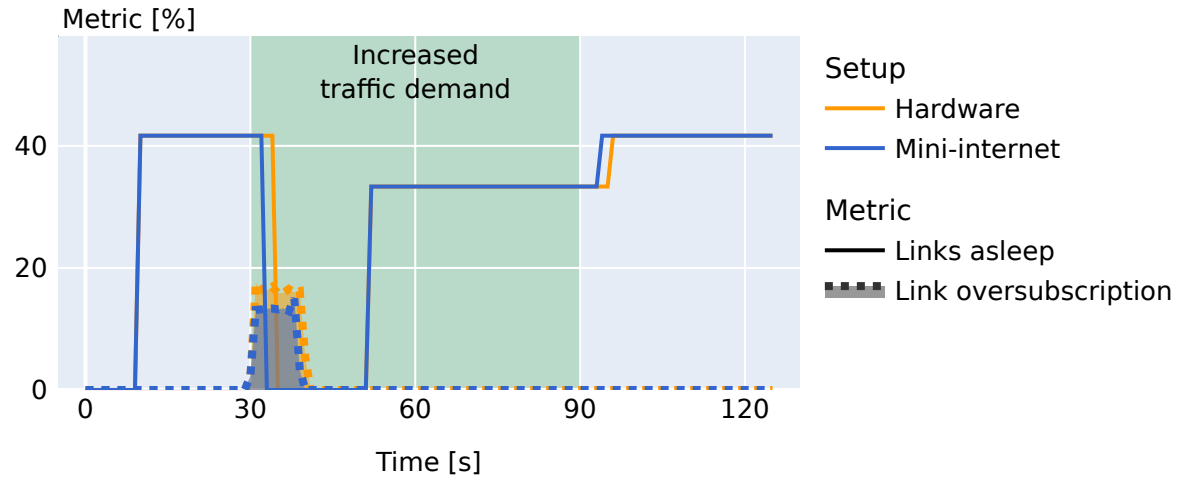


Figure 5.16: The Sleep and Congestion pattern between hardware and emulation match.

Result We confirm that our power plane controller behaves similarly in our hardware and emulated setups. As we see in Fig. 5.16 the two implementations follow the same patterns regarding how many links can be put to sleep and also incur the same kind of loss.

Conclusion This suggests that (i) we can design power plane controllers that are compatible with today's hardware and standard protocols; (ii) we can hopefully rely on emulation to experiment with more complex topologies and traffic patterns.

Chapter 6

Outlook

6.1 Use Cases for Link Sleeping

We laid out some first steps for understanding the dynamics of link sleeping. We present an overview of the characteristics of traffic dynamics that matter most for making sleeping work with today's hardware. We identify the traffic demand ramp as an essential characteristic that is fundamental to the fact that today's interfaces take multiple seconds to wake up. Especially for networks with slow-changing traffic demand, we believe there is a chance to save energy while the network is not used as much. The tradeoffs between energy savings and network disruption due to sleeping will need to be weighed up against each other for specific networks and use cases. While we cannot quantify the exact amount of energy that is saved due to the lack of power models and information on the interfaces used in networks, we hope to bridge this gap in the future.

Link sleeping today is not quite fast enough to shut down links on a millisecond timescale, but we believe that during long periods of lower demands, *e.g.*, at night, the effects on the operation of the network could be minimal. Link utilization data presented in [7] shows that the traffic demand for backbone links follows a daily pattern with lower activity during the night. It also shows that there is a clear use case for link sleeping with the average link utilization never reaching more than 30%.

6.2 Feature Support

Instead of needing to write a script that allows a controller to turn on and off links remotely, we hope that in the future such a feature will be included in the router's software following some standard between different vendors. This would allow a network operator to have one central power plane controller that could interface with devices from different vendors without having to adapt the code for each device present in the network.

The other part that would help implement link sleeping is better support for network observability. While the OSPF TE metrics extension exists, it is not widely supported, as shown by the only experimental support in FRR and no support on our Cisco devices. Additionally, having wide support for even more fine-grained load information would allow the controller to make even better decisions on which links to shut down.

6.3 Transceiver Optimization

As we have shown the biggest bottleneck for waking up the network is the transceiver wake-up delay; our measurements in Fig. 5.15 show that decreasing this delay helps with avoiding congestion. There is a physical limit to the time it takes to turn on the interface from being completely off *e.g.*, it takes some time for the laser to stabilize in optical transceivers. But we hope that there might be a way to add an additional power state in the form of a low-power sleep mode that keeps certain elements powered on enough to enable fast wake-up times while not wasting too much energy. The lower the transceiver wake-up delay is, the more feasible link sleeping becomes.

6.4 Routing Loops

We are aware that routing loops could appear if we are turning on and off links but we did not see any in either the hardware implementation or in the network emulation. This does not mean that they do not exist but that our topology might not be susceptible to them or that they get resolved extremely fast. So while it did not manifest itself here, it is something to keep in mind for future experimentation.

Chapter 7

Summary

The goal of this work is to demystify link sleeping and guide future research toward the missing pieces needed for more sustainable systems. It offers some intuition about the inner workings and characteristics of a simple power plane controller putting links to sleep using today's hardware.

We show that some of the previous assumptions surrounding link sleeping do not hold for today's networks. Waking up interfaces within a few milliseconds is not possible today and in fact, is even the slowest step within a link sleeping system. Speeding up the interface wake-up time would enable big improvements for such systems. We hope to inspire further research into improving the interface wake-up time.

But even with today's interface wake-up times, we demonstrate that such a system is possible. The system's performance and effectiveness depend highly on the network and traffic demand. We outline the speed of traffic demand changes as one of the major factors that determine the effectiveness of link sleeping. We present an equation that can be used to determine if link sleeping can be used with little disruption to the network. Another important factor is the proportion of UDP vs TCP traffic, with TCP's congestion control mechanism helping with congestion that occurs due to reduced network capacity while sleeping. We show that the congestion control mechanism helps so that packet loss can be minimized during network wake-up while not affecting flow completion time in most cases.

We also expand our experiment to larger topologies and show that our simple power plane still works. We outline some optimizations to improve its performance in this scenario. Lastly, we also implement one of our setups in hardware and show that it behaves similarly to our network emulation.

Our insights can help advance research into practical link sleeping. This thesis presents only a first prototype of practical link sleeping. Possible improvements include more accurate sleep decisions and transceiver sleep modes for faster wake-up times.

Bibliography

[1] 2020. GN4-3N. https://network.geant.org/gn4-3n/.

[2] 2023. FRRouting. https://frrouting.org/.

[3] Aruna Prem Bianzino, Luca Chiaraviglio, Marco Mellia, and Jean-Louis Rougier. 2012. GRiDA: GReen Distributed Algorithm for Energy-Efficient IP Backbone Networks. *Computer Networks* 56, 14 (Sept. 2012), 3219–3232. https://doi.org/10.1016/j.comnet.2012.06.011

[4] Fahimeh Dabaghi, Zeinab Movahedi, and Rami Langar. 2017. A Survey on Green Routing Protocols Using Sleep-Scheduling in Wired Networks. *Journal of Network and Computer Applications* 77 (Jan. 2017), 106–122. https://doi.org/10.1016/j.jnca.2016.10.005

[5] M. D'Arienzo and S. P. Romano. 2016. GOSPF: An Energy Efficient Implementation of the OSPF Routing Protocol. *Journal of Network and Computer Applications* 75 (Nov. 2016), 110–127. https://doi.org/10.1016/j.jnca.2016.07.011

[6] Spencer Giacalone, David Ward, John Drake, Alia Atlas, and Stefano Previdi. 2015. *OSPF Traffic Engineering (TE) Metric Extensions.* Request for Comments RFC 7471. Internet Engineering Task Force. https://doi.org/10.17487/RFC7471

[7] Avinatan Hassidim, Danny Raz, Michal Segalov, and Ariel Shaqed. 2013. Network Utilization: The Flow View. In *2013 Proceedings IEEE INFOCOM*. IEEE, Turin, Italy, 1429–1437. https://doi.org/10.1109/INFCOM.2013.6566937

[8] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. 2010. ElasticTree: Saving Energy in Data Center Networks. (2010).

[9] Kin-Hon Ho and Chi-Chung Cheung. 2010. Green Distributed Routing Protocol for Sleep Coordination in Wired Core Networks. In *INC2010: 6th International Conference on Networked Computing*. 1–6.

[10] Thomas Holterbach, Tobias Bühler, Tino Rellstab, and Laurent Vanbever. 2020. An Open Platform to Teach How the Internet Practically Works. *ACM SIGCOMM Computer Communication Review* 50, 2 (May 2020), 45–52. https://doi.org/10.1145/3402413.3402420

[11] Jackie Lim. 2023. How Much Does It Burn? Profiling the Energy Model of a Tofino Switch. (May 2023), 33 p. https://doi.org/10.3929/ETHZ-B-000618002

[12] Sergiu Nedevschi, Lucian Popa, Gianluca Iannaccone, Sylvia Ratnasamy, and David Wetherall. 2008. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*.

[13] Daniel Otten, Max Ilsen, Markus Chimani, and Nils Aschenbruck. 2023. Green Traffic Engineering by Line Card Minimization. In *2023 IEEE 48th Conference on Local Computer Networks (LCN)*. IEEE Computer Society, 1–8. https://doi.org/10.1109/LCN58197.2023.10223344

[14] Meng Shen, Hongying Liu, Ke Xu, Ning Wang, and Yifeng Zhong. 2012. Routing On Demand: Toward the Energy-Aware Traffic Engineering with OSPF. In *NETWORKING 2012 (Lecture Notes in Computer Science)*, Robert Bestak, Lukas Kencl, Li Erran Li, Joerg Widmer, and Hao Yin (Eds.). Springer, Berlin, Heidelberg, 232–246. https://doi.org/10.1007/978-3-642-30045-5_18

[15] Sharada Yeluri. 2022. Longest Prefix Matching in Networking Chips. https://www.linkedin.com/pulse/longest-prefix-matching-networking-chips-sharada-yeluri/.

[16] Mingui Zhang, Cheng Yi, Bin Liu, and Beichuan Zhang. 2010. GreenTE: Power-aware Traffic Engineering. In *The 18th IEEE International Conference on Network Protocols*. 21–30. https://doi.org/10.1109/ICNP.2010.5762751

Appendix A

Line Card Sleeping

The benefit of being able to turn off entire line cards is that it would allow for even more energy savings than turning off individual links. While this was the project's initial focus, it was set aside as soon as we found out that the transceivers themselves already take multiple seconds to turn on. But we still gathered interesting insights on speeding up the wake-up time of line cards. We think it might be useful for future research that is not only concerned with shutting down single links but whole line cards.

We first give some background on modular routers and the architecture of line cards. We then discuss modifications to how line cards could be sent to sleep by storing their state before shutting down. This is because our measurements suggest that populating the line card data structures is a major contributing factor to the boot time of BGP routers handling nearly a million prefixes.

A.1 Line Card Background

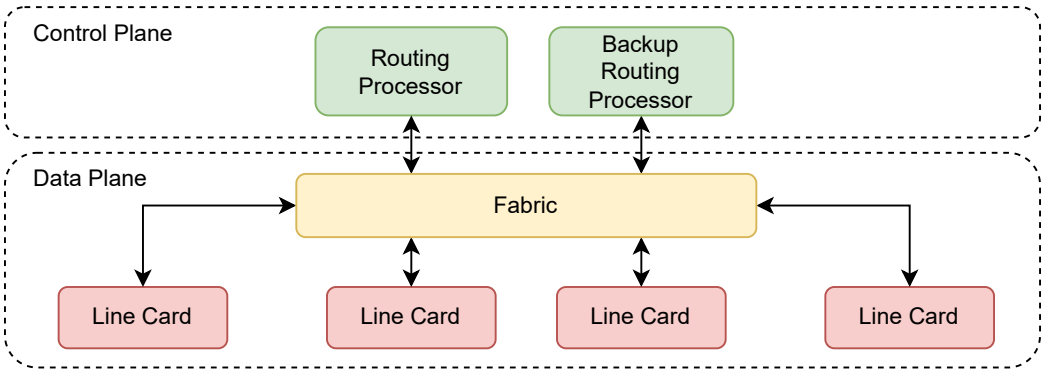


Figure A.1: High-level overview of a modular router.

A.1.1 Modular Routers

To be able to shut down individual line cards, we need to consider modular routers that contain multiple of them. Otherwise, if our router contains only one line card, we will disconnect the router from the network every time we turn the line card off. Modular routers can be divided into three parts (Fig. A.1). The routing processor takes care of all routing protocols and computes the

RIB (Routing Information Base), containing the information about all available routes and their ranking. The second part is the line card that does all the forwarding/routing with the help of the RIB that it gets from the route processor. It translates this into the FIB (Forwarding Information Base), which only contains the most essential information to forward the traffic. The third part is the fabric that connects all the line cards to one big system. Its job is to forward traffic from one line card to another when necessary. A modular router typically contains one routing processor plus backup, multiple line cards and one fabric to connect them all.

A.1.2 Line Card Architecture

Modern line cards are standalone devices that contain both a CPU and Memory in addition to the datapaths for the actual forwarding (Fig. A.2). The onboard CPU is used mainly for management tasks that control the functionality of the datapath. This means that the line card CPU interfaces with the route processor to receive information about how the forwarding is supposed to happen and then translates this into commands that control the behavior of the datapath itself. This is most likely done so that there can be a standardized interface between the route processor and the line card without the route processor knowing about the internals of the line cards. This could be a significant advantage, especially if we populate different types of line cards into the same system.

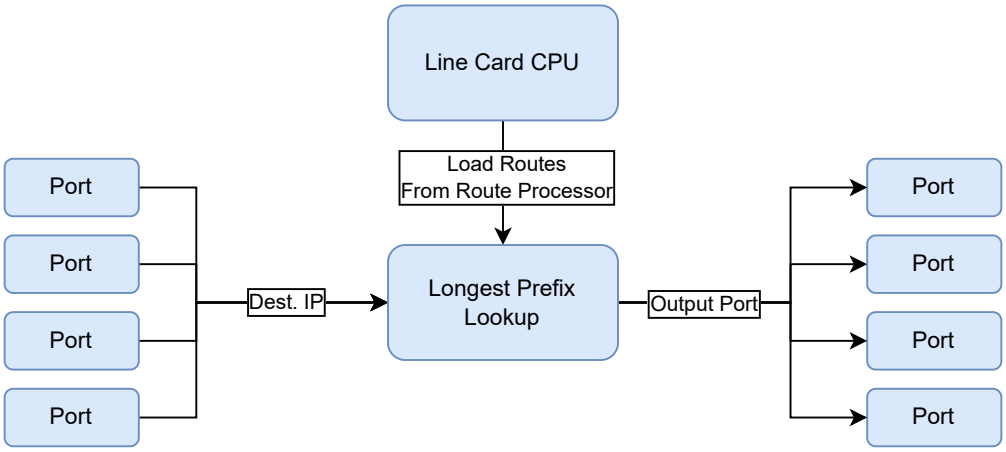


Figure A.2: Simplified view of a line card architecture.

A.1.3 FIB Datastructures

The FIB (Forwarding Information Base) is a data structure in the data plane that matches destination IPs to the port on which a packet should be forwarded.
I will present three major architectures that accomplish this goal:

- Trie
- TCAM
- Hybrid (Combination of TCAM and Trie)

Trie Trie data structures are search trees that are used to match bitstrings. They work by splitting up the bitstring and matching one part on each level until the longest matching entry is found. This is called longest prefix matching and is basically what a router does when looking up where to route an IP address.

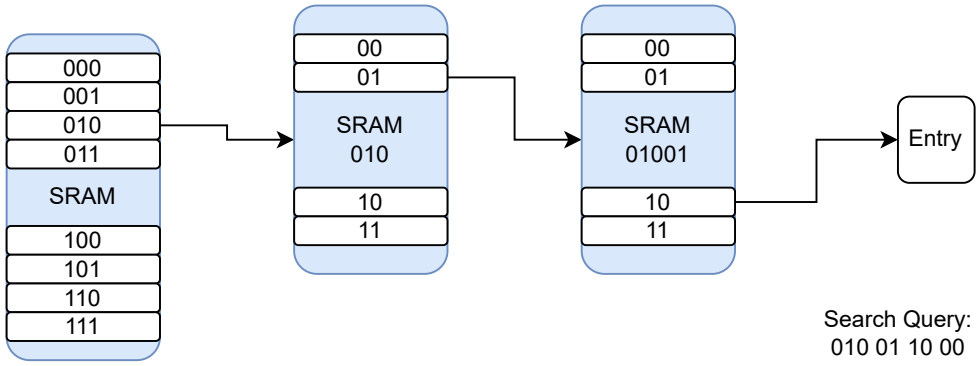


Figure A.3: Simple example of a search in a trie implemented with SRAM. Note that the trie nodes get assigned space in bigger physical SRAM modules and there is not one SRAM module for each trie node.

There are different ways to implement tries; it is possible to have one bit per trie level or have larger strides that match multiple bits per level. There is a trade-off between storage space and number of memory accesses per lookup. While larger strides use fewer levels and therefore need fewer memory lookups, they might also result in more memory usage since every node in the trie needs an entry for all possible combinations, which is 2^{stride}. A simple example of a trie lookup implemented with SRAM can be found in Fig. A.3.

TCAM While TCAMs have become more important, especially with the invention of programmable switches like the Tofino, they are not scalable enough yet to store the entire FIB of a backbone router without some trickery. TCAM memory does an LPM search of all its entries by parallelizing it in hardware where each entry gets matched simultaneously, and the longest prefix entry is returned. While this sounds perfect, it also has the drawback of using more chip area and power than SRAM. Because of that, they are mainly used for applications with fewer entries, like firewall rules.

Hybrid The hybrid approach combines tries and TCAM, where the first level of the trie is implemented with TCAM memory to improve the number of SRAM levels needed. In this case, the limited size of TCAM memories is not a problem since it only needs to be big enough to store all entries for the first few bits.

Today's architectures are even more complex, using additional tricks to cache often-used entries with the help of bloom filters. More information on this topic can be found here [15].

A.2 Today's bottleneck for line card boot times

We measure that routers today take minutes to boot after being completely switched off. While the router itself takes less than a minute to boot, installing all the routes on the line card takes minutes in our setup. We propose a way of sending a line card to sleep and waking it up within seconds instead of minutes.

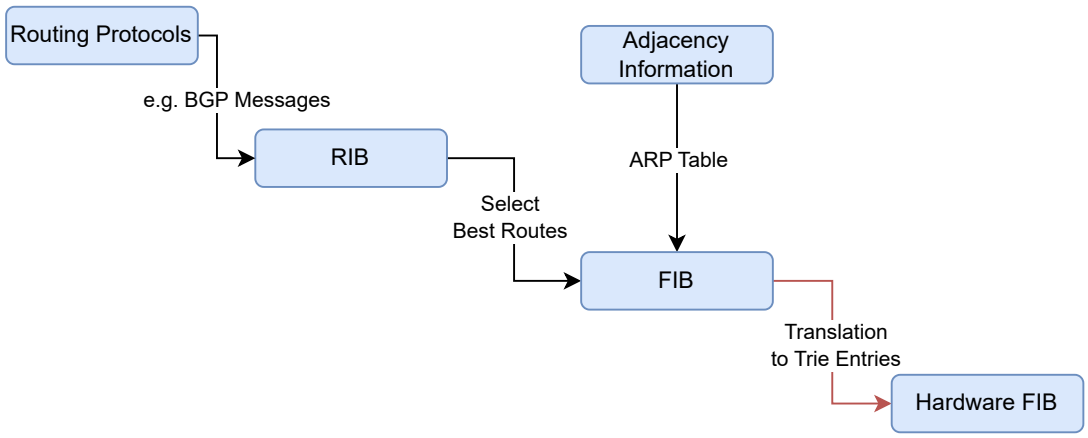


Figure A.4: Steps from the routing protocol to the final hardware FIB. The bottleneck is marked with a red arrow.

Search for the Bottleneck Measurements on our Cisco Nexus 9300 router showed that most of the components of this device could be booted within less than a minute. But it takes the line card around 7 minutes to install roughly 900k IPv4 prefixes into its FIB database. A simplified version of the steps from receiving the routing information to the final hardware FIB is shown in Fig. A.4.

One part that often gets blamed for this is the BGP messages that need to be exchanged when a router comes back online. But we can show that, at least for our device, the router received all 900k routes from his BGP neighbor after a few seconds.

The next step is transferring the routing information from the route processor to the line card. Since our device is not a modular router with multiple line cards, the route processor and line card are on the same module. We can thereby also rule out this step as the bottleneck.

```
Total number of IPv4 host trie routes used : 0
Total number of IPv4 host tcam routes used : 3
Total number of IPv4 LPM trie routes used : 929299
Total number of IPv4 LPM tcam routes used : 5132
Total number of IPv6 host trie routes used : 0
Total number of IPv6 host tcam routes used: 0
Total number of IPv6 LPM trie routes used : 0
Total number of IPv6 LPM tcam routes used : 7
```

Figure A.5: Output of our Cisco Nexus 9300 router showing that the FIB is implemented, at least for the most part, as a trie.

We confirm with the command output shown in Fig. A.5 that the FIB in our router is implemented as a trie. Tries are implemented with SRAM/DRAM, which is basically as fast for reading

URIB ROUTES:	user nodes	total nodes	elem size bytes
default	934453	1706115	24
management	6	8	24

Figure A.6: The number of trie nodes is around 1.7 million with a node size of 24 bytes. This means the total state that needs to be saved is around 40MB.

as it is for writing. So, there is no reason why writing to the FIB database with high bandwidth would not be possible.

This leads us to believe that the actual bottleneck in the start procedure is the computation the line card has to do to get from the forwarding information to the exact FIB format that can be pushed into the memory. This translation is necessary because the entries are not just directly saved one after the other in memory. The memory needs to be populated in a way that reflects the trie that we are using to search through the entries.

Looking through the log files of our device, we can find hints of this. We can see that the routes get pushed into the FIB in bulk updates of about 4 thousand entries, taking roughly 1.8 seconds per bulk update. Calculating the time it would take to install 900k entries, we get a time of about 7 minutes.

Proposal While we need to translate the RIB information into the FIB format if we boot up a line card, we propose a different approach when sending a line card to sleep. Since a running line card has already computed the correct format of the FIB, we can save this format into non-volatile memory. If we do this before sending a line card to sleep, we can reload the previous state from the non-volatile memory when we wake up the line card again. This allows us to skip the translation step from the RIB to the hardware-specific FIB. After the previous state is loaded into the hardware FIB, we only need to apply the changes that have happened since the line card went to sleep. The number of updates necessary will highly depend on how long the line card was asleep and how likely it is for entries to change.

Wake-up Time Improvements While we cannot give a concrete answer on how much time this proposal would save in practice, we hope it could bring the wake-up time below one minute. This is because loading even a few hundred MB of data from non-volatile memory does not take longer than a few seconds. As described in Fig. A.6, the total FIB state we need to save in our example is around 40 MB. This is why we expect the load time of the old state to be only a few seconds instead of the 7 minutes even if we have to update a few entries that changed in the meantime.