ETH zürich

Networked Systems
ETH Zürich — seit 2015

Towards a better understanding of FIB architectures

Semester Thesis

Author: Tim Distel

Tutor: Lukas Röllin

Supervisor: Prof. Dr. Laurent Vanbever

February 2025 to June 2025

Abstract

Modern routers need to operate at extremely high speeds while maintaining increasingly large Forwarding Information Bases (FIBs). However, most routers are black boxes and it is difficult to obtain meaningful insights from them and detect potential bottlenecks. This thesis presents the *Fibulator*, a modular and extensible simulation environment designed to test and analyze hardware forwarding implementations, focusing on Longest Prefix Matching (LPM) algorithms. By replicating key aspects of hardware forwarding, the Fibulator aims to bridge the gap between the development of novel algorithmic ideas and practical hardware implementation. As a proof of concept, a variable-stride trie algorithm is implemented and evaluated.

Contents

Chapter 1

Introduction

1.1 Motivation

Routers are a cornerstone of today's Internet and they become increasingly faster while simultaneously having to maintain ever-growing forwarding tables. Modern routers and switches can achieve throughputs on the order of terabits per second (Tbps), which corresponds to multiple packets being forwarded per nanosecond. To achieve such high throughput, the data plane must rely on specialized hardware components. However, due to the proprietary nature of many routers, only little is known about their internal hardware architectures. It can be hard to obtain meaningful insights and detect potential bottlenecks from such black boxes.

A router maintains a Forwarding Information Base (FIB), which can be described as a large lookup table that maps different destination IP prefixes to corresponding egress ports of a router. The FIB should be structured in such a way that it allows for high-speed forwarding of packets, while also enabling fast updates of entries. The Border Gateway Protocol (BGP) relies on preferring longer, more specific, prefixes to shorter ones. This leads to one of the main algorithmic challenges in designing a forwarding implementation, called Longest Prefix Matching (LPM). To efficiently perform LPM, most algorithms are implemented as a hardware pipeline, based on the logical model of a trie [9]. However, this model still leaves much room to fine-tune and optimize a hardware LPM implementation.

The growing sizes of FIBs are a problem because the capacity of memory components that allow the required forwarding speeds is very limited. Hence, it is important to find potential optimizations of existing solutions. There exist many approaches to reduce the size of a FIB. For example, ORTC [2] tries to compress a FIB by reordering and merging entries with the same next-hop. An other example is DRAGON [4], which aims to minimize FIBs on an even higher level by simplifying the global Routing Information Base (RIB). PopTrie [1] is an example of an algorithm that aims to speed up in-software implementations of forwarding. The focus of this thesis lies on the implementation of LPM in hardware using a pipeline design. It is further assumed that the FIB itself stays more or less fixed. The goal is to find ways to efficiently represent pre-existing FIBs in hardware.

At the time of writing this thesis, it is difficult to try out and assess different data plane hardware implementations. Actually implementing a design in hardware is both time-consuming and costly. This is why in this thesis, a simulation environment called the Fibulator is presented that focuses on analyzing different in-hardware forwarding implementations. This allows for a time-effective analysis of new implementation ideas and might catch design weaknesses early on. The Fibulator can also be used to gain a better understanding of existing hardware implementations.

1.2 Task and Goals

The main task of this thesis is the development of the Fibulator, a modular and extensible environment to test and analyze different LPM hardware implementations. Due to the high cost of prototyping in-hardware implementations, this simulation environment should enable a more accessible way of quickly testing different architectural ideas.

As a proof of concept, this thesis includes the implementation and evaluation of a variable-stride trie algorithm. Even for this rather simple model, valuable insights regarding memory usage and access patterns can be gained. This should demonstrate the utility of the simulation environment provided by the Fibulator.

1.3 Overview

Chapter 2 covers the necessary background for this thesis. This includes the basics of longest prefix matching as well as common approaches for in-hardware LPM implementations.

In Chapter 3, an overview of the Fibulator is given. It highlights the most important architectural decisions and key components. Following this, Chapter 4 goes into more detail regarding the actual implementation of the Fibulator.

Chapter 5 demonstrates the capabilities of the Fibulator through the example of a variable-stride trie implementation. It also presents performance metrics of the Fibulator.

Finally, Chapter 6 provides a short summary of the thesis as well as an outlook on potential future work.

Chapter 2

Background

This chapter provides the necessary background for this thesis. This includes the basics of Internet routing and the resulting challenge of longest prefix matching. It also presents common approaches to implement LPM in hardware.

2.1 Internet Routing

The Internet uses Internet Protocol (IP) addresses to route packets to a chosen destination. Those IP addresses can be grouped into different prefixes, also called networks. The so called prefix length determines, how many of the first bits, starting from the most significant bit, make up the respective prefix. Addresses that match those first bits are said to belong to the same network or prefix. This thesis uses the following octet notation for a 32-bit IPv4 prefix: 123.45.67.0/24 where the integer after the "/" indicates the prefix length. The example above maps to a network with $2^{(32-24)} - 2$ host addresses, ignoring the first address (network address: 123.45.67.0) and the last address (broadcast address: 123.45.67.255).

Using prefixes can simplify the definition of a desired routing behavior. This stems from the fact that two destination addresses that need to be forwarded to the same egress can be grouped to one prefix and only require a single entry. The same applies for consecutive prefixes with the same egress that can be merged into a larger prefix. The Internet relies on the premise that longer, more specific, prefixes are preferred over shorter ones. This leads to an algorithmic challenge that is quite unique to networking: **Longest Prefix Matching**.

Routers keep a lookup table of many forwarding entries, also called the FIB. A forwarding entry can be described as a pair of a prefix and a corresponding egress port of a router. Usually, routers use the abstraction of two different planes: the data plane (also known as forwarding plane) and the control plane. The task of the data plane is to forward packets as fast as possible. In order to achieve this, it stores the FIB in an optimized structure to allow efficient in-hardware forwarding. On the other hand, the control plane is responsible for the RIB. The RIB represents higher level routing decisions that do not need to be as fast as the data plane. This includes, for example, BGP updates and modifying the entries in the data plane accordingly.

Nowadays, routers have FIBs with more than one million entries, spanning many different prefix lengths. Especially as routers are required to forward packets at even faster speeds and FIBs continue to grow larger, efficient LPM implementations become increasingly important. The goal of an LPM algorithm is to efficiently find the longest prefix, i.e., the most specific network to which the destination address of an incoming packet belongs to.

2.2 LPM Implementations

Modern routers are capable of a throughput of multiple Tbps while maintaining FIBs with over one million prefixes. For example, a new switch from FS [3] supports a forwarding rate of 22'200 Mpps. This corresponds to performing more than 22 lookups per nanosecond! To achieve such a performance, in-software approaches are infeasible. Instead, specialized forwarding hardware is used.

2.2.1 Tries

A common strategy for efficiently storing FIBs are tries. The simplest implementation would be a binary trie, comparing each individual bit of an address until a leaf node, which corresponds to an egress, is reached. One disadvantage of this naive approach is that in the worst case it would take 32 memory accesses to forward an IPv4 packet since each of the individual bits could have to be checked. The number of sequential memory accesses needed for a lookup give a very good estimate on the latency of an LPM implementation [6, Chapter 11].

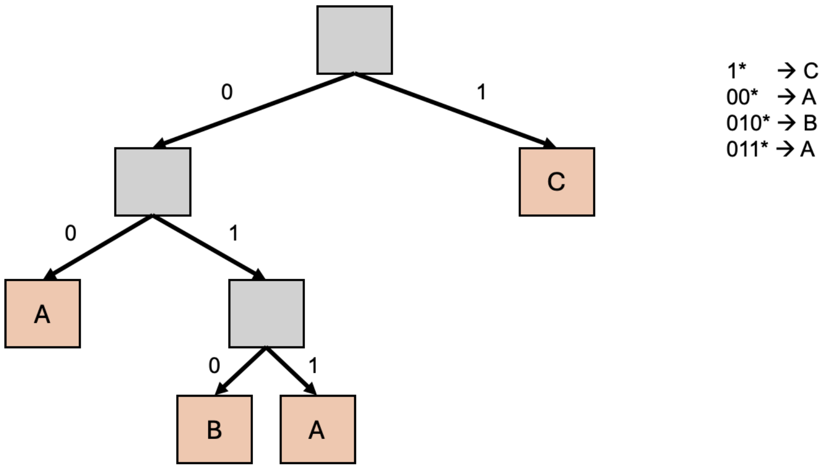

Figure 2.1: An example of a binary trie with the corresponding forwarding behavior.

One approach to reduce the number of levels in a trie, and hence the worst-case number of memory accesses needed for a lookup, is to group the bits into segments, also called strides. A possible implementation could be to use a constant stride of 4, meaning that an IPv4 address containing 32 bits is split into 8 bit segments of length 4. Other than in a binary trie, this N-ary trie now has $2^4 = 16$ children (each combination of the 4-bit strides) per node. This reduces the depth of the trie to 8. In general, using longer strides reduces the latency, which corresponds to the worst-case number of memory accesses, but also increases the memory needed to store a given FIB. There are two main reasons for the increase in memory usage:

1. Not all prefixes *fit* into a stride configuration. For example, 128.0.0.0/7 does not fit into a constant-stride 4 trie. To deal with this problem, the prefix needs to be extended, i.e., split

into two equivalent longer prefixes, until it fits into the strides. In this example 128.0.0.0/7 needs to be extended to 128.0.0.0/8 and 129.0.0.0/8 which then fits into the strides. Obviously, this can lead to more entries, especially if the strides are not well-aligned with the majority of the FIB entries.

2. For the algorithm to work correctly, the trie requires to be normalized, i.e., each node must have either zero or 2^N children, where N is the size of a stride. This means that even if many children point to the same egress, all the memory cells need to be populated. Hence, the normalization cost grows exponentially with the length of the stride.

A generalization of the constant-stride trie is the variable-stride trie. This type of trie allows for strides to have different lengths. An example for an IPv4 trie would be a stride configuration of $[8, 8, 4, 4, 8]$. The individual strides need to sum up to a total of 32 bits.

There are many different strategies to further reduce the latency and memory footprint of LPM implementations. For further details, the reader is referred to [6, Chapter 11].

2.2.2 Hardware Implementation

In hardware, a trie can be implemented as a pipeline. Each stage of the pipeline represents one level of the trie. Each stage contains some processing logic and one (or multiple) memory components where the entries of the nodes are stored. The two most relevant memory components in practice are Static Random-Access Memory (SRAM) modules and Ternary Content-Adressable Memory (TCAM) modules.

TCAM

TCAMs belong to the family of Content-Addressable Memories. This means that instead of retrieving an entry by its address, an entry can be accessed by its content. A TCAM module can search all of its entries in parallel and is able to retrieve matches in only one cycle. *Ternary* refers to the fact that a TCAM supports three possible states per bits, also allowing *don't-care* bits. This is usually implemented as a second equally sized entry that serves as a bit-mask. Therefore, prefixes can be naturally stored as entries, where all the bits that belong to the host address-space of a network can be set to don't care. This makes a TCAM an ideal candidate to be used for hardware LPM implementations, as it allows LPM lookups to be performed in only one cycle. However, simply using one big TCAM to store an entire FIB comes with two major downsides:

1. **Power Consumption and Area Overhead:** Storing one bit in a TCAM requires around 16 transistors, whereas SRAM typically only requires 6 transistors per cell. This leads to higher power consumption and worse storage density. Additionally, for the parallel search, each cell needs to be activated, leading to an even higher power consumption compared to an SRAM module.

2. **Slow Updates:** Depending on the implementation, a TCAM module typically returns the result for the first match of a lookup. This implies that for a correct LPM lookup, prefixes must be stored from longest to shortest prefix length in a TCAM. Hence, inserting a single entry has a cost of $O(n)$ since it might require shifting many entries.

SRAM

An implementation purely based on SRAM is also possible. This can be achieved by building a trie structure in hardware in the form of a pipeline consisting of one stage per level.

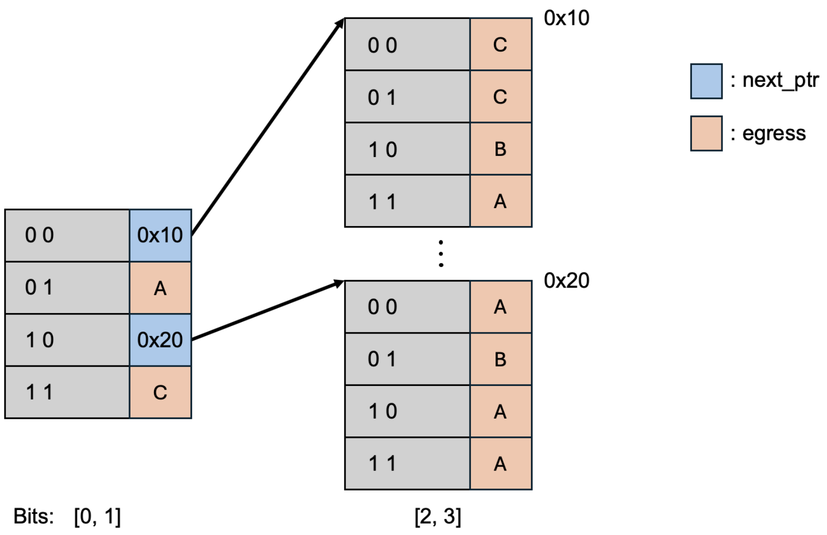

Figure 2.2: Simplified hardware implementation of a trie with constant stride of 2.

Figure 2.2 shows a simplified example of an in-hardware LPM lookup for 4-bit addresses. The pipeline has two stages with a constant stride of 2 which means that each stage checks 2 bits of the destination address. An entry of the SRAM module can be either an egress or a pointer to a memory block in the next stage. For example, a packet with address $0b0101$ gets its egress already set to A in the first stage by accessing the cell that corresponds to its first two bits. On the other hand, a packet with address $0b0010$ needs to traverse to the second stage with base address $0x10$ where it then gets its egress set to B. In the second stage, bits 2 and 3 are used as an offset to obtain the actual memory address. The actual address accessed in the second stage would be $0x12$ which is omitted in the graphic for simplicity. It is also assumed that each memory entry, holding either an egress or a pointer to the next stage, has a consecutive address assigned.

This relatively straight-forward implementation requires the underlying trie to be normalized, meaning that a node has either zero children, i.e., is an egress or has all 2^N children, where N is the size of the stride. This comes from the fact that all possible combinations of bits in a given stride must map to a valid SRAM address.

Hybrid approaches, combining both TCAM and SRAM modules, are popular in practice. This can be achieved by, for example, implementing a basic trie structure using SRAM modules and beneficially adding TCAM modules to certain stages.

Chapter 3

Design

This chapter provides an overview of the intended simulation flow of the Fibulator. It highlights key design decisions and the fundamental building blocks.

3.1 Overview

The Fibulator provides a modular and extensible environment to simulate and test different FIB hardware algorithms. The core logic of the Fibulator is split into two parts: the control plane and the data plane. This design choice was made because real-world implementations generally follow this abstraction as well.

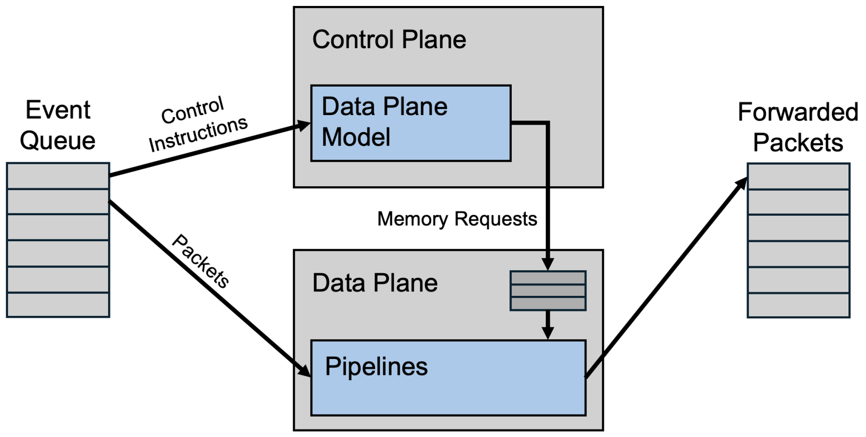

Figure 3.1: Simplified program flow of the Fibulator.

Figure 3.1 shows a simplified execution flow of a simulation using the Fibulator. The event queue serves as a predefined input to the Fibulator. There are two types of events: control instructions, which are processed by the control plane and packets, which are forwarded by the data plane through its pipelines.

The Fibulator, as the name suggests, focuses on the FIB and the implementation of a correct forwarding algorithm. That is why it was decided that control instructions are considered to be fairly *low-level* instructions concerning the FIB, as, for example, inserting, removing or updating a single FIB entry. The underlying RIB decisions are outsourced into the setup of the simulation by preparing an intended sequence of control instructions in the event queue to emulate certain *higher-level* behavior.

3.2 Architecture

3.2.1 Data Plane

The basic model chosen for the hardware implementation in the data plane is that of a pipeline. It is assumed that most in-hardware FIB implementations can be represented as a pipeline architecture, leading to no heavy restrictions on implementing different approaches. This model leads to a natural time representation in the Fibulator: one cycle, which is equal to the simulated latency of a single stage. Making the simulation event-based, i.e., processing one packet at a time was considered as well, but the cycle-based approach using a pipeline model was preferred due to its clear structure and time representation. It is further assumed that each stage contains some in-hardware logic (which will also be referred to as the stage function) and one or multiple memory components. These could, for example, be TCAM or SRAM modules. The latency for one stage and hence the duration of a single cycle is estimated to be the read-latency of the corresponding memory components.

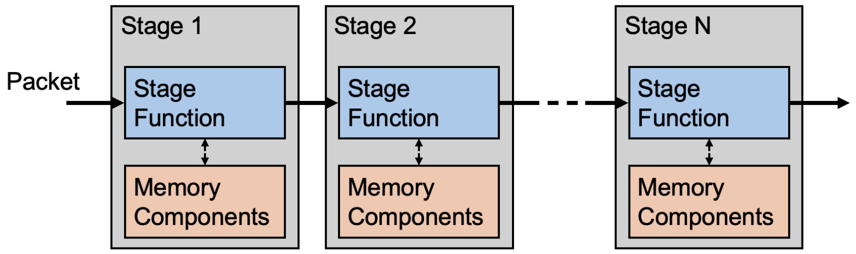

Figure 3.2: Pipeline representation in the Fibulator.

To test a proposed algorithm, the user is supposed to provide his own implementation of the data plane. This includes three main design decisions:

1. **Pipeline Configuration:** This includes defining, how many and what pipelines (e.g. IPv4 and IPv6) are used and how many stages the individual pipelines contain. Additionally, memory components with pre-defined properties need to be assigned to the corresponding stages.

2. **Stage Behavior:** This aspect characterizes the stage function. It includes, for example, the specification of the stride of a stage and how a stage interacts with its memory components.

3. **Data Plane On-Cycle Behavior:** This embodies all the decisions the data plane makes on a cycle-basis. It is necessary to define how incoming packets are buffered, through which pipeline they are forwarded, and how memory requests from the control plane are processed and timed.

Memory Components

In addition to the user-defined implementation of the data plane itself, specific memory components can also be implemented. Those memory components can then be used in the different stages of a pipeline. The current version of the Fibulator already provides an implementation for TCAM and RAM modules.

3.2.2 Control Plane

In contrast to the data plane, the control plane does not need to be as fast in practice. Generally, the control plane logic is executed on a normal processor. As described above, the control plane is supposed to handle the incoming control instructions, such as, for example, inserting a new FIB entry. To achieve this, the control plane maintains a simplified model of the data plane. From this model and a control instruction, it produces memory requests which are then sent to the data plane. A memory request contains the exact memory component, the address, and the new value of the entry that needs to be updated. This mechanism involving memory requests is used to provide a clear separation of the data plane and the control plane.

Similarly as in the data plane, the user has to provide his own implementation of a control plane in order to test a proposed algorithm. This implementation needs to cover three main aspects:

1. **FIB Representation:** This defines the logical model of how the FIB is stored in the control plane. A possible representation is a variable-stride trie, as discussed in Section 2.2.
2. **Generate Memory Requests:** This aspect defines a mapping from the logical FIB model to actual memory entries that need to be written to the data plane pipelines.
3. **Control Plane On-Cycle Behavior:** For simplicity, both the data plane and the control plane are executed once per simulated cycle. One way to handle control plane latencies is to outsource them into the setup of the event queue. Events can occur with a delay and are then processed instantaneously by the control plane.

3.3 Data Collection

The current version of the Fibulator provides two approaches to collect data in a simulation. The first method involves the forwarded packets. Relevant metrics can be attached to a packet while it traverses the pipeline, and can later be analyzed. The second approach focuses on memory-related statistics. The user can save the current state of a memory component to a file. This can contain metrics such as the number of bits currently used or the ratio of different entry types a component holds.

Both approaches offer valuable insights but target different aspects of the simulation. They can be used independently or in combination, depending on the goals of the analysis.

Chapter 4

Implementation

This chapter discusses the implementation details of the Fibulator, including its structure and how the different components are implemented and interact with each other. It also highlights the reasons behind some key implementation decisions.

4.1 Code Organization

In order to be able to efficiently handle large FIBs and long packet traces, the core simulation of the Fibulator is written in C++. The project is compiled with C++ 17. However, to simplify the overall simulation process, a Python wrapper has been implemented which automatizes the configuration and build process of the Fibulator. The analysis and visualization of the simulation output is handled in Python as well. This aims to make the configuration and evaluation workflow more user-friendly. The project is organized into four main directories:

1. `inputs`: This directory contains the input files required for a simulation. Typically, it contains a FIB and a packet trace.
2. `outputs`: The results generated by the Fibulator are stored in this directory. This includes the forwarded packets as well as the memory statistics.
3. `python`: This folder holds the Python utilities which can be used for the setup of a simulation and the evaluation of its output.
4. `src`: This directory contains the C++ source code of the Fibulator. The main focus of this chapter lies on this implementation.

4.1.1 Fibulator

The Fibulator itself, meaning the C++ implementation, is divided into 6 subdirectories: `algorithm`, `base`, `control_plane`, `data_plane`, `memory` and `util`. The `algorithm` directory contains the code for the algorithm that is to be implemented by the user. This aims to strictly separate the user implementation from the base structure of the Fibulator. Multiple components of the other directories will be explained in more detail in the following sections.

4.2 Event Data Structure

This section describes the most important data structure in the Fibulator: events. Events are implemented as a hierarchical structure and include packets and, for example, the insertion of a FIB entry. The main idea is that pointers to the event objects are passed instead of passing the events by value. This has two main advantages: Firstly, moving pointers is cheap, i.e., there is no need to copy potentially big events, and secondly, the usage of pointers allows for polymorphism.

All events in the Fibulator are derived from the IEvent abstract class. It follows the "I" prefix notation, which is further described in section 4.4. The most important attribute of an event is `time_in`, which defines when it gets processed in the simulation. In addition, it contains an `EventType` which determines whether it is a control instruction or a packet.

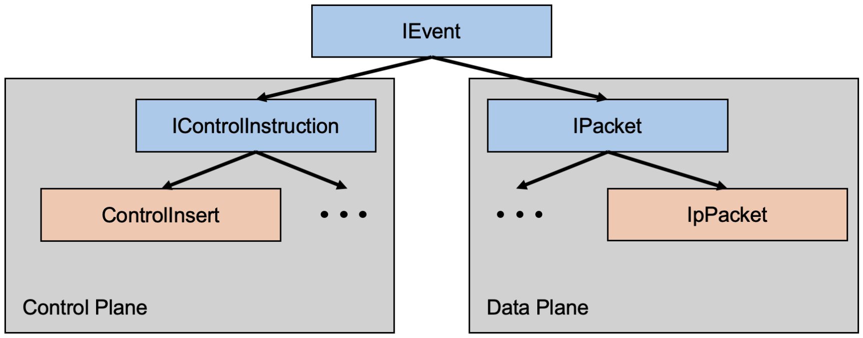

Figure 4.1: Hierarchy of the event data structure.

Figure 4.1 shows the inheritance model of the different types of events used in the Fibulator. The main advantage of using polymorphism is that it simplifies the handling of different types of events and allows for more modularity and extendability. As a result, the simulation only needs to hold one event queue that contains all future events. As soon as an event occurs, it gets casted to a more specific type and pushed to the corresponding queue: `IEvents` of derived type `IPacket` get pushed into the packet queue of the data plane and events of derived type `IControlInstruction` get pushed to the control plane queue. This naturally leads to a hierarchy, such that each component receives an object that is just as specialized as it needs to be in order to be processed. It also enables the user to derive new types from the `IControlInstruction` and `IPacket` classes. In the current version of the Fibulator, only IP packets (`IpPacket` class) and FIB entry insertions (`ControlInsert` class) are used.

However, handling events as pointers introduces a new challenge: determining who owns the object being pointed to. This information is relevant when we do not want to allow, for example, dangling pointers and other potentially dangerous ambiguities. The next section describes how correct ownership is ensured in the Fibulator project.

4.3 Ownership Model

A clear and intuitive ownership model was chosen, utilizing smart pointers in a modern C++ fashion. More specifically, `std::unique_ptr` are used to pass events from one component to another. In this way, it is ensured that the ownership is always safely passed to the component that currently handles an event. Apart from more robust code, this implementation also allows for more flexibility. In particular, a component can safely delete an object it owns and free up unnecessary memory. This can, for example, be useful if stages are allowed to drop packets.

This ownership model is not limited to the events that propagate through the Fibulator. It is also applied to all components containing, or more precisely owning, other components. For example, each stage owns its memory components. The most important components are further described in the next section.

4.4 Key Components

In order to achieve modularity, the Fibulator is divided into different components. Most components are structured as a hierarchy of classes with an abstract base class as the root, because this allows for an easy way to implement a new component that fits seamlessly into the Fibulator. For brevity and to emphasize that those classes cannot be instantiated, the code base uses the prefix "I" to mark those abstract base classes. However, to avoid confusion, it is important to note that those abstract base classes are technically not interfaces, since they do contain member variables and non-virtual functions.

4.4.1 Simulation

The `Simulation` class is the foundation of the Fibulator. This class contains both the control plane and the data plane. It keeps track of the simulation time and calls the cycle function of its members during each cycle. In addition, it handles the processing of the event queue. Each event that has a `time_in` smaller or equal to the simulation time is casted to a more specific type and added to either the data plane or the control plane queue.

4.4.2 Planes

Both the control plane and the data plane are implemented as an abstract base class, namely `IControlPlane` and `IDataPlane`. Those base classes only provide the minimal functionalities. This should ensure that the user-provided algorithm can be implemented as freely as possible while still providing some basic structure.

Control Plane

The abstract base class `IControlPlane` holds a `std::queue` of `IControlInstruction`s. Additionally, it contains a `DataPlane_Model` instance. This data plane model provides a simplified view of the memory components of the data plane and can be used to keep track of the current state in the pipelines. This additional structure is used to maintain a clear separation of the two planes and does not require the control plane to access the memory components in the data plane.

Data Plane

The `IDataPlane` base class contains a `std::queue` for both incoming packets and memory requests. In addition, it holds the hardware components, which are structured as a hierarchy of multiple pipelines containing multiple stages:

1. **Pipelines:** Each instance of the `Pipeline` class contains a `std::vector` of stages. The pipeline cycles each individual stage from last to first. The stages are called from last to first because each stage writes its output to the input of the next stage at the end of a cycle. In every cycle, a pipeline returns the packet that is outputted from its last stage.
2. **Stages:** Each instance of the `Stage` class contains a `std::vector` of `IMemoryComponents` and a `std::function<void(Stage&)>` which serves as the stage function. It was decided to use the `std::function` wrapper that uses a reference to its stage as its only parameter, because it allows a stage function implementation to be reused across different stages while also providing maximum flexibility.

At the end of each cycle, the data plane returns all forwarded packets, i.e., all packets that were returned from its pipelines.

4.4.3 Memory Components

Following the same methodology as used for the control plane and the data plane, memory components are derived from an `IMemoryComponent` base class. Among others attributes, memory components contains two integers defining the number of bits of a key, i.e., address and the number of bits of an entry. The size of an entry was chosen to be variable in order to potentially simplify accessing the data and making the implementation of an algorithm clearer. `IMemoryComponent` also implements `IMemoryLogger` which is further described in Section 4.6.3.

The Fibulator already contains two implementations of memory components: TCAM and RAM. However, if needed, additional memory components can be easily added by deriving a new specialization from the `IMemoryComponent` base class.

RAM

The content of a RAM module is stored as an `std::unordered_map` from an address, stored as a sequence of bits using the `Bits` class (further described in section 4.6.1) to a `RAMCell`. Each RAM cell contains a value that is also stored as a `Bits` instance. The decision of using a `std::unordered_map` was made because RAM modules can be sparsely populated, which would lead to a lot of wasted memory in the simulator if all cells were stored in a `std::vector`. Only storing used, i.e., valid cells in a `std::vector` would lead to an unnecessary search complexity for a given address. Using `std::unordered_map` seems to be a good compromise regarding insertion and lookup complexity as well as storage overhead. The current implementation supports reading and writing cells to a given address.

TCAM

The TCAM module stores its cells as a `std::vector` of `TCAMCells`. A `std::vector` was chosen since TCAMs are considered to be smaller and more densely populated than RAM modules. Each `TCAMCell` contains a key and a key mask, stored as `Bits` instances, and a value. The TCAM implementation supports lookup, read and write functionality.

It is intended, that first a lookup for a certain key is made, which returns the indices of all cells that match the content. This lookup, which normally happens in parallel for a hardware implementation, is implemented as a simple linear search of all entries in the TCAM. The hash maps used for a `std::unordered_map` do not support don't-care bits that are required for TCAMs. This necessity of a linear search is another reason why `std::vector` was chosen as the container. There might be more efficient approaches to perform TCAM-like lookups in software but this was not considered to be a relevant bottleneck in the Fibulator. After the lookup, the value of the desired cell can be read with the obtained index.

4.5 Variable-Stride Implementation

To provide a guiding example of how an algorithm can be implemented in the Fibulator environment, this section describes an implementation using a variable-stride trie. The implementation is split into three main parts: the configuration, the data plane and the control plane.

4.5.1 Configuration

All parameters related to the definition of the data plane are kept in one `algo_config.h` file to provide a clear separation of all configuration parameters. This includes, for example, the number of egress ports of the simulated router and the different stage configurations. A `StageConfiguration` contains the stride, i.e., the number of bits a stage handles and the type of its memory components. This implementation only uses one SRAM module per stage. The SRAM modules are further defined by their number of entries as well as the length of an address and the number of bits stored in one entry.

Additionally, it specifies the file paths for reading the input data (packets and FIB entries) and writing the output data (forwarded packets and memory logs). Currently, the Python wrapper is set up in such a way that a C++ configuration file is automatically generated from the parameters set in Python.

In a separate file, the memory layout of an SRAM cell is defined. This serves as an additional abstraction layer to make accessing the different entries of the SRAM modules clearer. This memory layout defines which bits of an SRAM entry map to what variable. As described in section 2.2.2, each entry can be either an egress or a pointer to an block in the next stage. A cell has a big enough capacity that it can contain either values. We need an extra variable `is_egress` in the form of a single bit that determines whether a given cell is an egress or a pointer entry.

4.5.2 Control Plane

In each cycle, the control plane collects all events in its control instruction queue. In this implementation, only FIB insertions are used. As soon as a certain threshold of pending insertions is reached, `update_fib` is called, which can be split into three main tasks:

1. **Update Trie:** The control plane holds a list of all FIB entries in the form of a `std::set`, sorted by prefix length. First, the set of FIB entries is updated according to all pending changes. Then, the current variable-stride trie is cleared and rebuilt from the new state of the FIB set.

2. **Map Trie to Memory Requests:** After constructing this initial version of the trie, it is normalized. This means that every node of the trie must have either zero or 2^N children, where N is the stride of the respective level. This normalized trie is then used to obtain a sequence of memory requests that is equivalent to the trie. Those requests are obtained by

processing the trie in a breadth-first manner. Each level maps to a set of memory requests for its corresponding stage in the data plane.

3. **Send Memory Request to Data Plane:** Finally, the memory requests are compared to the simplified data plane memory representation stored in the control plane. This process filters out unnecessary memory requests, meaning SRAM entries in the data plane that already hold the correct values. The remaining memory requests are then sent to the data plane and the internal data plane model is updated.

This implementation aims to be as simple as possible while still providing a clear structure that can be used as a template for other algorithms. A possible improvement would be the implementation of a variable-stride trie that is capable of handling incremental updates. The current implementation allows for incremental inserts, but upon deleting a single entry, the whole trie needs to be rebuilt to ensure correctness. Another improvement could involve ensuring that the number of issued memory requests is minimized by incorporating the current state of the data plane into the memory request generation and not just checking for potentially redundant requests.

4.5.3 Data Plane

Pipeline Configuration

To handle the setup process of a data plane according to its configuration, it is recommended to use a factory design pattern. The `create_dataplane` factory function creates an instance of the specialized data plane implementation and returns it as a pointer to a `IDataPlane` base class. Through this base pointer the ownership of the data plane can then be transferred to the `Simulation` class. The IDataPlane base pointer exposes all functionalities required by the `Simulation` class. All the logic related to building a correct data plane from a given configuration is therefore confined to this function. This includes creating instances of the different memory components, adding them to the correct stages, and adding those stages to the correct pipelines.

Stage Behavior

The stage function is implemented as described in Section 2.2.2. The stage function has a reference to its own stage as its only parameter. This allows the function to change its behavior according to the properties of its stage, which makes it possible to reuse one general stage function for multiple stages. The `next_ptr` is handled the following way in this hardware implementation: each packet contains a metadata structure, which contains a field `next_mem_base`. So instead of *following* the pointer value, a stage writes the base address of the next memory block into this field and then forwards the packet including the metadata to the next stage.

When a packet arrives in a stage, at first, the `next_mem_pointer` is extracted from the metadata. This address, combined with the offset given by the current stride, results in the relevant SRAM address that needs to be accessed by a stage. This memory address is then read and handled according to two cases:

1. **Egress:** If the memory entry is an egress entry, the egress port of the packet gets set in its metadata. All subsequent stages ignore this packet that has its egress already set and simply forward the packet to the next stage without further modifying it.

2. **next_ptr:** The stage writes the value of the pointer to the metadata of the packet and forwards it to the next stage.

Apart from the metadata, a packet also contains a statistics structure. This structure can be used to track relevant properties of the forwarding behavior. These additional statistics can later be used in the evaluation of a simulation. In the current version, the stage index, at which the egress of a packet was set, as well as the memory address of the corresponding egress entry, are attached to the packet.

4.6 Utilities

This section describes various smaller components and utilities of the Fibulator.

4.6.1 Bits Representation

An important design decision was determining how to implement a generic memory abstraction that is capable of representing the bits that are stored in the different memory modules. To achieve maximum flexibility, this memory container should allow for single-bit granularity. In C++, two promising approaches come to mind: `std::vector<bool>` and `std::bitset<N>`. However, both have their pros and cons as shown in Table 4.1.

	`std::vector<bool>`	`std::bitset<N>`
Pros	- Supports dynamic size changes at runtime	- Small memory and performance overhead - Supports useful operators for equally-sized bitsets
Cons	- Memory and performance overhead - No out-of-the-box operators between two instances with equal length	- Size needs to be known at compile-time - Many template instantiations for different sizes - Leads to code that is hard to write and verbose

Table 4.1: Comparison of `std::vector<bool>` and `std::bitset<N>`

Using `std::bitset<N>` would be desirable due to its many already implemented functionalities and its small overhead. However, during development it became clear that the size of a bitset having to be known at compile-time imposes too many restrictions. In particular, the code becomes very verbose due to many template usages. That is why it seemed beneficial to implement an own `Bits` class that aims to embody the desirable properties from both options.

The `Bits` class uses a `std::vector` of bytes (`uint8_t`) as its internal data representation. The individual bits are stored in a compact way by converting sequences of 8 bits to one byte. The size of a `Bits` instance can be changed at runtime by resizing the `std::vector`. It exposes many functionalities and operators that would be available for two equally-sized `std::bitset`s, as, for example, bitwise AND or flipping bits at certain indices.

However, the `Bits` class is not complete and could be further extended in future work if additional features are required. We suspect that a big portion of the Fibulator's memory usage stems from a suboptimal way of storing `Bits` instances. Each simulation uses multiple memory components containing many entries, typically resulting in a total of more than one million entries. Usually, an entry consists of fewer than 100 bits and is internally represented as a `Bits` instance. Each of those `Bits` instances handles its content with a `std::vector` that is allocated on the heap. The large number of small heap allocations might cause fragmentation and lead to inefficient memory usage. Since the Fibulator was still able to run on a standard commercial machine while simulating real world FIB sizes, addressing this issue was not prioritized.

4.6.2 IP Addresses and Prefixes

The Fibulator project contains lightweight `IPv4Helper` and `IPv6Helper` classes to simplify the handling of IP addresses in C++. This includes, among other things, converting IP addresses from a string to a `Bits` representation and vice versa. All IP addresses are stored as `Bits` instances with sizes 32 and 128 for IPv4 and IPv6, respectively.

Also included are utilities to handle prefixes (or networks). A prefix is stored as two `Bits` instances. The first one holds the actual address and the second, equally-sized one, is used as a mask according to the prefix length. This implies that checking if an address belongs to a certain prefix can be done by checking equality after applying the mask with bitwise AND to the address in question.

4.6.3 Memory Logging

To provide modular data collection capabilities, all memory components are required to implement the `IMemoryLogger`. This memory logger expects the snapshot functionality of a memory component to be implemented. A snapshot of a memory component contains, for example, the number of bits currently used and the number of reads and writes a component has handled yet. A user can add more metrics to a snapshot according to his needs when evaluating a simulation.

The resulting memory log of a snapshot is structured hierarchically. Each component from the data plane, down to the individual pipelines, stages and memory components, recursively generates its memory log by requesting all children logs and aggregating them to their own log. This memory log structure can be saved to a file and later analyzed in the evaluation process.

4.6.4 CSV Input and Output

All inputs and outputs from and to the Fibulator are handled in the Comma-Separated Values (CSV) format. This format was chosen because of its simplicity that allows for a straightforward way of serializing and parsing objects.

Serialization

Serialization is handled by the `ISerializable` interface. Every object that should have CSV serialization capabilities needs to implement this interface. This includes defining either of the following functionalities:

1. `serialize_item`: This function is used for the serialization of simple data structures that can be serialized to a single line in a CSV. For example, each packet corresponds to one row of a CSV.

2. `serialize:` This function is used for more complex data structures if the representation of an object in a single row is not possible. An example is the recursive `MemoryLog` structure from section 4.6.3.

Parsing

In order to be able to parse an object from a CSV, the object in question needs to implement a static `parse_item` function. This function serves as a factory function to construct an object from a `std::vector` of strings, i.e., a row of a CSV.

4.6.5 Debug Printing

The Fibulator contains a simple debug printing mechanism. The `debug.h` file in the `util` directory contains different debugging definitions to enable or disable debug printing for a given component. This allows for an easy toggling of debug printing in a desired part of the Fibulator.

Chapter 5

Evaluation

This chapter presents the results obtained from a variable-stride trie, implemented as a proof of concept of the Fibulator. All results in this section are based on IPv4 addresses, but the same experiments could be performed for IPv6. In addition, it contains a short performance benchmark of the Fibulator.

5.1 Variable-Stride Implementation

This section presents the insights that can be gained using Fibulator simulations. The variable-stride trie implementation, as described in Section 4.5, is used to run the simulations. The analysis of the forwarded packet traces and the memory statistics is performed in Python. The code used for the evaluation of the simulations can be found in the `python` directory of the Fibulator project.

The following three questions will be discussed in the following sections.

1. How is a typical FIB structured with respect to different prefix lengths and overall distribution?
2. How do the number of stages in a pipeline and the stride configuration impact the memory required to store a FIB in a variable-stride trie hardware implementation?
3. What access patterns occur in the different memory components during the forwarding of a typical traffic trace?

Especially the second and the third research questions can be analyzed using Fibulator simulations. The first question is primarily intended to allow for a better contextualization of the other two questions. Before diving into more detail, a very useful visualization technique, used in this chapter, is presented: space-filling curves.

5.1.1 Space-Filling Curves

Visualizing an entire FIB or an IP address space in general can be challenging because of the potentially large ranges involved. The IPv4 standard contains 2^{32} (roughly 4 billion) possible addresses, which is too few for the entire Internet, but still too many to be visualized on a one-dimensional plot. Luckily, space-filling curves [8] can be used to map one-dimensional data series like IP addresses to a two-dimensional image representation.

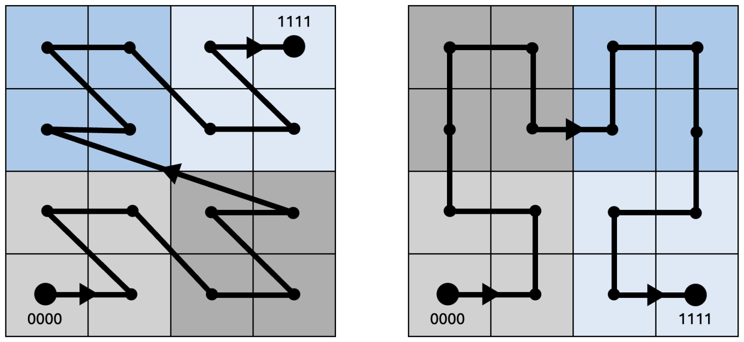

Figure 5.1: 4-bit example of a Morton curve, mirrored along the x-axis, (left) and a Hilbert curve (right)

Both the Morton curve, also known as z-curve, and the Hilbert curve are examples of space-filling curves that preserve spatial locality. Both curves are good candidates to be used for IP addresses since all addresses belonging to the same prefix, i.e., networks containing a power of two number of addresses, are always confined to the same rectangle. Figure 5.1 illustrates an example of a Morton curve and a Hilbert curve using 4-bit addresses. The in total 16 addresses are clearly grouped into the four /2 prefixes (marked as four different colors), containing four entries each. This grouping property also holds for longer addresses and other prefix lengths thanks to the recursive definition of both the Morton curve and the Hilbert curve.

This thesis uses a Morton curve representation, which is mirrored along the x-axis. The Morton curve was preferred over the Hilbert curve because of its always identical and therefore more intuitive (z-)ordering of IP addresses within a given prefix.

5.1.2 Experimental Setup

The Fibulator requires a traffic trace and a FIB to perform a simulation. Apart from generating FIBs and packet traces at random, the Fibulator project also contain utility scripts to generate realistic traces and FIBs from real-world data.

To obtain realistic traffic traces, Packet Capture (PCAP) files from the MAWI archive [7] were used. A Python script extracts the destination addresses and packet sizes to create a simplified packet trace that can be used as an input for the Fibulator. For all subsequent experiments, the first one million packets of a packet trace from `samplepoint-F` from February 1, 2025, were extracted.

To generate realistic FIBs, MRT files from the RouteViews project [5] were used. These MRT files contain snapshots of RIB tables taken from different vantage points. A Python script processes the `TABLE_DUMP_V2` entries with subtype `RIB_IPV4_UNICAST` to extract IPv4 prefixes and their associated next hops. Finally, next hops are mapped to numeric port identifiers to simplify further processing. In particular, for all evaluations, a RIB table from `route-views2.oregon-ix.net` was selected. The snapshot is from February 1, 2025 and results in a FIB containing 1'030'747 entries. This FIB will be further analyzed in the next section.

5.1.3 Real-World FIB Distribution

This section addresses the first of the three research questions: how is a typical FIB structured with respect to different prefix lengths and overall distribution? As described in the experimental setup, the evaluated FIB was extracted from a real-world RIB dump that contains roughly one million entries.

Prefix Length Distribution

Figure 5.2 provides two different perspectives of the prefix length distribution. The left plot shows the absolute number of entries belonging to a certain prefix length in this FIB. The /0 entry corresponds to the default route. The FIB does not contain any entries for prefix lengths smaller than 8. The number of entries per prefix length grows with increasing prefix length from 8 to 24. Two peaks can be observed at a prefix length of 16 and 24. The /24 FIB entries alone make up 61% of all entries. It is common for routers to not store entries with a prefix length longer than 24 to, for example, reduce the risk of BGP hijacking. However, this FIB contains entries for all lengths ranging from 8 to 32.

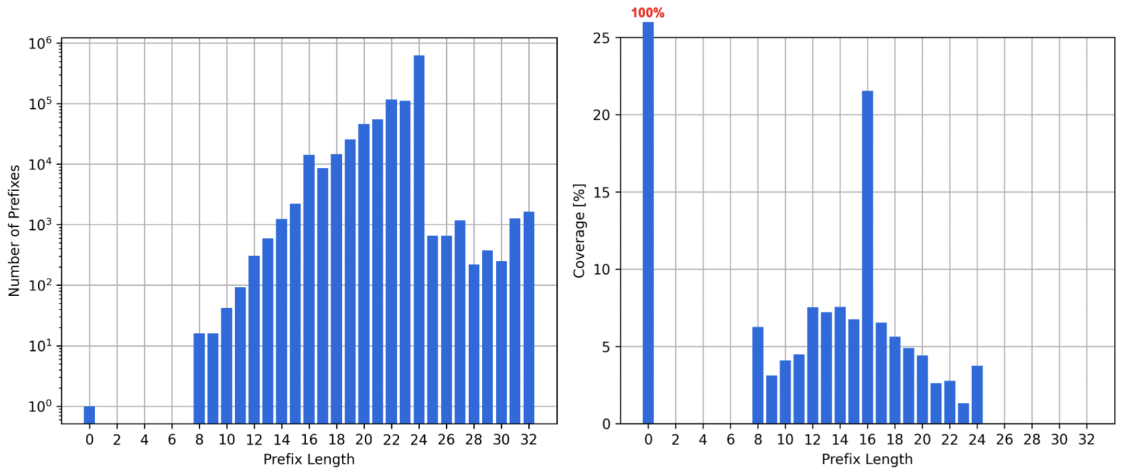

Figure 5.2: Number of entries per prefix length (left) and coverage per prefix length (right).

The right plot depicts the coverage of all entries belonging to a given prefix length. The coverage is calculated as the number of prefixes in the FIB belonging to the same prefix length, divided by the total number of prefixes available at the respective prefix length. For example, there exist 256 /8 prefixes, which means that the 16 /8 entries result in a coverage of $\frac{16}{256} = 6.25\%$. This can be interpreted as the probability of a random packet being covered by one of the entries belonging to a certain prefix length. This could potentially be a good indicator on how many packets will be forwarded by certain prefix lengths. Interestingly, the entries with prefix length 16 have by far the largest coverage of around 22% (apart from /0 which obviously has a 100% coverage). The /24 entries only have a coverage of around 3.7%. The number of possible prefixes per prefix length grows faster than the number of entries per prefix length in this FIB.

Spatial Prefix Arrangement

However, Figure 5.2 offers no insights regarding the arrangement and possible overlaps of the different entries belonging to certain prefix lengths. Figure 5.3 visualizes the spatial arrangement of the different prefix lengths using a Morton curve. To reduce the resolution of the graphics, only prefix lengths up to /24 are depicted. Both plots have a resolution of 4096 x 4096 ($= 2^{12} \cdot 2^{12} = 2^{24}$) pixels. Each pixel corresponds to a single /24 prefix. The default route (/0 entry) is depicted as black pixels. To provide a better orientation, all /8 prefixes of the FIB are labeled accordingly from 0 to 255.

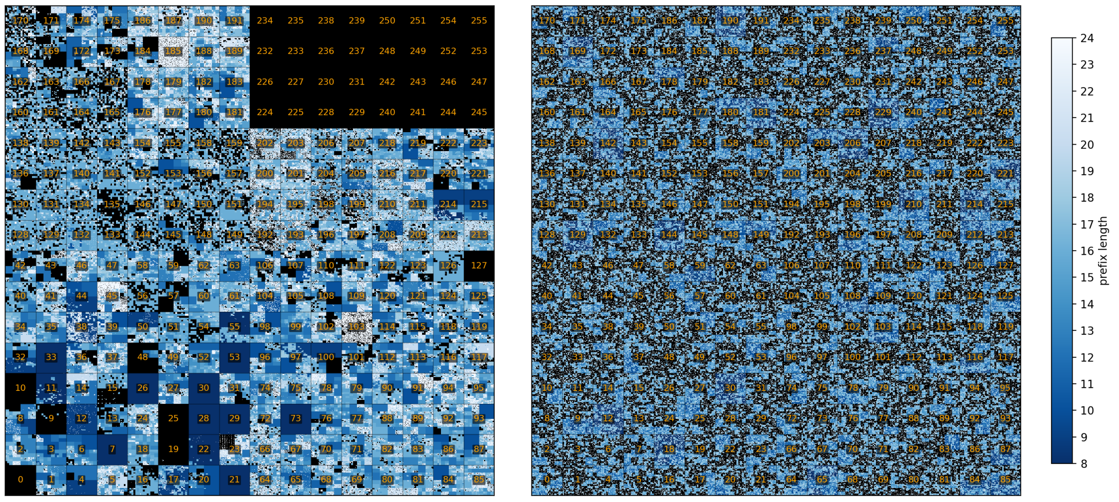

Figure 5.3: Morton curve visualization of a real-world FIB (left) and a randomly generated FIB with the same prefix length distribution (right).

The left plot of Figure 5.3 depicts the real-world FIB. It can be seen that, for example, 10.0.0.0/8 (private address space) and 127.0.0.0/8 (loopback) are not represented in the FIB, which makes perfect sense. The /8 prefixes ranging from 224.0.0.0/8 to 255.0.0.0/8 which are reserved for `MULTICAST` and `FUTURE USE`, are not mapped as well. It can be observed that the FIB looks generally quite structured and there is little overlap of entries belonging to different prefix lengths. This becomes especially clear, when comparing it to the randomly generated FIB on the right, which has the same prefix length distribution.

In order to compare the overlap of the entries, belonging to different prefix lengths, between a random and a real-world FIB, the coverage of all the entries with a prefix length between 8 and 24 was calculated. All black pixels, which correspond to the default route, count as not covered, whereas the rest counts as covered. The two FIBs can be compared regarding their coverage since they both contain an equal number of prefixes, following the same prefix length distribution. A smaller coverage implies more overlapping prefixes. The real-world FIB achieves a coverage of 72.6%, while the prefixes of the randomly generated FIB only cover 63.3% of all addresses. This confirms the less overlapping visual impression of the real-world FIB Morton plot.

Conclusion

When storing a FIB in a trie structure it is important to understand its prefix length distribution. The /24 entries alone make up 61% of all FIB entries. However, they only achieve a coverage of 3.7%. At 22%, the /16 prefix entries have the greatest coverage. Another interesting feature of real-world FIBs is their distinct spatial arrangement. They have significantly less overlaps than a randomly generated FIB with the same prefix length distribution. This illustrates that randomly generating a FIB according to a realistic prefix length distribution might not always lead to realistic results.

5.1.4 Impact of Stride Configuration on Memory Usage

This section discusses the second of the previously presented research questions: how do the number of stages in a pipeline and the stride configuration impact the memory required to store a FIB in a variable-stride trie hardware implementation? One of the main challenges encountered is the large number of possible stride configurations. Since the Fibulator requires some time to finish its simulation, it is difficult to find an optimal stride configuration for a given number of stages.

Selecting a stride configuration is equivalent to the problem of distributing n balls (the number of bits of an IP address) among k bins (the different stages). This is a typical combinatorics problem and the number of possibilities is equal to $\binom{n+k-1}{k-1}$. When we add the additional constraint that each stage needs to handle at least one bit of a destination address, which translates to each bin being required to contain at least one ball, we obtain $\binom{n-1}{k-1}$ possible stride configurations. For an IPv4 implementation ($n = 32$) with, for example, $k = 8$ stages, this leads to more than 2.6 million possible configurations.

Constant-Stride Comparison

A good starting point to analyze how the number of stages relates to the required memory is by first looking at equally sized (constant) strides only. Figure 5.4 presents the different memory usages for storing the FIB in a given constant-stride configuration. Four different constant stride lengths were tested: 8, 4, 2 and 1, leading to 4, 8, 16 and 32 stages respectively.

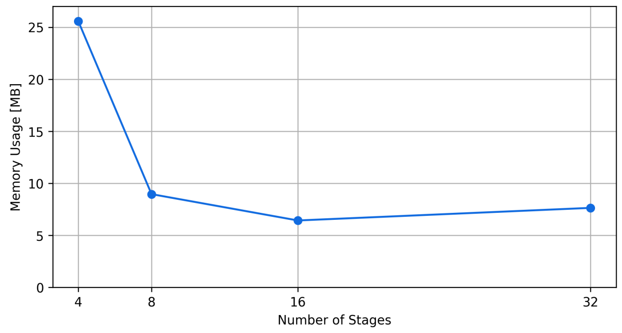

Figure 5.4: Memory usage of FIB for different constant strides.

It can be observed that increasing the number of stages can generally help to decrease the memory footprint of a FIB. This can be mainly explained by the fact that fewer prefixes need to be extended (see Section 2.2.1 for more details). However, reducing the constant stride from 2 to 1 (going from 16 to 32 stages) leads to an increase in memory usage. This behavior is due to many stages functioning primarily as "pointer stages", with the majority of their entries serving as pointers to the next stage rather than storing egress information. Hence, there is a trade-off between not introducing too many pointers in a stage and the number of prefixes that need to be extended. It was generally observed throughout all simulations that a stride of 2 is the smallest stride, not leading to an increase in memory usage.

Split-Value of Stages

We have seen that additional stages can reduce the memory needed to store a FIB. However, it is still unclear, where exactly those additional stages should be added. The question could be formulated as follows: are there positions in the pipeline, where inserting additional stages is more beneficial than in others?

In order to answer this question, the notion of a *split-value* of a stride or stage is introduced. This split-value is defined as the factor by which the total memory, needed to store a FIB, gets reduced, when a stage with a certain stride is split into two stages, each having only half the original stride. Therefore, a split-value of 1 corresponds to no effect on the memory usage, while bigger split-values correspond to more memory being saved. To obtain split-values, a constant stride of 8 (resulting in 4 stages) and a constant stride of 4 (resulting in 8 stages) were used as a base case.

Figure 5.5 shows the split-values of the different stages, displaying the base case of a constant stride of 8 on the left and the base case of a constant stride of 4 on the right. For example, in the left plot, the split-value of stage $[1, 8]$ is equivalent to the factor, the memory usage is reduced by using a stride configuration of $[4, 4, 8, 8, 8]$ instead of the $[8, 8, 8, 8]$ base case, i.e., splitting stage $[1, 8]$ into two stages $[1, 4]$ and $[5, 8]$.

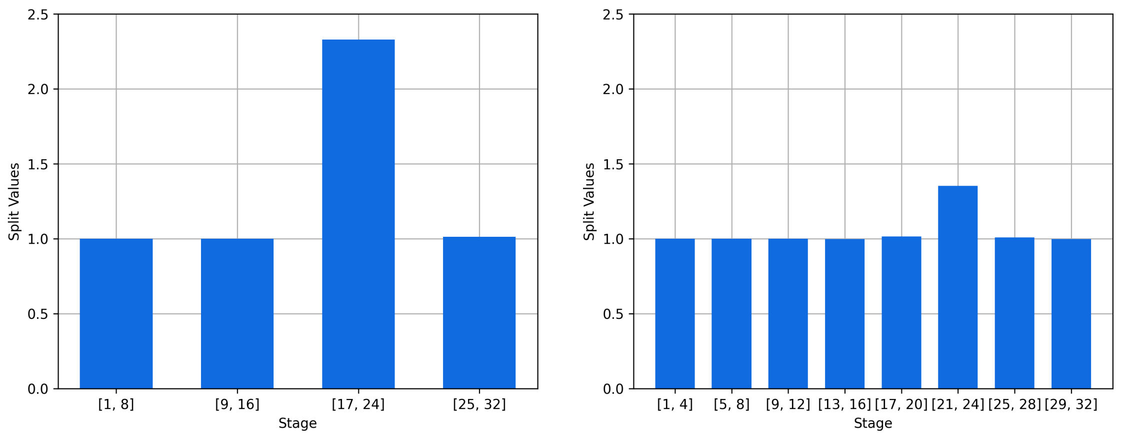

Figure 5.5: Split-values for configurations with a constant stride of 8 (left) and 4 (right)

From Figure 5.5 it becomes clear that choosing smaller strides at the right positions is crucial. In particular, in the left plot, splitting the stage corresponding to the stride [17, 24], containing the egresses of the FIB entries with prefix lengths from 17 to 24, results in a reduction of the memory usage by a factor of 2.3. This can be explained by the fact that most of the FIB entries are located in this range. The memory reduction is due to fewer prefixes having to be extended. Splitting the other strides has a negligible effect on the memory usage and would only introduce unnecessary stages and latency. The right plot with the base case of a constant stride of 4 displays similar insights. Only splitting the stage responsible for prefix length from 21 to 24 results in a significant memory usage reduction by a factor of 1.35. In general, the splitting value decreases as the stride length of a stage becomes shorter.

It was observed that the splitting values of a constant-stride 2 base case tend to be even below 1, meaning further splitting the strides results in an increase in memory. Additionally, it became clear that even merging the first two strides in the $[8, 8, 8, 8]$ configuration, leading to a $[16, 8, 8]$ stride configuration does not result in a significant increase in memory usage. This can be explained by the fact that the first 16 bits can only result in a worst-case of $2^{16} = 65'536$ entries, which is only a small fraction of the total number of entries a FIB typically stores.

This leads to the next experiment with the assumptions that 1) the first 16 bits can be combined into one stage and 2) Adding extra stages is only interesting in the range of prefix lengths from 17 to 24. When only looking at this small range, simulating all possible stride configuration becomes feasible. The following experiment assumes a fixed number of 5 stages, leaving 3 variable stages to handle the relevant middle part, containing the /17 to /24 prefixes. Figure 5.6 shows all the different configurations for the range $[17, 24]$, sorted by their corresponding memory usage. This again demonstrates the importance of choosing the correct stride configuration. Even only varying the stride configuration of 8 bits, while keeping the total number of stages constant, can lead to memory footprints that vary by a factor of 2.1, when comparing the best and the worst configuration.

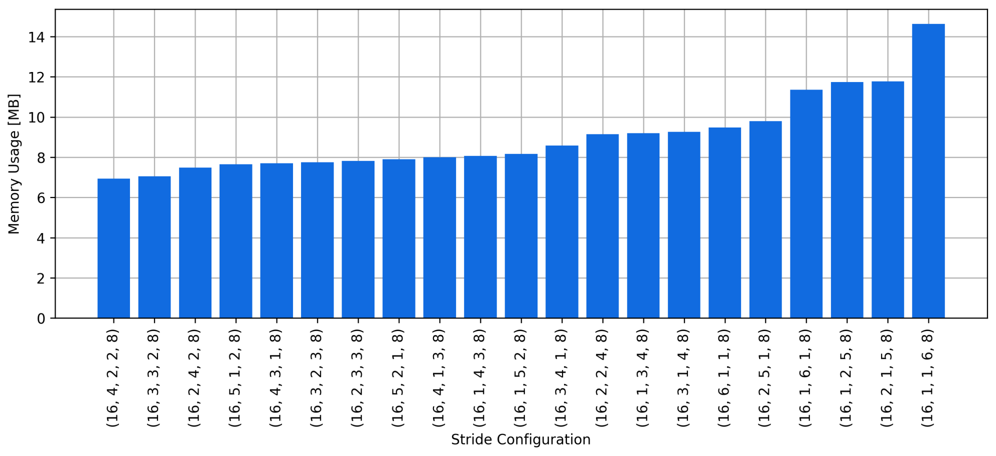

Figure 5.6: Memory usage of different stride configurations with varying middle parts.

The stride configuration $[16, 4, 2, 2, 8]$ is the best configuration found for a total of 5 stages, assuming all strides are a power of 2. There might exist even better configurations for 5 stages, but the potential memory savings are expected to be negligible. Figure 5.7 shows a more detailed plot of the $[16, 4, 2, 2, 8]$ configuration. It displays the number of entries used and the pointer-to-egress-entries ratio of each stage. The width of the bars are proportional to the stride a stage covers. For comparison, Figure 5.8 displays the same plot for a constant-stride 4 configuration.

The constant-stride 4 configuration results in a total memory consumption of 8.96 MB, whereas the $[16, 4, 2, 2, 8]$ configuration only requires 6.94 MB. This exemplifies that more stages do not automatically lead to a smaller amount of memory needed to store a FIB. Instead, the positions at which the additional stages are inserted is much more relevant. By comparing Figures 5.7 and 5.8 it can be observed that the constant-stride 4 configuration wastes stages, by maintaining four individual stages which are mostly made of pointer entries, covering the $[1, 16]$ stride. The larger memory footprint is mostly due to stage [21, 24], containing many egress entries, which could be split into two to reduce the number prefixes that need to be extended.

An important aspect that is not considered in the above analysis is the sizes of the RAM modules in each stage, which impacts the pointer sizes that need to be stored in the preceding stage. A bigger module leads to longer addresses and, therefore, longer pointers. For simplicity, the sizes of the different memory modules were chosen such that in theory all possible entries belonging to the respective prefix length ranges could fit into it. For example, a stage handling /16 prefixes would get enough capacity to store 2^{16} memory blocks, leading to the previous stage having to store 16-bit pointers. This is generally much more than actually needed, and reducing the module sizes to what is realistically necessary can reduce the pointer sizes and hence the overall memory usage. Using this optimization, the overall memory footprint of the [16, 4, 2, 2, 8] stride configuration was reduced by an additional 29% from 6.94 MB to 4.91 MB.

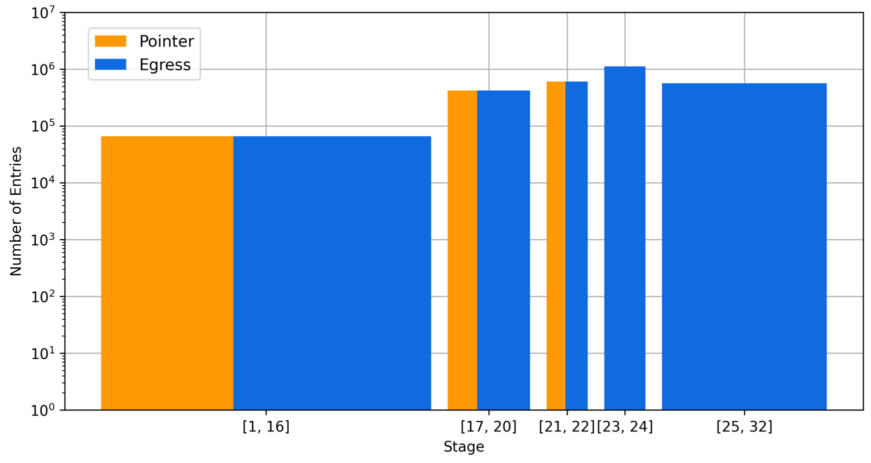

Figure 5.7: Number of entries and egress-to-pointer ratio of each stage using a [16,4,2,2,8] stride configuration. Total memory usage: 6.94 MB

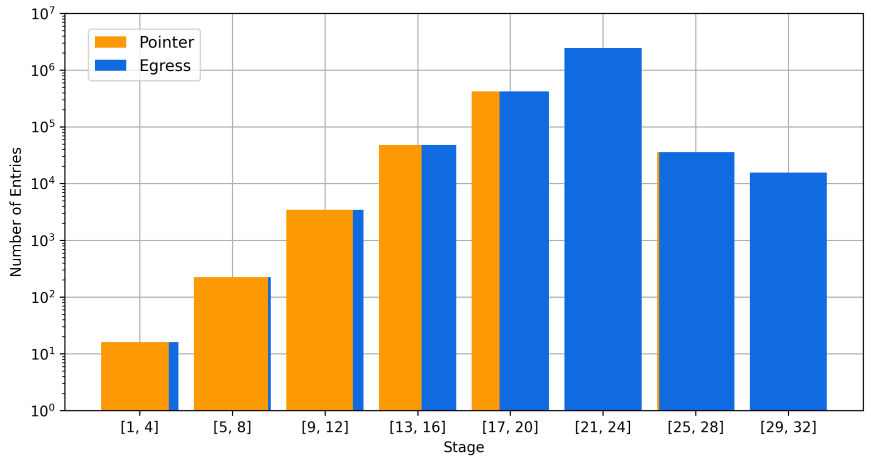

Figure 5.8: Number of entries and egress-to-pointer ratio of each stage using a constant-stride 4 configuration. Total memory usage: 8.96 MB

Conclusion

Using more stages and hence smaller strides can lead to a reduced memory footprint. However, it turns out that selecting the right places to introduce additional stages is much more important

than the mere number of stages.
To find a good stride configuration, two rules of thumb can be applied:

1. Stages that contain many entries, with most of them being egress entries, are generally worth splitting. This aims to reduce the cost of extending many prefixes.

2. Stages that contain few entries, which are mostly pointers to the next stage, can be considered to be merged. This reduces the overall number of stages and potentially reduces memory usage as well.

In addition, it is important to keep memory modules as small as possible. Bigger modules and therefore longer addresses lead to longer next-stage pointers and increased memory usage.

5.1.5 Memory Access Patterns

This section examines the last of the three research questions: what access patterns occur in the different memory components during the forwarding of a realistic traffic trace? As described in the experimental setup, a real-world traffic trace containing one million packets was used to perform the following experiments. It is important to note that the used traffic was captured on the same day as the snapshot of the evaluated FIB, but does not originate from the same location.

An intuitive starting point is to have a look at how many reads, also referred to as accesses in this section, a stage has served after the packet trace has completed. The plot on the left in Figure 5.9 shows the number of reads divided by the total number of packets forwarded in the simulation. This leads to the average number of accesses a packet triggers in each stage. A constant-stride 4 configuration was used to provide an equal granularity across all stages. Unsurprisingly, the first three stages are accessed by almost every packet. This is in line with Figure 5.8 which shows that the first two stages almost exclusively contain pointers to the next stage, leading to no packets having their egress set in these stages.

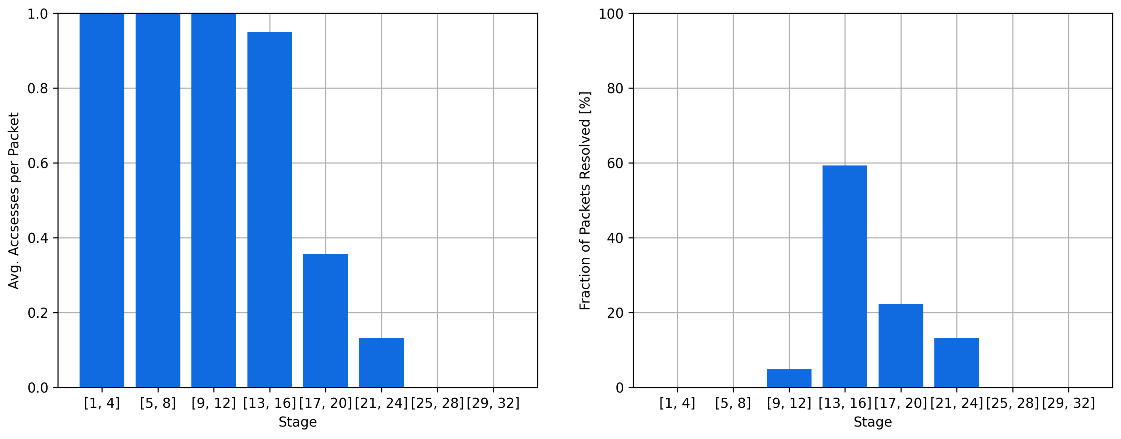

Figure 5.9: Number of both pointer and egress entry accesses per packet (left) and fraction of packets having their egress set per stage (right).

The right plot of Figure 5.9 shows the same information as the left plot from a different perspective. It shows the number of packets that have their egress set at a given stage. This is equivalent to looking at the differences in the left plot. As expected, the distribution looks similar to the coverage plot in Figure 5.2. Almost no packets have their egress set in the first stage through the default route. This makes sense since the default route practically only covers IP address that belong to reserved ranges. Using randomly generated traffic instead of real-world traffic leads to an increased usage of the default route, and hence the first stage, because many reserved addresses are forwarded as well. This implies that using randomly generated traffic on real-world FIBs can lead to distorted results.

However, these aggregated plots do not offer any insights on more granular information, such as, for example, what exact memory entries are accessed by how many packets. In theory, all egresses in a given stage could be set by a single memory entry. Again, a Morton plot can be useful to get a sense of the spatial distribution of memory accesses in a respective stage.

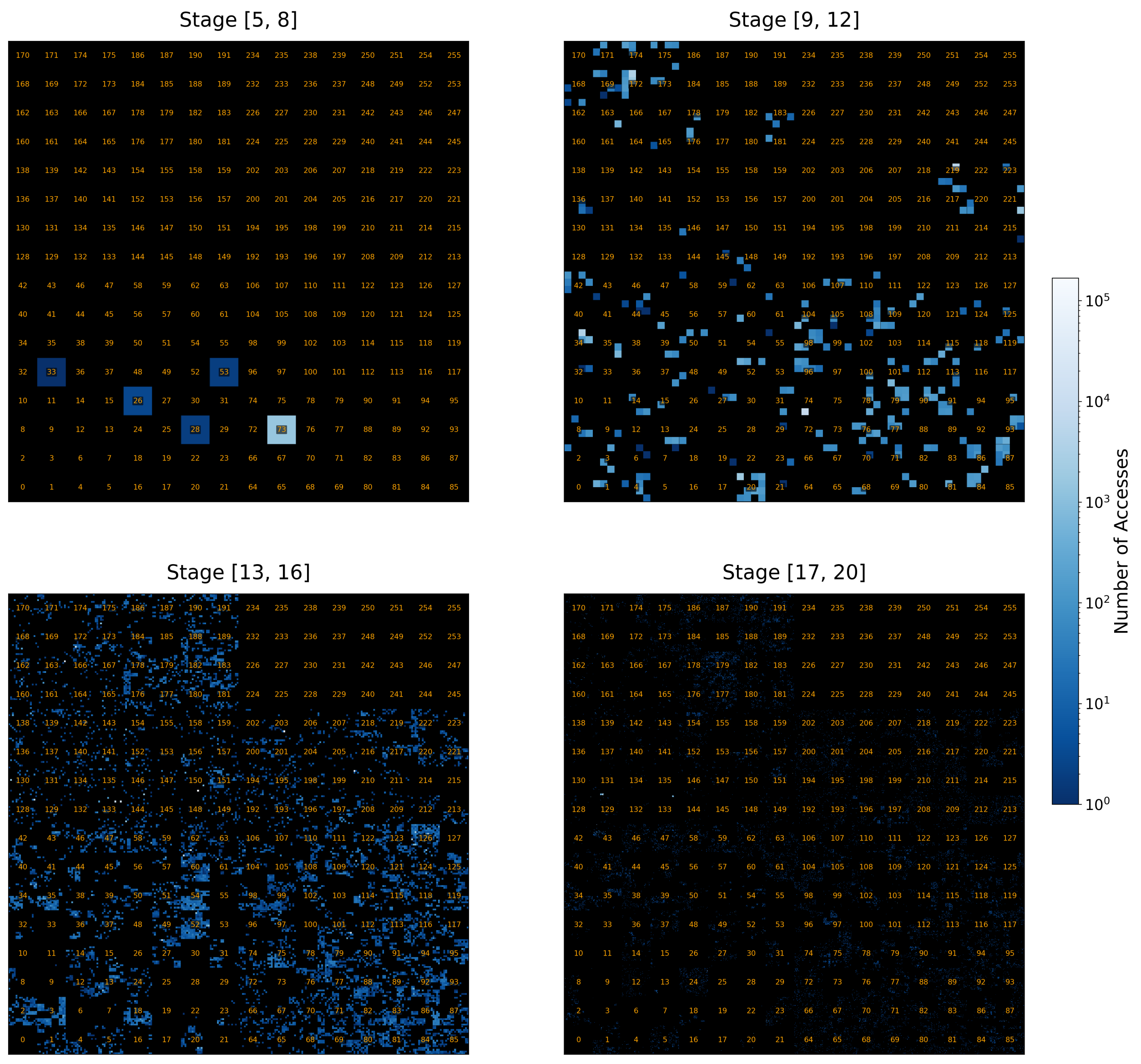

Figure 5.10: Heatmap displaying the accessed egress entries across different stages using a Morton curve.

Figure 5.10 visualizes the spatial accesses of egress entries in the form of a heatmap representation using a Morton curve. It is important to understand that each memory entry in a stage maps to a prefix that has the same length as the longest prefix length contained in its stride. All shorter prefixes need to be extended to multiple entries with this longest prefix length. This explains the resolution of the four different stages. For example, in the plot of the stage covering the stride $[9, 12]$ on the top right, each pixel corresponds to an entry representing a /12 prefix. This leads to a resolution of 64 x 64 ($= 2^6 \cdot 2^6 = 2^{12}$) pixels.

It becomes clear that there exists a high variance in the number of accesses among different egress entries, spanning 5 orders of magnitude. This poses the question: what fraction of egress entries is responsible for resolving, e.g., 90% of all forwarded packets in the span of one million packets?

Figure 5.11 displays the fraction of packets that have their egress set by the fraction of what $x\%$ of the most frequently accessed egress entries. This plot clearly shows the unequal distribution of the number of times different egress entries are accessed. 90% of all packets have their egress set by only 0.18% of all egress entries, which corresponds to 4855 of the roughly 2.7 million entries. 99% of all packets are forwarded by 1.65% of all egress entries. Only the small fraction of 2.1% is responsible for setting the egresses of the entire packet trace. Even among all the entries that were accessed at least once during the simulation, there are big differences in how frequently they were accessed. 90% of all packets have their egress set by roughly 10% of the accessed entries. The orange curve in Figure 5.11 can also be interpreted as a zoomed-in version of the blue curve, stretching out the part containing all the accessed entries.

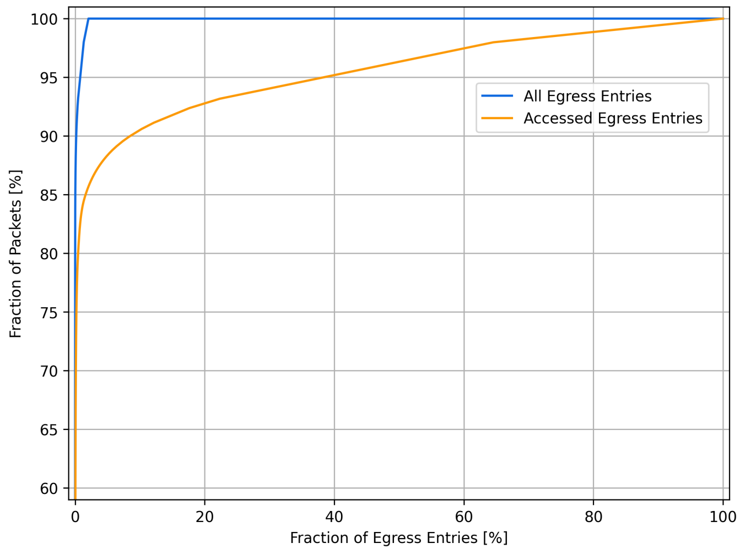

Figure 5.11: What fraction of egress entries is responsible for setting the egress of how many packets.

Of course, it needs to be considered that a packet trace consisting of only one million packets is not even able to access all of the roughly 2.7 million egress entries. Nevertheless, there exists a big imbalance in the number of accesses per egress entry. This demonstrates how effective a caching solution can be in hardware forwarding. We believe that clever caching mechanisms play an important role in achieving high throughput in modern forwarding hardware.

Conclusion

Most of the packets have their egress set through a /16 prefix entry. This aligns with the fact that the /16 prefixes achieve the highest coverage in the evaluated FIB. The number of accesses per egress entry are highly uneven. Only 1.65% of all egress entries are responsible for the forwarding of 99% of the one million packets in the evaluated trace. Therefore, caching of recently used FIB entries is considered to be an important aspect of achieving high throughput.

An interesting side-note is that the implemented trie structure would support caching almost out-of-the-box. A challenge in caching FIB entries is the cache-hiding problem, meaning cached shorter prefixes might *hide* longer, more specific prefixes. However, the normalized trie structure allows for a save cache insertion of each egress node since all egress nodes are leaf nodes with zero children, meaning there exist no longer prefixes. The cache would only need to be cleared (or other measures to prevent cache-hiding would need to be put in place) as soon as the FIB structure is updated.

5.2 Fibulator Performance

This thesis does not include a detailed performance benchmark of the Fibulator. Instead, it presents some approximate measurements to provide a general sense of execution time and memory usage. Both memory usage and execution time mostly depend on the size of the FIB and the length of the packet trace. Potential bottlenecks introduced in the algorithm implementation can slow down the Fibulator as well. The main bottleneck of the current variable-stride trie implementation is the creation of a FIB that is set up according to a strides configurations. Especially configurations with longer strides, where many prefixes need to be recursively extended, lead to a significant increase in execution time.

The benchmark is executed on a MacBook with an M2 chip on a single core. The Fibulator binary is compiled with the build type set to `Release`. The benchmarks are based on the variable-stride trie implementation and use the same setup as described in Section 5.1.2, with a FIB containing roughly one million entries and a packet trace of one million packets. Both a constant-stride 8 and a constant-stride 4 configuration were tested to demonstrate the increased memory usage and execution time for longer strides. The results are shown in Table 5.1.

	Execution Time	**Memory Usage (RAM)**
constant-stride 8	52 s	3.9 GB
constant-stride 4	25 s	1.7 GB

Table 5.1: Execution time and memory usage comparison of the Fibulator using two different stride configurations

Chapter 6

Summary and Outlook

The Fibulator is a modular simulation environment that aims to simplify the prototyping process of new algorithmic ideas for in-hardware forwarding. It can also be used to gain a better understanding of already existing solutions. The following summarizes four important lessons learned during the evaluation of the variable-stride trie implementation:

- **Lesson 1:** Morton curves are a useful tool for visualizing large IP address ranges. This makes them a good fit to be used as a locality-preserving FIB representation that maintains high granularity and preserves spatial information.
- **Lesson 2:** Using randomly generated FIBs or traffic traces can lead to distorted results when evaluating LPM algorithms because they might not contain key characteristics of real world FIBs and packet traces.
- **Lesson 3:** The chosen stride configuration can have a big impact on the memory required to store a FIB. Not necessarily the number of stages, but rather where the additional stages are inserted plays a crucial role.
- **Lesson 4:** Some egress entries are significantly more accessed than others. A small fraction of entries is responsible for most forwarded packets. This is why caching can be very effective in forwarding.

To further improve the simulation capabilities of the Fibulator, future work could focus on implementing additional components and more sophisticated algorithms. In particular, the effects of caching and adding TCAM modules to the pipeline seem to be interesting avenues to be further explored. Another aspect that could be pursued is to improve the performance of the Fibulator. Introducing parallelization and other optimizations may lead to shorter execution times and less memory usage.

The current implementation of the variable-stride trie algorithm only evaluates constant FIBs, meaning the FIB does not change over time. Future work could also focus on evaluating implementations with dynamically changing FIBs to better reflect real-world conditions.

Bibliography

[1] Asai, H., and Ohara, Y. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. *SIGCOMM Comput. Commun. Rev. 45*, 4 (Aug. 2015), 57–70.

[2] Draves, R., King, C., Venkatachary, S., and Zill, B. Constructing optimal ip routing tables. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)* (1999), vol. 1, pp. 88–97 vol.1.

[3] FS.com. N9600-64OD, 64-Port Ethernet HPC/AI Data Center Switch. `https://www.fs.com/de/products/250955.html`, June 2025.

[4] Sobrinho, J. a. L., Vanbever, L., Le, F., and Rexford, J. Distributed route aggregation on the global network. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2014), CoNEXT '14, Association for Computing Machinery, p. 161–172.

[5] University of Oregon Route Views Project. RouteViews: BGP Data Collection Project. `https://routeviews.org/`, June 2025.

[6] Varghese, G., and Xu, J. *Network Algorithmics*, 2 ed. Morgan Kaufmann, 2022.

[7] WIDE Project. MAWI Working Group Traffic Archive. `https://mawi.wide.ad.jp/mawi/`, June 2025.

[8] Wikipedia. Space-filling curve. `https://en.wikipedia.org/wiki/Space-filling_curve`, June 2025.

[9] Yeluri, S. Longest Prefix Matching in Networking Chips. `https://www.linkedin.com/pulse/longest-prefix-matching-networking-chips-sharada-yeluri/`, June 2025. LinkedIn article.

Appendix A

Build and Execution Workflow

The Fibulator project uses *CMake* as a cross-platform build generator. In order to set up and build the project, follow the instructions in the `README` file in the root directory of the Fibulator project.

In order to run a simulation, it is recommended to use the Python wrapper located in the `python` directory. The `run.py` script demonstrates a possible implementation of such a wrapper. This wrapper is tailored for the variable-stride trie implementation. It might need to be adapted when an other algorithm is implemented. The wrapper covers the following steps:

1. Provide a pipeline configuration for which a simulation should be performed. This includes, for example, the number of stages and their corresponding strides, the number of egress ports and the capacities of the different memory modules.
2. Provide the path to the FIB and the packet trace that should be used for the simulation.
3. The Python wrapper automatically generates an `algo_config.cpp` file from the provided configuration.
4. The wrapper builds the project by executing the `make` command and runs the resulting `fibulator` binary.
5. As soon as the Fibulator has finished, `run.py` checks whether all packets were forwarded correctly according to the provided FIB. This is done with the help of a Patricia trie implementation in Python using the `pytricia` module. This only serves as a sanity check and has to be adapted if the simulation involves a changing FIB over time.

Once the `run.py` wrapper finished, the Fibulator should have generated multiple CSV files containing the forwarded packets and the requested memory logs in the `output` directory. The `notebook` subdirectory in the `python` directory provides many different plotting capabilities in the form of Jupyter notebooks that were used for the evaluation in this thesis. The `fibulator_py_tools` directory contains other useful helper functionalities that can be used and adapted for future applications involving the Fibulator.