CoFIB: A Memory-Efficient NDN FIB Design for Programmable Edge Switches

Eduardo Castilho Rosa, Daniel Nunes Corujo, and Flávio de Oliveira Silva, *Senior Member, IEEE*

***Abstract*—Designing high-performance and memory-efficient data structures for the *Forwarding Information Base* (FIB) in *Named-Data Networking* (NDN) is a challenging task. Since the FIB size is orders of magnitude larger than IP routing tables, scaling it to store millions of prefixes in SRAM/TCAM memory in programmable switches is an open problem. To address this issue, we propose a *Compressed FIB* data structure called CoFIB, designed to run on edge programmable switches in an SDN-based environment. The CoFIB is a collection of exact-match tables placed in an optimized manner in both ingress and egress pipelines. We propose a LNPM algorithm that carefully recirculates packets in the pipeline. We also introduce the concept of canonical name prefixes to reduce memory footprint and propose an algorithm to extract canonical prefixes from the *Routing Information Base* (RIB). Experimental results show that CoFIB can compress memory up to 16.58× compared to the state-of-the-art, with no significant impact on throughput compared to hardware-based solutions. Additionally, our proposed table placement optimization for LNPM increases the number of packets processed at a line rate by 23.17% compared to the linear table placement approach using a large NDN name dataset.**

***Index Terms*—FIB, SDN, P4, name lookup.**

I. Introduction

NAMED Data Networking (NDN) [1] is a communication paradigm that provides an innovative way for applications to retrieve data from remote nodes. Instead of using fixed-length IP addresses, NDN communication is based on exchanging Interest Packets (Ipkts) and Data Packets (Dpkts) containing a variable-length name identifier. The *Routing Information Base* (RIB) residing in the NDN control plane is responsible for storing and aggregating routes to named content from different sources (e.g., routing protocols, manually-set routes, etc). The RIB contents are periodically stored in the NDN data plane through the *Forwarding Information Base* (FIB) to allow for fast lookups during the forwarding process [2].

One of the key challenges in the NDN research regarding the data plane is designing the FIB to allow a good trade-off between name lookup performance and memory footprint [3], [4]. Although there are software-based FIB solutions that process NDN packets on a microsecond time scale at *Gbps* [2], [5], scaling NDN to the size of the current Internet requires decreasing per-packet processing latency by at least one order of magnitude, to alleviate the bottleneck at the core. Nevertheless, moving from *Gbps* to *Tbps* in NDN is still challenging, especially regarding the FIB [6].

Developments in re-configurable hardware pipelines, such as RMT [7] and dRMT [8], have enabled implementing the NDN FIB in programmable switches using P4. However, offloading the FIB into the switch's ASIC is quite challenging. First, the FIB is significantly larger than IP routing tables, and the amount of on-chip TCAM/SRAM memory available in today's ASIC is limited. Second, ASIC typically does not support features such as dynamic memory allocation, pointers, and recursion, which make trie-based data structures in hardware difficult or even unfeasible. Bloom filter-based solutions (BF) [9], [10] appear as alternatives to overcome these limitations. Although these approaches are memory-efficient, they are probabilistic models and produce false positives that might impact packet forwarding accuracy.

Hash-based FIB for programmable switches has been proposed to improve latency. In NDN.p4 [11], the FIB is designed as a single P4 table stored in TCAM, containing a sequence of named sub-prefix hashes used as table keys. A Hashtray-based method (HBM) is proposed in [12] to optimize memory by not inserting additional sub-prefixes for each FIB entry. However, neither FIB design includes the control plane implementation. Moreover, although the *Longest Name Prefix Matching* (LNPM) is performed in one pipeline pass, the FIB is deployed only at the ingress pipeline, wasting the on-chip memory available in the egress control block. Given the limited TCAM/SRAM memory available in today's programmable switches and the large amount of memory required to store the FIB, such a design principle is not a good choice regarding scalability.

ENDN [13] and NDNFab [14] implement the control plane and data plane of the NDN architecture using P4 and the SDN paradigm. However, while the former does not implement its underlying FIB data structure (FCTree [15]) in P4, the latter deploys the FIB in slow DRAM memory in the control plane. Even Tofino-based implementations of NDN, such as NDNTofino [6], and Pegasus [16], [17], only offload the

Received 2 January 2025 revised 8 October 2025 accepted 27 November 2025. Date of publication 5 December 2025; date of current version 13 January 2026. This work has been partly developed in the scope of the project EXIGENCE. EXIGENCE has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant 101139120. This work was supported in part by Fundação para a Ciência e Tecnologia under Grant UIDB/00319/2020; in part by Fundação de Amparo à Pesquisa do Estado de São Paulo under Grant 18/23097-3; and in part by Conselho Nacional de Desenvolvimento Científico e Tecnológico under Grant 421944/2021-8. The associate editor coordinating the review of this article and approving it for publication was M. J. Khabbaz. *(Corresponding author: Eduardo Castilho Rosa.)*

Eduardo Castilho Rosa is with the Department of Information Systems, Goiano Federal Institute, Catalão 75705-040, Brazil (e-mail: eduardo.rosa@ifgoiano.edu.br).

Daniel Nunes Corujo is with the Department of Electronics and Telecommunications, Universidade de Aveiro and Instituto de Telecomunicações, 3810-193 Aveiro, Portugal (e-mail: dcorujo@av.it.pt).

Flávio de Oliveira Silva is with the Department of Informatics, University of Minho, 4710-057 Braga, Portugal (e-mail: flavio@di.uminho.pt).

Digital Object Identifier 10.1109/TNSM.2025.3641145

Pending Interest Table (PIT) into the switch ASIC. At the same time, the FIB is stored in commodity servers. In this sense, the research question that we aim to answer in this paper is: *How can we design a FIB data structure for a programmable switch in an SDN-based network that scales to millions of prefixes, fully utilizes the SRAM memory available in the chip, and keep the packet processing latency on a nanosecond time scale*? To answer this question, we propose a *Compressed Forwarding Information Base* (CoFIB) designed to fill the gaps existing in current FIB solutions in the literature.

In contrast to previous P4-based NDN FIB, CoFIB is designed to run on programmable edge switches in an SDN-based network. CoFIB uses *Hash Tables* (HT) as the underlying data structure, similar to most FIB models in the literature. However, instead of using one large P4 table to store the named prefixes, CoFIB reduces the hash collision probability by using multiple tables in the pipeline and a longer hash length. Additionally, CoFIB explores the SRAM memory available in both ingress and egress control units. Consequently, the proposed LNPM algorithm can perform two match-action operations per pipeline pass, requiring multiple packet recirculations for some groups of *Ipkts*.

To minimize the recirculations needed to complete the LNPM, we propose an optimization algorithm to carefully place tables in the ingress and egress pipeline according to name distribution. Furthermore, we introduce the concept of canonical name prefixes to compress the FIB to fit into the small ASIC memory. Along with additional compression methods, our design can scale to millions of named prefixes in SRAM, while keeping the average per-packet latency on a nanosecond time scale. To the best of our knowledge, CoFIB is the first NDN FIB data structure to fully explore SRAM memory availability in programmable ASICs to store name prefixes instead of relying only on scarce, expensive, and power-hungry TCAM memory.

In summary, the main contributions of this paper are as follows:

- A novel NDN FIB data structure, CoFIB, fully compatible with programmable switches. The CoFIB utilizes the SRAM memory available in both ingress and egress pipelines (Section III-B3, and III-B4).
- An LNPM algorithm that carefully recirculates packets in the pipeline, performing match-action operations in both ingress and egress pipeline (Section III-B7).
- The concept of canonical prefixes to improve the compression ratio of the NDN FIB and propose an algorithm to extract canonical name prefixes from the *Routing Information Base* (RIB) (Section III-B4, and III-A2).
- An algorithm to optimize the table placement in the ingress and egress pipeline to reduce the packet recirculations and improve the throughput (Section III-B6).

The remainder of this work is organized as follows. In Section II, we present the related works in literature. In Section III we present the CoFIB data structure in terms of its elements in both data and control plane. In Section IV, we evaluate the CoFIB regarding memory footprint, throughput, and per-packet latency. Finally, in Section V we present our concluding remarks and future works.

II. Related Work

The internal tables of an NDN router are typically implemented using general-purpose data structures such as *Tries*, *Bloom Filters* (BF), and *Hash Tables* (HT) [3]. Regarding the FIB, many software and hardware-based solutions have been proposed to accelerate name forwarding using fixed-function and programmable data planes. This section summarizes the most prominent FIB solutions implemented using different methods and data structures in the literature.

A. Tries

A *Trie* is a tree-like data structure commonly used to store and retrieve strings and dictionaries. It can reduce the memory consumption of the FIB by aggregating common sub-prefixes and naturally supporting the LNPM operation.

Name Component Encoding (NCE) [18] is a type of *trie* designed to store NDN named prefixes into the FIB. The NCE rationale is to assign unique codes to named components in the prefix and to perform the lookup through an in-depth traversal of the *tree*. The memory footprint can be reduced because unique codes are inserted into the *trie* instead of the actual named components. However, the average depth of the *trie* is high using such an approach, impacting the lookup speed. Solutions such as CONSERT [19] and *compactTrie* [20] can be used to reduce the *trie*'s depth. However, they are software-based and difficult to deploy onto physical hardware targets.

In [21], the FIB is compressed in SRAM memory by using two *Patricia tries* (*dualPatricia*). This binary representation provides more opportunities to compress shared sub-prefixes between different prefixes. On the other hand, *Patricia tries* tends to increase the tree's depth, impacting the lookup speed.

OnChipFIB [22] is a FIB design aimed at improving forwarding performance using an FPGA platform [23]. In OnChipFIB, the NDN name is represented as a collection of strides of the same size. These strides are inserted into a binary search tree, and the LNPM is performed through an in-depth search. Regarding memory consumption, the complexity of the OnChipFIB is $O(n)$, where n is the number of names stored in the FIB. However, in the worst case, OnChipFIB is $O(nk)$, where k is the number of strides. Consequently, OnChipFIB can not scale to millions of prefixes.

As a reference software-based forwarder, *NDN Forwarding Daemon* (NFD) [24] implements the FIB by using a HT and a *trie*. One of the main limitations of NFD is that it only uses a single thread to process both *Ipkts* and *Dpkts* in the pipeline, significantly limiting its forwarding throughput. Alternatively, YaNFD [2] is proposed to improve throughput using multi-thread processing. However, as software-based solutions, both NFD and YaNFD have some performance and overhead issues that limit the scalability of such solutions.

FCTree [15] was introduced as a software-based FIB data structure that compresses named prefixes by storing common sub-prefixes only once. However, due to its reliance on recursion and dynamic memory allocation, it poses challenges for ASIC implementation. In contrast, the more recent PtCAM [25] was designed to be deployed in hardware combining a trie-based data structure and a TCAM device. However,

PtCAM can reach high-speed performance only when TCAM and RAM are well-coordinated and carefully engineered.

B. Bloom Filters

BFs are probabilistic data structures used for membership queries. Because BFs are memory-efficient, many BF-based solutions have been proposed to implement the FIB.

The NLAPB [9] consists of an adaptive prefix BF proposed to optimize physical memory consumption. The main feature of NLAPB is to divide the NDN prefixes into two segments and perform the name lookup by combining a counter Bloom Filter (CBF) and a *trie*. Experiments indicate that NLAPB is scalable in terms of memory consumption, even when large *datasets* are used. However, NLAPB design might incur false positives in packet forwarding, and the *trie* impacts the LNPM latency for longer prefixes. Additionally, NLAPB is designed to run in software and does not fit well into hardware targets.

MaFIB [28] is a BF-based FIB that employs a *Mapping Bloom Filter* (MBF) proposed in [33]. The MBF filter combines a regular BF and a bit vector stored in SRAM. The BF is used to check whether elements exist in the MBF, and the bit vector is used to address the output interfaces stored in DRAM. Compared to NCE and similar methods, MaFIB provides a better FIB compression rate. However, its main limitation is the high frequency of access to DRAM memory to extract the output interfaces. Additionally, false positives might occur, compromising the accuracy of forwarding.

B-MaFIB improves MaFIB's memory footprint by modifying the MBF structure using a Bit Mapping Bloom Filter (B-MBF). The central idea of B-MBF is to enable dynamic memory allocation to reduce memory consumption. However, B-MaFIB still needs to access DRAM memory to read information such as output ports and time to live.

An FPGA-based FIB that uses BF as the underlying data structure is proposed in [30]. The FIB prefixes in *fAccelerator* are divided into two groups based on the number of components. The prefixes of the first group are stored in an external lookup table that runs in software, while the ones from the second group are stored in the FPGA's on-chip memory. This division is due to the limited on-chip memory of modern FPGA boards. The main limitation of *fAccelerator* is the high latency caused by frequent external memory access to perform name lookups.

C. Hash Tables

HT is another data structure used to design the FIB. Since HT offers the advantage of fast name lookup, many hash-based FIB solutions have been proposed recently.

The first content-based router supporting high-speed name forwarding is Caesar [26]. Caesar's router distributes the FIB across different line cards and performs LNPM separately for each prefix subgroup. Caesar uses HT based on the *crc64* functions to store prefixes in the FIB. Segregating the FIB among different line cards optimizes memory usage but increases LNPM complexity and the number of packet-switching operations. Therefore, Caesar can not process *Ipkts* within a nanosecond time scale.

SACS [29] is a FIB design that combines HT and *tries*. Prefixes in SACS are transformed into shapes before being stored in a trie-based data structure. False positives are eliminated using precursor pointers in each hash table in a *trie* shape. When a named prefix does not match any entry in a hash table, SACS uses pointers to perform the LNPM. Compared to traditional HT-based methods, SACS improves LNPM latency since several memory accesses are eliminated due to prefix shapes. However, SACS requires more memory when compared to software-based solutions such as NFD [24].

NDN-DPDK [5] is the state-of-the-art software NDN forwarder in terms of performance, whose table design is a modified version of a *2-stage* LPM introduced in [31]. NDN-DPDK implements the FIB using an HT keyed by the name prefixes, aiming to speed forwarding up to 100 Gbps while running on commodity x86 hardware. However, in NDN-DPDK design, the input thread that processes *Ipkts* becomes the bottleneck when 8+ forwarding threads are used, which might impact the scalability. In addition, NDN-DPDK does not provide any memory consumption analysis to assess the feasibility of the FIB design when millions of named prefixes are stored.

Over the last few years, hash-based FIBs that use P4 and programmable switches have been proposed. NDN.p4 [11] appears as the first attempt to implement the NDN router in P4. The FIB in NDN.p4 is designed as one match-action table at the ingress pipeline and configured to store fixed-size entries. A Hashtray-based method (HBM) [12] is proposed to reduce memory footprint when storing named prefixes into programmable ASICs. NDN.p4 and HBM compress the FIB entries using a *crc16* hash function, which increases collision probability. Moreover, half of the on-chip memory in both solutions is wasted because the FIB table is located only at the ingress pipeline. Therefore, the FIB can't scale to store millions of prefixes using this approach.

A P4-based NDN router is implemented on a Tofino switch ASIC in [6], [16], and [17]. However, only *Dpkts* are forwarded by the switch ASIC, while the *Ipkts* are redirected to an NDN engine at the control plane to be processed in software. The approach to implement the FIB using the more abundant DRAM memory available in the control plane can easily scale to millions and even billions of named prefixes. Nevertheless, this approach severely impacts the name lookup performance, increasing the LNPM latency in the FIB to the order of microseconds or even milliseconds.

While the focus remains on Content-Centric Networking (CCNx) [34], [35], a unified status table (U-Table) [32] is a recent FPGA-based solution that unifies FIB, PIT, and *Content Store* (CS) into one single table. U-Table can store 10 million named prefixes using HT. However, named prefixes are stored in large-capacity but low-speed DRAM, which impacts the LNPM performance. The SRAM memory available on the FPGA board is dedicated to resolving hash collisions.

Table I summarizes the most prominent FIB solutions in the literature. We present the papers according to hardware and software features, programmability support, and scalability, indicating the level of support for a given feature. Based

TABLE I
SUMMARY OF THE RELATED WORKS

FIB Design	HW Based	P4 Based	SDN Based	Data Structure	Hardware Features						HW/SW Co-Design	LNPM Latency Magnitude	On-Chip Memory Scalability
					Use of DRAM	Use of TCAM	Use of SRAM	Memory Optimization	Throughput Optimization	Parallelism Support			
NCE [18]	○	○	○	Trie	●	○	○	●	○	○	○	ms	○
Caesar [26]	●	○	○	HT	○	○	●	○	○	●	○	μs	○
fatTree [27]	○	○	○	Trie,HT	●	○	○	●	○	○	○	ms	○
NLAPB [9]	○	○	○	BF,Trie	●	○	○	●	○	○	●	ms	○
dualPatricia [21]	◐	○	○	Trie	◐	○	◐	○	●	○	●	μs	○
compactTrie [20]	○	○	○	Trie	●	○	○	●	○	○	○	ms	○
CONSERT [19]	○	○	○	Trie	●	○	○	●	●	○	○	ms	○
NDN.p4 [11]	●	●	○	HT	○	●	○	○	○	●	○	ns	○
MaFIB [28]	●	○	○	BF	◐	○	◐	○	○	●	●	ms	○
B-MaFIB [10]	●	○	○	BF	◐	○	◐	●	●	●	●	ms	○
SACS [29]	●	○	○	HT,Trie	○	●	●	○	●	○	○	μs	○
HBM [12]	●	●	○	HT	○	●	○	◐	○	●	○	ns	○
fAcellerator [30]	●	○	○	BF	●	○	●	○	●	●	●	μs	○
OnChipFIB [22]	●	○	○	Trie	○	○	●	○	○	●	○	μs	●
2-stage LPM [31]	○	○	○	HT	●	○	○	○	○	○	○	μs	○
NDN-DPDK [5]	○	◐	○	HT	●	○	○	○	●	○	○	μs	○
FCTree [15]	○	○	●	Trie	●	○	○	●	○	○	○	ms	○
NFD [24]	○	○	○	Trie,HT	●	○	○	○	○	○	○	ms	○
YaNFD [2]	○	○	○	Trie	●	○	○	○	●	●	○	μs	○
NDNFab [14]	○	◐	●	HT	●	○	○	○	○	●	●	μs	○
NDNTofino [6]	○	◐	○	HT	●	○	○	○	○	●	●	μs	○
Pegasus [16], [17]	○	◐	○	HT	●	○	○	○	○	●	●	μs	○
U-Table [32]	●	◐	○	HT	○	○	●	○	○	●	○	μs	○
PtCAM [25]	●	○	○	Trie	◐	◐	○	●	○	●	○	ns	●
CoFIB	●	●	●	HT	◐	◐	●	●	●	●	●	ns	●

The symbol (◐) indicates a partial support for a given feature.

on the literature, we highlight two significant limitations of the current FIB solutions for programmable switches. Firstly, except for NDN.p4 and HBM, all other NDN routers designed for programmable switches deploy the FIB into the control plane to the best of our knowledge. Secondly, proposed solutions that offload the FIB into ASIC memory use only the ingress control block, wasting half of the switch's resources. Additionally, they do not provide any optimization to balance memory consumption with name lookup latency. Therefore, CoFIB is designed to fill these gaps, scaling to millions of named prefixes while keeping the lookup latency on a theoretical nanosecond time scale.

III. CoFIB Design

Scaling NDN to the size of the current Internet requires increasing the *Ipkts* processing rate at the core and optimizing memory usage in NDN routers. In this regard, offloading the NDN FIB into the hardware is crucial. At the same time, storing all named prefixes in the data plane is impossible due to limited on-chip memory. In this sense, co-design approaches appear as a compromise between flexibility and performance. Following this rationale, this section presents the CoFIB as an approach to offload part of the NDN FIB into the data plane using programmable edge switches.

The CoFIB data structure is presented in terms of its control and data plane elements. Regarding the control plane, we provide an overview of the components implemented as *Virtual Network Functions* (VNF) and eBPF/XDP program. These components are used to create the P4 table entries and to forward *Ipkts* stored at the control plane efficiently. Additionally, we detail all the elements in the data plane and the LNPM algorithm.

Fig. 1 presents an overview of the architecture on which the CoFIB is designed to be deployed. It is important to note that this architecture is not the only one possible. The data plane comprises programmable *Edge Switches* (ESw) connected to

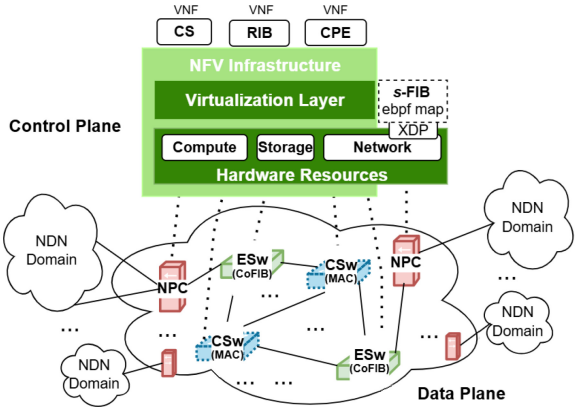

Fig. 1. CoFIB architecture use case.

hybrid *Core Switches* (CSw) in a mesh topology. These switches are configured centrally through an SDN controller integrated with NFV Infrastructure. We have designed some data structures as VNFs in the control plane to support the CoFIB that runs on the data plane.

As shown in Fig. 1, the CoFIB is deployed only at the edge. This configuration improves the end-to-end throughput because CoFIB might delay the processing for some *Ipkts* while the packets in the core can be forwarded at the line rate. The hybrid core switches can be programmable or fixed-function, where forwarding is done by performing an exact match lookup at the MAC table.

Edge programmable switches are connected to devices named *NDN Packet Converter* (NPC). An NPC element extracts prefixes from transit *Dpkts* and sends them to the control plane as prefix announcements. Additionally, the NPC converts native NDN packets that cross the SDN domain into packets encoded in a P4-friendly format to facilitate parsing.

Algorithm 1 Canonical Prefix Extractor (CPE)

1: INPUT: Prefixes $P = \{p_1, ..., p_m\}$ from RIB
2: OUTPUT: f-FIB and s-FIB
3: Create hash table H;
4: **for each** $p_j \in P$ **do** {populating H}
5: **for each** nc_i in p_j **do**
6: $v \leftarrow H.get(nc_i)$;
7: **if** $v == null$ **then** $H.put(nc_i, [0, .., 0])$;
8: **else** $v[i] \leftarrow v[i] + 1$; $H.replace(nc_i, v)$;
9: **for each** $p_j \in P$ **do** {calculating weights}
10: $w \leftarrow 0$;
11: **for each** nc_i in p_j **do**
12: $v \leftarrow H.get(nc_i)$; $w \leftarrow w + v[i]$;
13: **set** weight w to p_j
14: **add** p_j into an ordered list L based on its weights
15: **for each** p_j in L **do** {extracting canonical names}
16: **if** p_j is non-canonical **then**
17: s-FIB.$put(p_j)$;
18: **for each** nc_i in p_j **do**
19: $v \leftarrow H.get(nc_i)$;
20: **if** $v[i] > 0$ **then**
21: $v[i] \leftarrow v[i] - 1$; $H.replace(nc_i, v)$;
22: **else** f-FIB.$put(p_j)$
23: **for each** p_j in f-FIB **do** {removing shared prefixes}
24: **if** p_j share common prefix in s-FIB **then**
25: f-FIB.$remove(p_j)$; s-FIB.$put(p_j)$

A. Control Plane

The control plane uses NFV technology to manage native NDN tables (e.g., RIB, CS) as well as additional data structures designed to perform specific functions within the SDN domain (e.g., *s*-FIB). In the following sections, we present some of these data structures.

1) Routing Information Base: In NDN, the RIB is a data structure running in the control plane that stores and aggregates routes pointing to data contents whose prefixes were learned from different sources. The RIB in our design stores the named prefixes added by the operator and the prefixes announced by NDN nodes. Advertised prefixes include those the NPC sends to the controller and the *Dpkts* from specific routing protocols such as NLSR [36].

Each edge switch is connected to an NPC device, which can be connected to one or more native NDN routers, as shown in Fig. 1. This work assumes that only one NDN router connects to an NPC. Thus, each entry in the RIB consists of only the announced prefix and the *ids* of the switches (*swId*) that received such an announcement.

It is known that the amount of on-chip memory available in today's programmable switches is limited. Thus, the *swId* parameter is encoded as an 8-bit value to optimize memory consumption on edge switches. The *swId* represents the edge switches that point to where *Dpkts* containing that given prefix can be fetched. For example, if *swId* is 00010001, the *Ipkt* that contains the correspondent named prefix will be forwarded to switches 1 and 5 using a multicast group. Because *swId* is encoded using 8 *bits*, this approach supports up to 8 edge switches per domain, where each ESw_i is identified by the value 2^i.

The RIB is implemented as a hash table $H : K \rightarrow V$, where *K* represents the named prefixes and *V* the *swIds* that point to where the *Dpkts* containing the prefix can be located. Representing the RIB as a hash table allows the *swIds* parameters to be computed quickly, considering the complexity of accessing an element in the hash table is *O*(1).

2) Canonical Prefix Extractor (CPE): In NDN, applications can freely define their own namespaces. The definition of namespace used in our study is given by [37], referring to a structured and hierarchical collection of names used to identify and organize data within an NDN domain. Because conventions and agreements on which naming scheme should be used are still an open issue in NDN [38], [39], depending on the applications' characteristics, we can assume a namespace containing only canonical prefixes. The formal definition of a canonical name prefix is shown in Def. 1. Overall, a given prefix is canonical if all its components appear in the same position in all prefixes in the FIB. Thus, the hierarchical nature of the NDN names reinforces the idea of having canonical namespaces. However, in cases where such an assumption cannot be made, such as at the NDN core, we propose an algorithm called *Canonical Prefix Extractor* (CPE), to extract canonical named prefixes offline from the RIB and populate the P4 tables in the data plane, as shown in Algorithm 1.

Definition 1 (Canonical Name Prefix): For a given set of *m* prefixes $P = \{\rho_1, .., \rho_m\}$, a prefix $\rho_j \in P$, represented by a sequence of components $/nc_1/nc_2/../nc_k$, is considered canonical if all possible occurrences of its nc_i components in all prefixes in *P*, for $1 \leq i \leq k$, appear at position *i*.

The key idea of CPE is to associate an array $[v_1, .., v_8]$ for each component nc_j in all prefixes in the RIB, where v_i is the frequency of nc_j at the position *i*. A given prefix is canonical if the arrays of its components nc_i contain 0 in all indexes except *i*. CPE calculates the weight for each prefix $/nc_1/../nc_k$ in the RIB by summing up v_i in the arrays of each nc_i for $1 \leq i \leq k$. Then, the prefixes are sorted in descending order according to the weight values. The canonical ones are inserted sequentially into the P4 tables, called *Fast Path* FIB (*f*-FIB). At the same time, the non-canonicals are offloaded from the RIB in the userspace to the *Slow Path* FIB (*s*-FIB) in the *Network Interface Card* (NIC) space. Algorithm 1 details the CPE operation.

3) Name Component Hash Table: The data structure CPE uses to split the RIB into canonical and non-canonical prefixes is the *Name Component Hash Table* (NCH). NCH is created between lines 4-8 of Algorithm 1 and is implemented as a hash table $H : K \rightarrow V$, with *K* being the set of all named components extracted from the RIB and *V* the set of all position mapping vectors associated with the components from prefixes in the RIB.

Initially, the position mapping vectors are reset, and iteratively, as the named components are extracted from the RIB and inserted into NCH, the indices of these vectors are incremented. Because NCH uses a hash table, the time complexity of CPE is *O*(*kn*), where *k* is the maximum number of components supported and *n* is the number of prefixes in the RIB. As *k* is constant in our design, the complexity of CPE is reduced to *O*(*n*).

The non-canonical prefixes extracted from the RIB by CPE will be offloaded to *s*-FIB, implemented as an eBPF program attached in the eXpressed Data Path (XDP) hook point. Because the non-canonical prefixes can't be stored in the data plane, offloading the *s*-FIB to the NIC space is a trade-off between performance and flexibility. When no matching

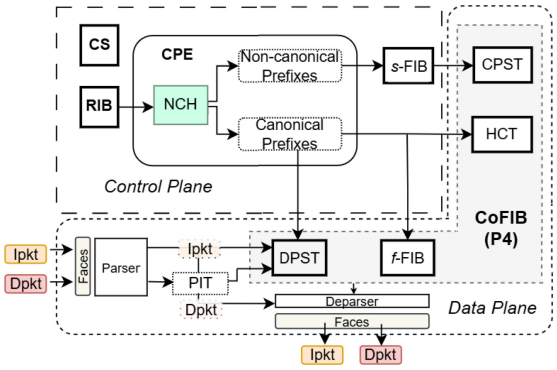

Fig. 2. Data Plane and Control Plane Elements.

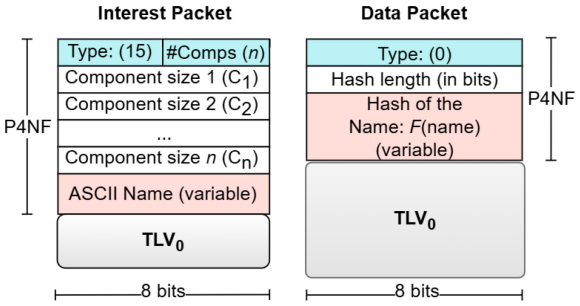

Fig. 3. P4NF Format.

prefixes exist in CoFIB on the data plane, *Ipkts* arriving at edge switches can be forwarded to the control plane. Since *s*-FIB is implemented in XDP/eBPF, the LNPM in the control plane can be performed at high speed.

One problem that might occur in our approach is the cache-hiding problem [40]. Such a condition happens when the LNPM in CoFIB returns a prefix, but a longer one exists in *s*-FIB. To solve this problem, after CPE extracts the canonical prefixes from the RIB, those that share sub-prefixes with the non-canonical set are moved to the *s*-FIB in the control plane. This operation is detailed in lines 23-25 of Algorithm 1.

B. Data Plane

In our design, the CoFIB is responsible for forwarding *Ipkts* from one edge switch to another through a fabric containing hybrid core switches. Hence, the SDN-based network on which CoFIB is deployed can be abstracted as a large switch connecting multiple NDN domains.

The canonical and non-canonical prefixes generated in the control plane are used to populate the *Control Plane Shape Table* (CPST) and the *Data Plane Shape Table* (DPST) in the data plane. In addition, the *Hash Conflicting Table* (HCT) is populated based on the canonical prefixes extracted by CPE. The tables CPST, DPST, and HCT are detailed in the following sections. Fig. 2 summarizes the main components of the CoFIB in the control and data plane.

Considering that CoFIB is deployed only at the edge, the memory footprint and the per-packet processing latency are reduced at the core switches since performing the LNPM at each hop is unnecessary.

In our approach, *Ipkts* converted successfully by the NPC are encapsulated in a special Ethernet header containing the tuple $\langle marker, swId \rangle$, where the marker is a standardized sequence of bits used to distinguish packets that have already undergone the LNPM operation.

Since core switches can be programmable or fixed-function, the exact match key for determining the output ports varies. For example, if the core switch (S_i) is fixed-function, the SDN controller will randomly select one of the edge switches indicated in *swId* (S_k) and compute the shortest path between S_i and S_k. The MAC table at switch S_i will be populated with the tuple $\langle swId, port \rangle$, where the value *swId* is accommodated in the Ethernet destination MAC address. The value *port* is the interface of S_i linked to the shortest path between S_i and S_k. If the core switch is programmable, the MAC table is also used but interpreting the *port* parameter as a *multicast* group of 16 bits that includes the edge switches indicated in *swId*. Notably, multicast is only possible if the core switches are programmable.

1) P4 Name Friendly Format: The first challenge when designing an NDN FIB that runs on programmable switches is how to manipulate variable-length names efficiently in P4, since the language does not support string-based processing. The original TLV format in NDN imposes some complexity on the header parsing due to the possibility that a name contains nested TLV sub-blocks. In addition, for each named component to be identified separately, an NDN name with n components is encoded using n TLV blocks. As an NDN name is composed of a prefix and a suffix, several assignment operations must be done on the ingress pipeline to concatenate all named components into a single key so that the lookup operation can be performed correctly since the suffix is ignored in the FIB lookup.

To simplify the processing of NDN names in P4, we propose *P4 Name Friendly Format* (P4NF), a new name encoding for NDN packets that adheres to the P4 target constraints and aims to facilitate parsing and lookup operations. The P4NF format is hybrid in the sense that it contains specific fields to represent the name, as well as native TLV blocks representing other packet fields. As shown in Fig. 3, in the proposed format, the lengths of each prefix component are placed in sequence and precede the other fields that remain in TLV format.

For *Ipkts*, the first four bits in the P4NF format are set to 1, with the following four bits representing the number of components in the prefix. Conventionally, although four bits make it possible to represent up to 15 components, the maximum number of components supported in our design is 8. This is not a problem since larger NDN prefixes are uncommon, according to statistical analyses performed on name datasets in [41]. Thus, the second group of four bits contains values ranging from 1 to 8, where any value other than this makes the *Ipkt* invalid. The following n fields (C_i) store the sizes of each component represented in P4 as a *header_stack* construct, where C_i is an 8-bit value. Although 8 bits allow the representation of components of up to 255 characters to reduce the on-chip memory consumption when storing prefixes in CoFIB, the maximum number of characters per component supported in our design is 31. The following

field stores the name NDN encoded in ASCII with 8-bit characters. This field is represented in P4 using the *varbit* type. The length of this field (in bytes) is obtained iteratively during the header extraction process by the sum $L = \sum_{i=1}^{n} C_i$, where $1 \leq L \leq 248$.

Since LNPM is not required for *Dpkts*, encoding this type of packet is more straightforward and consists of preceding the TLV_0 block with the P4NF header as shown in Fig. 3. This block's *type* field is represented as an 8-bit constant containing a sequence of zeros. The *length* field is also represented in 8 *bits*, indicating the hash size of the prefix computed in the NPC. Thus, the proposed format supports *hashes* of up to 255 *bits*. By convention, in this work, the hash size in *Dpkts* is set to 32. The following *value* field stores the name *hash* itself computed in the NPC.

2) Shape Tables (CPST and DPST): To avoid processing *Ipkts* containing named prefixes that do not exist in CoFIB, we use the concept of prefix shapes, initially introduced by [29]. The idea is to avoid unnecessary LNPM when the prefix does not exist in the *on-chip* memory. In simple terms, the prefix shape represents the sequence of lengths of each component in the name in their respective positions. For example, the shape for the prefix */br/ufu/facom* is */2/3/5*. Thus, two supplementary tables are used in the data plane in addition to those required to store the canonical named prefixes. The first table is the DPST and stores the shapes of all canonical prefixes existing in the data plane. The second is the CPST, which stores the shapes of prefixes existing in the control plane, specifically in *s*-FIB. Both DPST and CPST are stored in TCAM, where each table entry is given by the shape of the prefix as key and the number of its components as action data.

Given that entries in both CPST and DPST tables need to have a fixed length due to the requirements of the P4 language, it is necessary to convert variable-length shapes into fixed-size shapes. This conversion is performed by concatenating the lengths of individual components and padding them with zeros up to a maximum limit. To illustrate, the shape for the prefix $/nc_1/../nc_k$ is given by a 40-bit key obtained by concatenating $l(nc_1)$++,..,++$l(nc_k)$++$p(k)$, where k is the number of components, $l(x)$ is a function returning a 5-bit value representing the number of characters in component x, and $p(k)$ is a function returning a string of zeros whose length is given by $40-5k$. Thus, if a lookup in the DPST table hits, the maximum number of components is obtained by calculating $max=8-k$.

When an *Ipkt* arrives at the NDN router, the DPST and CPST tables are queried to check for any matching shape. If the prefix shape contained in the *Ipkt* does not exist in these tables, the packet is dropped according to the standard operational flow of the NDN architecture. This avoids unnecessary LNPM and contributes to reducing packet processing latency. On the other hand, if the prefix exists only in the DPST or the CPST, the LNPM is executed by recirculating the packet in the pipeline, or the packet is sent to the control plane to be queried in the *s*-FIB, respectively. The core idea of using these tables is to reduce the number of unnecessary recirculations in the processing pipeline.

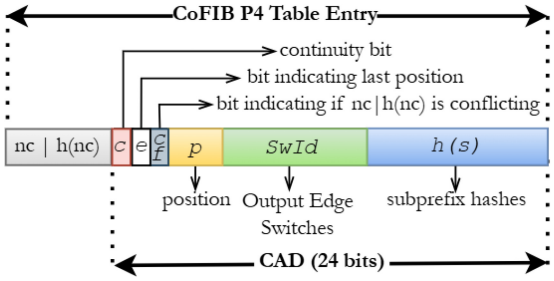

Fig. 4. CAD Structure.

3) Fast Path FIB: CoFIB uses a set of P4 tables $T_1, .., T_k$, namely f-FIB, distributed across both the ingress and egress pipeline. Each table T_i stores named components with i characters, for $1 \leq i \leq k$, and is configured with an exact-match key to force the compiler to use SRAM memory. Because each named component's size is encoded as a 5-bit value, $k=31$ in our design.

To conserve memory, the width of table T_i in bits is set to $8i$ when $1 \leq i \leq 4$ or 32 for $i>4$. Thus, table T_i stores named components in ASCII when $1 \leq i \leq 4$, while the remaining tables ($T_{i>4}$) store *crc32* hashes obtained from the named components. For example, the name prefix */ab/cde/fghij* is stored in CoFIB by inserting '*ab*' into table T_2, '*cde*' into table T_3, and $h(fghij)$ into table T_5, where $h(x)$=**crc32**(x).

As observed, the CoFIB stores named components individually in different tables according to their length. An issue that arises from this is how to associate each component so that it is possible to identify each named prefix. The following subsection describes how this problem is solved.

4) CAD Structure: To attach individual named components to form the name prefix, we associate each P4 table entry with a value, namely *Component Action Data* (CAD). CAD includes essential information to enable the LNPM in the data plane. Thus, a FIB entry in CoFIB is represented by the tuple $\langle nc, cad \rangle$, where *nc* is the named component in ASCII when $1 \leq |nc| \leq 4$, or *h*(*nc*), otherwise.

Although the CAD structure must include the necessary information to enable the LNPM, it must use as few bits as possible to reduce its memory footprint. As shown in Section III-A2, we introduce in this paper the concept of canonical name prefixes to decrease the number of bits in the CAD. It is worth noting that the definition of canonical prefix presented in this paper differs from the canonical name (CNAME) used in the Domain Name System (DNS) [42].

To illustrate what a canonical prefix is, let's consider two sets of named prefixes: P={*/a/b/c*, */x/b*, */f/b/c/t*} and P'={*/a/b/c*, */b/c*, */x*}. As we can see, all prefixes in P are canonical because every named component in all prefixes appears in only one position. In contrast, in P' the prefix */b/c* is non-canonical since the named component '*b*' appears in the second position in */a/b/c* and in the first position in */b/c*. Assuming the prefixes in the FIB are canonicals, we can encode the position of any named component by using only three bits.

As we can see in Fig. 4, the CAD structure is given by the 6-tuple $cad=\langle c, e, cf, p, SwID, hs \rangle$ represented by a 24-bit

value. Thus, the FIB entry is given by the tuple $\langle nc, cad \rangle$. The bit *cad.c* in the CAD indicates if there are prefixes in the FIB containing the corresponding named component *nc* (or its hash *h*(*nc*)) and at least one named component at a position greater than the *nc*|*h*(*nc*) position. For example, for a set of canonical name prefixes *P*={*/a/b/c*, */x/b*, */f/b/c/t*}, the bit *cad.c* associated with the named component '*c*' and 't' are 1 and 0, respectively. The bit *cad.e* indicates if the named component *nc*|*h*(*nc*) appears at least once in the last position in any prefix. Taking a set *P* as an example, the bit *cad.e* associated with the named components '*c*' and 'a' are 1 and 0, respectively. The next bit *cad.cf* indicates whether the prefix that contains the corresponding *nc*|*h(nc)* is conflicting. Considering the set *P*, the bit *cad.cf* associated with the named components '*b*' and '*t*' are 1 and 0, respectively. A conflicting name prefix is formally described in Def. 2.

Definition 2 (Conflicting Name Prefix): Let *P* be a set of canonical prefixes and $\rho \in P$, with ρ=/nc_1/nc_2/../nc_k where nc_i=named component at position *i*. We consider ρ a conflicting name prefix if and only if $\exists \rho' \in P$ with $\rho \neq \rho'$ and ρ'=/nc'_1/nc'_2/../nc'_q, such that $\exists i$ where $2 \leq i \leq k$ with $nc_i = nc'_i$ and (/nc_1/../nc_{i-})$\neq$(/nc'_1/../nc'_{i-1}).

A conflicting name prefix contains at least one named component nc_j such that $nc_j.cad.cf = 1$. In such a case, an exact-match table, namely *Hash Conflicting Table* (HCT), is used to store the tuple $\langle key, swId \rangle$, where *key* is a 32-bit value given by the function $F(\rho, nc_i)$ described in Eq. (1), considering *m* a large odd number, and *swId*=switch *ids*, if $nc_i.cad.e = 1$, or *swId* = 0 otherwise ($nc_i.cad.e = 0$).

$$F(\rho, nc_i) = \begin{cases} m * F(\rho, nc_{i-1}) + h(nc_i), & \text{if } i > 1. \\ h(nc_1), & \text{otherwise.} \end{cases} \quad (1)$$

To understand the need to use the recurrence relation instead of calculating the hash of the sub-prefix at each iteration, it is necessary to comprehend how prefix matching is performed in the CoFIB. The LNPM operation is executed iteratively for each component of the name. Thus, the named component must be individually read from the packet and put into a fixed-size header at each iteration. This enables searching for the component (or its hash) in the corresponding P4 table. However, the hash value of the component at a given position is insufficient to identify the entire sub-prefix. Thus, it is necessary to successfully combine hash values as they are calculated at each match-action on f-FIB to perform the LNPM. The recurrence relation presented in Eq. (1) serves to perform this combination.

Since every prefix inserted into the P4 tables is canonical, only one possible position *i* at which any named component will appear, where $1 \leq i \leq 8$. This information is encoded in the 3-bit field *cad.p*. For the named prefix set *P* presented earlier, the *cad.p* associated with the named components '*b*' and '*c*' are 2 and 3, respectively.

The last field in the CAD structure is a 10-bit value (*cad.hs*) to uniquely identify the sub-prefix associated with the current named component $nc_i|h(nc_i)$. As *cad.hs* is a 10-bit field, its value is given by the 10 high-order bits obtained from $F(\rho, nc_{i-1})$. For example, if ρ=*/a/abcde*, then nc_2=*abcde* with $nc_2.cad.hs$=(*F*(*/a/abcde,a*)$\gg$22) and nc_1=a with $nc_1.cad.hs$=0, since there is no sub-prefix for components at the first position. With this configuration, we can support multicast with eight (8) possible output ports.

5) CoFIB Memory Optimization: The use of *crc32* hashes to compress some named components, the concept of canonical prefixes, and the reduced number of bits in the CAD structure are some of the memory optimization techniques used in this paper to allow CoFIB to fit into the limited TCAM/SRAM memory. Because the P4 tables required to store CoFIB contain only one action, it is possible to optimize memory consumption even further by using *action profile* and *action selector* constructs according to the P4 spec [43].

To illustrate, a traditional P4 table with *M* entries will typically consume *M*W* bits in the data plane, where *W* is the size of the action parameters. If the action parameters of the *M* entries are different from each other, it might not be possible to reduce the memory footprint. However, for a given table *T* that requires at most *N* different set of action parameter values, with *N*<*M*, the *action profile* construct can reduce the on-chip memory footprint by grouping multiple actions and using indirect tables. As the CoFIB tables require only one action to be executed regardless of the table lookup hits or misses, we implement a version of CoFIB that uses *action profiles* and *action selectors* instead of P4 tables. The memory consumption of a particular table *T* utilizing this approach is given by Eq. (2):

$$T_{mem} = M * log_2(N) + N * W \quad (2)$$

It is noteworthy that, in some cases, using *action profile* and *action selectors* might increase the memory consumption of a particular table. This includes scenarios where there are many distinct sets of action parameters. For example, assuming *W*=96 bits, a FIB with *M*=1,000 entries, and 900 distinct action parameters (*N*=900), then the memory consumption using a P4 table will be 96,000 bits, while using *action profile* will require 96,400 bits of storage. Therefore, to use *action profiles* instead of conventional P4 tables, the inequality in Eq. (3) must be true, where the key size in bits (*K*) is 8*i* for table T_i, with *i*<5 or *K*=32 for *i*>4. According to the characteristics of some name datasets used in this paper, all CoFIB tables are suitable to be implemented using *action profiles*.

$$M * (K + log_2(N)) + 24 * N < (K + 24) * M \quad (3)$$

6) CoFIB Throughput Optimization: The amount of TCAM/SRAM memory available in today's programmable ASICs is typically distributed equally across both ingress and egress pipelines. In such architecture, it would be much simpler to place the CoFIB tables only at the ingress pipeline to simplify the LNPM logic. However, such an approach would waste half of the on-chip memory. Thus, the design goal of CoFIB is to distribute the tables across both the ingress and egress pipelines. A straw-man solution is to place the tables linearly. We refer to this approach as a linear table placement, where the first 14 tables in f-FIB ($T_{i<15}$) are placed at ingress while the remaining tables ($T_{i\geq 15}$) are placed at egress.

The linear table placement strategy has some consequences. One is that the HCT table must be duplicated in both the

Algorithm 2 Table Placement Optimization

```
1: INPUT: Prefixes P = {p_1, ..., p_m} from RIB
2: OUTPUT: Ingress (IG) and Egress (EG) hash tables.
3: Let M be a matrix such that M ∈ Z^{32×9} with M[i][j] storing the # of
   components with size i at position j
4: Let H be an array of hash tables where H[i] stores crc32 digest of components
   with i characters
5: for each p_j ∈ P do
6:     for each nc_i in p_j do
7:         len ← length(nc_i); h ←crc32(nc_i)
8:         H[len].putIfAbsent(h)
9:         M[len][i] ← M[len][i] + 1
10: Let V be an array containing a serialized matrix M where V[i] is the triple
    ⟨frequency, position, size⟩ associated with each component nc_i.
11: Sort the array V in descending order based on frequency
12: for i from 1 to |V| do
13:     if V[i].frequency == 0 then
14:         break;
15:         tableName ← "T" + V[i].size;
16:         if (i mod 2) == 1 then
17:             if tableName ∉ (IG ∪ EG) then
18:                 IG.put(tableName);
19:         else
20:             if tableName ∉ (IG ∪ EG) then
21:                 EG.put(tableName);
22: return IG, EG
```

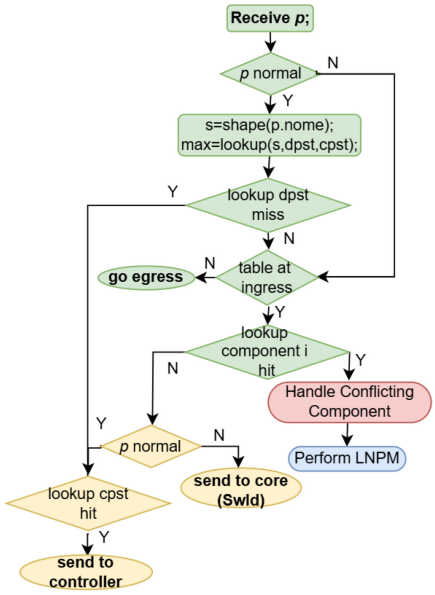

Fig. 5. Ingress Pipeline Processing.

ingress and egress pipelines. This is because the LNPM needs to identify conflicting prefixes whenever a match-action operation occurs. However, as we will see in the next section, since the HCT table size is some orders of magnitude smaller than f-FIB tables, such redundancy will not significantly impact the memory footprint.

Besides utilizing all the on-chip memory available in the switch ASIC, a linear table placement approach enables two match-action operations per pipeline pass in the LNPM. This is significant since it might reduce the number of pipeline passes for some packets, reducing the average per-packet latency and increasing throughput. For example, *Ipkts* containing two named components might experience only one pipeline pass if the first component matches at ingress and the second at egress. Two pipeline passes would be required to complete the LNPM if the tables are placed only at the ingress or egress.

The match-action in CoFIB works as follows. Whenever a table lookup occurs, whether at ingress or egress, an action is invoked to extract the *SwId* and the other fields from the CAD. In traditional FIB implementations, such as [15], [44], the FIB runs in all NDN routers, and the output interfaces to send the packet are retrieved from the memory when a match occurs. However, CoFIB is designed to run only at the edge switches. Therefore, the *SwId* does not point directly to the output faces but to the edge switches that can retrieve the corresponding *Dpkts*. The output ports to send the *Ipcks* to the core switches are selected randomly. This approach is impossible in traditional FIB implementations because the decision about which ports to send the packets to must be taken only at the ingress pipeline.

Despite the benefit of using a linear table placement, a question that comes up at this point is: *Is it possible to rearrange the CoFIB tables in both ingress and egress pipelines in a way to reduce the number of pipeline passes in contrast to the linear table placement approach*? To answer this question, we have to analyze the characteristics of the canonical prefixes, such as the size of the named components and their respective positions, before they are inserted into the CoFIB.

Based on the canonical prefixes extracted from the RIB, we propose a method to rearrange the tables in the pipeline, as shown in Algorithm 2. The idea is to calculate the frequency of each named component and iteratively place the tables at ingress and egress pipelines according to the named components that are more frequent, their respective sizes, and positions. For example, the named component '*com*' is the most frequent among all canonical prefixes and appears at position 1. Therefore, we place table T_3 at ingress. Then, let's suppose the second more frequent component is '*data*'. This indicates that table T_4 can be placed at egress. Consequently, *Ipkts* that carry prefixes containing both components will likely experience two matches in one single pipeline pass than the linear table placement. Algorithm 2 shows the table placement strategy in more detail.

7) Pipeline Processing Logic: Dpkt processing is beyond the scope of this work. However, to support *Dpkt* forwarding and complete the NDN packet flow, we provide a simple PIT implementation. The PIT is realized using P4 registers that store full name hashes. Upon the arrival of a *Dpkt*, an exact-match lookup is performed in the PIT. If the name hash is found, the packet is forwarded to the corresponding outgoing interfaces; otherwise, the *Dpkt* is dropped.

Each packet entering the pipeline carries additional information in the metadata structure provided by the target P4 [43]. The metadata is used to guide the LNPM process and enable the identification of whether the packet is entering the pipeline for the first time (normal packet) or is being recirculated or resubmitted. This is because certain operations will only be performed for normal packets, while others will be performed for all packets, including the ones that pass multiple times in the pipeline.

Algorithm 3 Handle Conflicting Component

```
1: INPUT: Ipkt (p), cad, component (nc), position (i)
2: if cad.cf is set then
3:     if lookup at HCT miss then
4:         Drop p;
5:     else
6:         Update SwId; Perform LNPM;
7: else
8:     if i>1 then
9:         if cad.hs == (F(p,nc_i >> 22) then
10:            Perform LNPM;
11:        else
12:            if current state is Ingress then
13:                Go to Egress;
14:            else
15:                Send p to next CSw with SwId
```

Fig. 5 shows the ingress pipeline processing logic focusing on *Ipkt* processing. When the *Ipkts* enters the control block after the parser, the prefix shape is built if the packet is normal, and two lookup operations are performed, one in the DPST and another in the CPST. If the DPST lookup is missed, the packet is sent to the control plane if the CPST lookup hits so that a lookup in the s-FIB table can be performed. The packet is dropped if the lookup in both shape tables is missed. It is important to note that, at the end of each successful lookup operation in some f-FIB table, the value of *SwId* is extracted from the CAD and stored in the metadata until the LNPM operation is finished.

When a lookup on the DPST table hits, as tables are distributed across both the ingress and egress pipeline, it is checked if the P4 table in f-FIB associated with the length of the current named component is at ingress. If it does, a match-action operation takes the first named component (or its hash) as a key to perform a table lookup in the corresponding table in f-FIB. Otherwise, it is possible that such a table is in the egress, and the packet proceeds directly to that stage.

During the pipeline traversal, whenever a lookup in any table of the f-FIB succeeds, the conflicting bit in the CAD is checked. If set, it indicates at least one conflicting prefix stored in the HCT table. In this case, a lookup in the HCT is performed, and the result is stored in the current *SwId*. If the lookup in the HCT fails even when the conflicting bit is set, it indicates an error, and the packet is dropped. On the other hand, if the conflicting bit is not set, the value of cad_i.hs is compared with the 10 most significant bits of the recurrence function $F(p, nc_i)$ to verify if the current named component is associated with any existing previous sub-prefix in the f-FIB. This is to avoid the cross-prefix problem, as discussed previously.

Algorithm 3 shows how to handle conflicting components for both ingress and egress processing. Once the *Ipkt* passes the conflicting bit verification and a lookup at HCT hit, the LNPM is executed as shown in Algorithm 4. In the LNPM process, the position of the current named component is compared with the position in the CAD. If the position is different, the LNPM is finished. We can observe from Fig. 5 and Fig. 6 that it is possible to send an *Ipkt* to the core even when the position of the current named component differs from the position in the CAD as long as the *SwId* $\neq$0. Thus, the LNPM might forward *Ipkts* that carries non-canonical names.

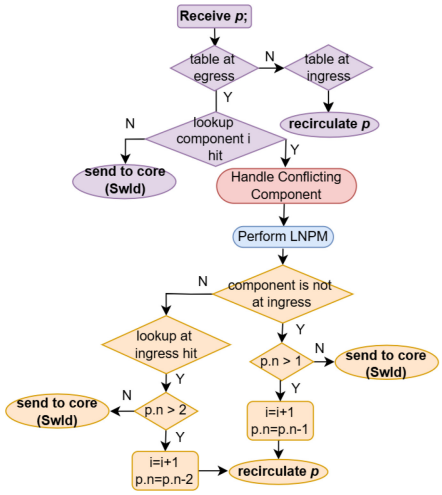

Fig. 6. Egress Pipeline Processing.

Algorithm 4 Perform LNPM

```
1: INPUT: Ipkt (p), cad, position (i), max components
2: if cad.p == i then
3:     if cad.c == 1 then
4:         if cad.e == 0 then
5:             if i < max then
6:                 if current state is Ingress then
7:                     i ← i + 1; Go to Egress;
8:                 else
9:                     Send p to next CSw with SwId
10:            else
11:                Update SwId; Go to line 5;
12:        else
13:            Update SwId; Send p to next CSw with SwId
14: else
15:    if current state is Ingress then
16:        if p is NORMAL then
17:            if lookup at CPST hit then
18:                Send p to the Controller;
19:            else
20:                Drop p;
21:        else
22:            Send p to next CSw with SwId
23:    else
24:        Send p to next CSw with SwId
```

On the other hand, if the position is the same, the continuity bit and the ending prefix bit from the CAD structure are used to decide if the packet should be sent to the core (LNPM is complete) or should be sent to egress control block to continue the LNPM operation.

The workflow shown in Fig. 6 illustrates the egress pipeline logic. As we can see, many operations in this stage are the same as the ones in the ingress. The main difference is what happens to the *Ipkt* once the egress processing terminates, which might include sending the *Ipkt* back to the ingress control block to continue the LNPM. In this regard, the *Ipkts* will be recirculated whenever the lookup in any table of the f-FIB hits, the continuity bit is set, and the corresponding position of the named component in the prefix is less than the *max* value. The *max* value is obtained during the lookup operation in the DPST table that occurs during the first traversal of the *Ipkt* through the pipeline. When the LNPM operation completes, the packet will be forwarded to the core through a random output port carrying the latest *swId* value.

IV. Experimental Evaluation

In this section, we present a comprehensive evaluation of CoFIB data structure. We used real and syntetic name datasets to conduct our analysis. First, we evaluate the CPE algorithm regarding how many canonical prefixes can be extracted from the RIB (Section IV-B). Second, we evaluate the CoFIB data structure analytically and experimentally in terms of memory footprint (Section IV-C) and throughput (Section IV-D), respectively.

In the analytical evaluation, we compare CoFIB against FCTree [15], NDN.p4 [11], and HBM [12] regarding memory consumption. We chose FCTree as a baseline for comparison because it is a recent novel data structure for the NDN FIB used in the ENDN [13] architecture. On the other hand, we decided to include NDN.p4 and HBM in our comparison because, to the best of our knowledge, both solutions are the only hardware-based FIB designs that are fully P4 compatible, focusing on programmable ASIC (see Table I). Recent P4-based NDN implementations such as NDNFab [14], NDNTofino [6], and Pegasus [16], [17] do not implement the FIB in the ASIC but in the control plane software instead.

To evaluate CoFIB experimentally, we instantiated an SDN-based network with CoFIB deployed at one edge switch. The CoFIB source code, including the components of the data and control planes, has been made available in our GitHub repository.[1] Mininet [45] and BMv2 [46] software switches were used to implement the SDN-based network, described in Section IV-D. Experiments were then conducted on this SDN instance to evaluate CoFIB's behavior in terms of throughput and the number of pipeline passes.

A. Datasets

We derived a realistic NDN name dataset, named 0.1K, from traffic traces collected from a worldwide NDN testbed [47]. The NDN traffic traces on which the 0.1K dataset was built were captured over a span of 830 consecutive days, covering the period from April 2023 to July 2025. To balance comprehensiveness and storage efficiency, we sampled a subset of trace files from the repository to generate the 0.1K dataset, following the methodology outlined in Table III. Specifically, we analyzed trace files from 76 equally spaced days between 2023-04-19 and 2025-07-28. This sampling approach aims to preserve temporal coverage while reducing the volume of data required to generate the 0.1K dataset.

Note that, from a total of 1,968,293 distinct NDN names extracted from *Ipkts* in the trace files, we could obtain only 150 distinct routable prefixes. This makes sense considering the limited number of geographical sites (26 different regions) and only 2 applications generating traffic (file transfer and video streaming). Additionally, we observed the applications generate traffic under the same prefixes, with the only difference being in the segment that identifies the required piece of content. These segments are ignored in the LNPM process and, therefore, are not stored in the CoFIB data structure.

[1] https://github.com/castilhorosa/ndn-cofib

TABLE III
Methodology Used to Generate the 0.1K Dataset

Feature	Value
Original dataset capture period	2023-04-19 to 2025-07-28 (830 consecutive days)
Dataset derived by sampling the original dataset.	Traces selected at 11-day intervals (76 days)
Number of trace files analysed	1,047
Total number of packets processed	306,118,381
NDN full names extracted from *Ipkts*	5,488,766
Distinct NDN full names	1,968,293
Number of geographical sites generating traffic	26
Number of applications generating traffic	2
Distinct routable prefixes	150

To increase variability in the datasets and enrich our analysis, we also adapted publicly available domain name datasets, such as the 440K [48] and 10M [49]. The former is the DMOZ dataset obtained with a crawler from the *Open Directory Project* (ODP), and the latter is the DomCop.com project that contains the 10 million most popular websites. Moreover, we synthesized an NDN-friendly dataset, named 2M, using random components extracted from the 440K and 10M datasets. To increase the average size of each prefix, the number of components in the 2M dataset follows the *Weibull* distribution with parameters μ=0.1, α=0.3, β=2. For each dataset, the URLs are converted into NDN-friendly names, and the prefixes with more than eight named components or containing components with more than 31 characters are removed. We have also derived 180K, 1M, and 4M datasets, including the canonical names obtained by the CPE algorithm from the 440K, 2M, and 10M datasets, respectively. Table II summarizes the main characteristics of each dataset used in our study.

As we see in Section III-B6, the tables in f-FIB are placed in both the ingress and egress pipelines in an optimized manner to reduce the number of pipeline passes. Therefore, for a given prefix containing k components and assuming the worst-case scenario, which is when the LNPM needs to perform k matches, there are 2^k different ways that such k components will match depending on where the tables are placed (ingress or egress pipeline). For instance, if we have a prefix with two named components, there are four different ways that component matches might occur in the LNPM: *ingress-ingress*, *ingress-egress*, *egress-ingress*, *egress-egress*. Thus, we have synthesized another dataset, named 0.5K, that contains 2^1 combinations of prefixes with one named component, 2^2 combinations of prefixes with two components, and so on to 2^8 combinations of prefixes with eight named components. The total number of prefixes in the 0.5K dataset is given by the sum $2^1 + 2^2 + \cdots + 2^8$, which results in 510 distinct prefixes.

TABLE II
DATASET CHARACTERISTICS

ID	Description	Total Names	Prefixes After Filtering	Avg. Comp. Length	Avg. No. of Comp.	Max. No. of Comp.
0.1K	Realistic NDN name dataset obtained from traffic traces collected from a worldwide NDN testbed.	1,968,293	150	6.51	3.73	5
0.5K	Random name dataset containing canonical names whose components matches in tables at both ingress and egress pipeline in all possible manners.	510	510	15.90	7.02	8
180K	Canonical name prefixes extracted from the 440K dataset using CPE.	184,684	184,684	7.23	2.08	6
440K	DMOZ dataset retrieved with a crawler from the Open Directory Project (ODP).	447,169	292,079	6.41	2.27	6
1M	Canonical name prefixes extracted from the 2M dataset using CPE	1,216,410	1,216,410	12.00	2.65	8
2M	Random canonical dataset generated by extracting random components from both 440K and 10M	2,000,000	2,000,000	12.00	3.13	8
4M	Canonical name prefixes extracted from the 10M dataset using CPE.	4,407,427	4,407,427	7.65	2.19	6
10M	DomCop.com dataset containing the 10 million most popular websites.	10,000,000	9,981,418	6.71	2.44	8

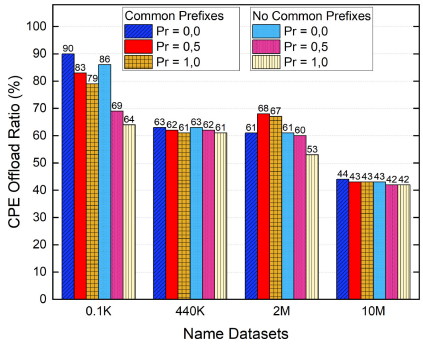

Fig. 7. Percentage of canonical prefixes extracted from RIB using CPE.

B. CPE Performance

The performance of the CPE algorithm is evaluated in terms of the percentage of canonical name prefixes that can be extracted from the RIB. We have inserted the 0.1K, 440K, 2M, and 10M datasets into the RIB to measure how many canonical prefixes can be extracted. For each dataset, we have added, with a probability of *Pr*, all possible sub-prefixes for each prefix. For example, if the dataset contains the prefix */a/b/c* and *Pr*=1, then we also added the sub-prefixes */a/b* and */a* into the dataset. As in NDN architecture, the content name can be constructed iteratively, the FIB stores partial name prefixes, and we use P_r to see the behavior of CPE when the RIB contains both prefixes and their sub-prefixes.

As we can see in Fig. 7, the offload ratio of CPE tends to decrease as the number of name prefixes in each dataset grows. The reason is that larger datasets are more likely to contain named components in different positions than smaller ones. When we compare different P_r values using the datasets 440K and 10M, we do not observe a significant impact on the offload ratio. However, we observe more variations in the offload ratio when using the 0.1K and 2M dataset, especially with higher P_r values. The 0.1K dataset with $P_r = 0$ already contains the prefixes with only one component (only the root), which explains the variations in the offload ratio observed. In addition, the 2M dataset contains named components uniformly extracted from the 440K and 10M datasets. This approach makes the CPE more sensitive to variations on P_r values because it uniformly disperses the randomly named components across many different positions. Conversely, 440K and 10M datasets contain named components only in a few distinct positions, which makes them less sensitive to variations on P_r values.

As described in Section III-A3, to solve the cache-hiding problem, CPE moves shared prefixes from the CoFIB to *s*-FIB. Fig. 7 shows that removing the common prefixes from the canonical set does not impact the offload ratio as much, especially for real-name datasets (440K and 10M).

C. Memory Footprint

We analyze the memory consumption of CoFIB by using the 0.1K, 180K, 1M, and 4M datasets shown in Table II. The theoretical on-chip memory consumption of CoFIB was compared against FCtree, NDN.p4, and HBM. To estimate the memory footprint of each method, we sum up all the bytes necessary to store a given prefix, assuming that TCAM/SRAM memory is used. For the tables in f-FIB that store the ASCII-named components (tables $T_{i\leq 4}$), we assume that an x-byte entry consumes x bytes in SRAM memory. We make this assumption because current programmable ASICs feature an architectural capability that allows sets of small entries to be packed together to reduce memory fragmentation without impairing the match function [7].

The P4 table entries in CoFIB, NDN.p4, and HBM consist of a matching key and the action data. In NDN.p4, each match entry requires 104 bits. However, to be conservative, we consider each entry 96 bits because combining the priority value and the output port into one byte instead of two is possible. The matching key in the HBM method consists of 128 bits, a sequence of eight *crc16* hashes. In CoFIB, as described in Section III-B3, each match key consists of x bytes if the component size is x-byte width and $x \leq 4$, or four bytes if the component size is greater than four (*crc32*

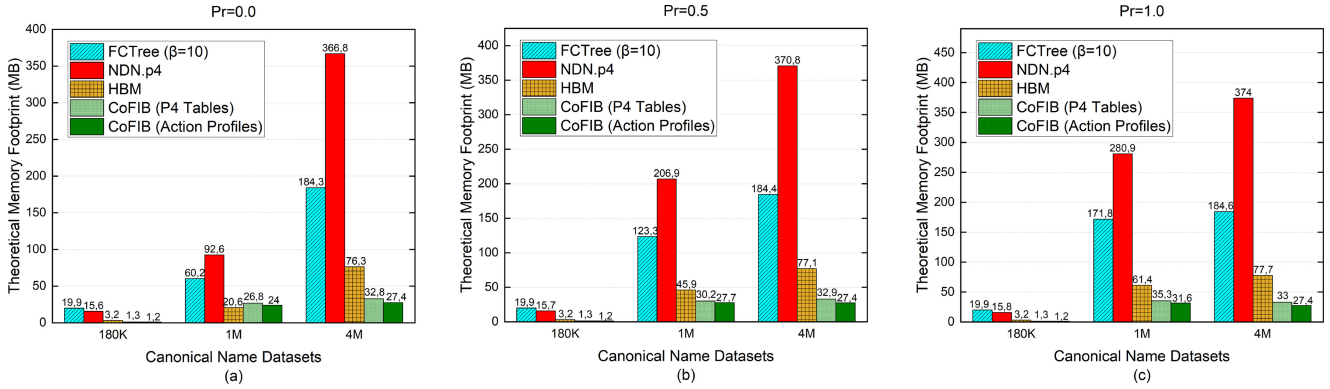

Fig. 8. Memory consumption with different P_r values and datasets.

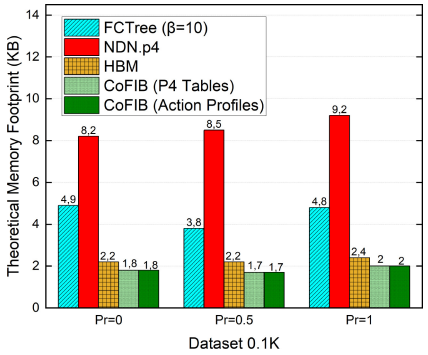

Fig. 9. Memory consumption using dataset 0.1K and different P_r values.

hashes). The action data for NDN.p4 and HBM is an 8-bit value representing the output port associated with the match key. Finally, the action data for each CoFIB entry is a 24-bit value corresponding to the CAD data structure.

Regarding the FCTree, it is worth noting that such a data structure is proposed to run in software. Then, we estimate its memory footprint considering the datasets' characteristics and the software overhead required to store each tree node (e.g., pointers, stack memory, etc.). Unfortunately, no public P4 implementation of FCTree is available so that we can consider its match key bits and action data bits. We have used FCTree with parameter $\beta = 10$, which is the version that provides a better compression rate among all the FCTree family algorithms.

We have used two versions of CoFIB to perform our memory consumption analysis. The first uses conventional P4 tables and the second uses *action profiles*. Both Fig. 8 and Fig. 9 show each method's theoretical on-chip memory consumption, considering datasets 180K, 1M, 4M, and dataset 0.1K, respectively. As expected, we can see that the optimized version of CoFIB outperforms the baseline CoFIB in all possible scenarios. The CoFIB implemented using P4 tables requires less memory than the other solutions. Indeed, we can see that the optimized CoFIB version requires up to 13.64× less memory than NDN.p4, up to 16.58× less memory than FCTree, and up to 2.83× less memory than HBM, considering the datasets 400K and 4M, regardless of the P_r value. When using the 0.1K dataset, we observe that CoFIB consumes from 16% up to 78% less memory than HBM and NDN.p4, respectively. Even with a random canonical dataset (1M), CoFIB provides a better memory compression ratio than all the FIB designs for P_r=1 and P_r=0.5.

When we observe different P_r values applied over the three datasets, we can see the memory consumption of all data structures does not change significantly when the 180K and 4M datasets are used. However, we can see an important increase in memory consumption as we vary the P_r values over the 1M dataset. The reason is that the average number of named components in both 2M and 1M datasets is greater than in the other two datasets, as shown in Table II. Hence, extra sub-prefixes are added to both 2M and 1M datasets in contrast to the other two datasets when using P_r=0.5 and P_r=1.0. As expected, with more prefixes added to the dataset, the memory consumption tends to increase accordingly.

Interestingly, Fig. 8 shows that CoFIB storing the 1M dataset and $P_r = 1.0$ requires more memory than the CoFIB storing the 4M dataset. This is because the 1M canonical dataset, including the extra sub-prefixes due to the parameter $P_r = 1.0$, becomes a dataset with 3.6M prefixes in total. Since the number of prefixes is close to the number of prefixes in the 4M dataset and the average number of prefixes in the 1M dataset is greater than in the 4M dataset, we see a difference in the memory consumption.

There are some reasons why CoFIB outperforms NDN.p4, FCTree, and HBM. Firstly, the CoFIB is designed to store each named component individually instead of using the entire prefix, as in NDN.p4 and HBM. This feature avoids storing FIB entries with common sub-prefixes multiple times. Also, CoFIB's named components with more than four characters are compressed via *crc32* hashes. The components with less than four characters are stored using as few bits as possible. It is important to mention that CoFIB is also optimized with *action profiles* that reduce the memory consumption compared to P4 tables since only one action is invoked when a lookup table hits. Regarding the FCTree, CoFIB does not include pointers' overhead, which adds to memory consumption.

Although the memory consumption in CoFIB exceeds that of HBM in some scenarios, this evaluation has not considered the hash collision effect. As both NDN.p4 and HBM use

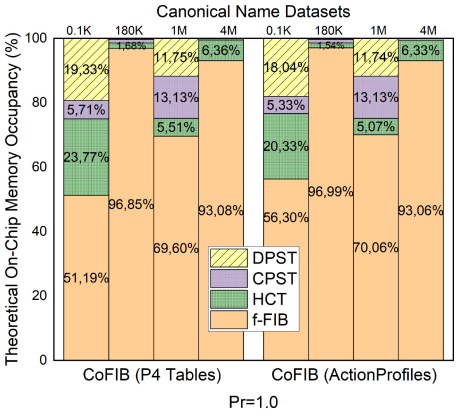

Fig. 10. Memory occupancy.

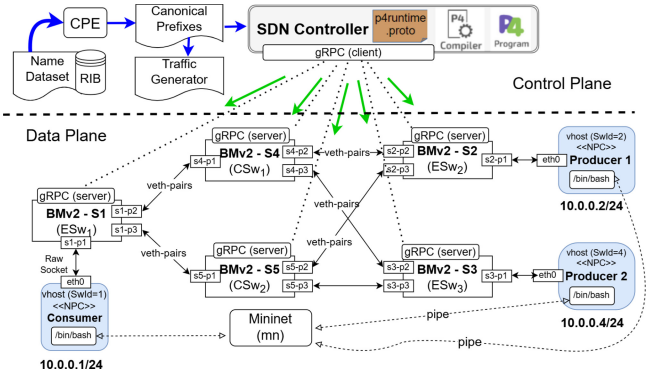

Fig. 11. Evaluation scenario.

crc16 hashes, when storing individual named components in HBM and CoFIB, we expect a much higher collision probability in HBM than in CoFIB for the same amount of components since CoFIB uses *crc32* hashes. To solve the collision problem, techniques such as linear probing, chaining, and rehashing need to be used at the expense of increasing memory consumption. One might argue that CoFIB is also subject to a hash collision. However, using different P4 tables to store named components decreases the collision domain since collided named component hashes are allowed as long as the length of such named components differs. Therefore, we could store all the datasets in CoFIB without any hash collision, which was impossible for NDN.p4 and HBM.

Regarding the on-chip memory occupancy, Fig. 10 shows how the switch memory is allocated among the main components of CoFIB when storing different datasets. Although the f-FIB occupies more than 50% of the on-chip memory in all scenarios, we observe that the overhead introduced by the auxiliary tables (DPST, CPST, HCT) is highest when using the 0.1K dataset. This is due to the characteristics of the name distribution in the 0.1K dataset and the fact that this dataset contains proportionally more conflicting prefixes compared to the other datasets.

Fortunately, Fig. 10 shows that the overhead introduced by the auxiliary tables decreases when the number of prefixes is higher. For example, considering the 180K, 1M, and 4M datasets, we do not see a significant impact on memory consumption when storing DPST, CPST, and HCT. Indeed, in the worst-case scenario (random dataset), less than 30% of memory is occupied by such tables. Observing the 180K and 4M datasets, we note that such tables occupy less than 4% of the total on-chip memory. As discussed previously, the 1M dataset has name prefixes containing more named components, which increases the number of conflicting prefixes and the control plane and data plane shapes. It is worth noting that both NDN.p4 and HBM require mostly scarce and power-hungry TCAM memory to store prefixes, while, in CoFIB, the tables in f-FIB are stored in the more abundant and cheap SRAM memory.

At this point, the analytical evaluation of CoFIB regarding memory consumption leads to two important questions. 1. *Is CoFIB able to store the 4.3 million prefixes from the 4M dataset into the SRAM and TCAM memory available in a real programmable switch?* 2. *Is it possible to store the 4.3 million prefixes from the 4M dataset into the TCAM, considering the FIB designed in both NDN.p4 and HBM?* To answer these questions, we need to compare the amount of SRAM and TCAM memory typically available in modern switch ASICs to the results we obtained analytically.

To help us answer these two questions, let's take the Intel Tofino ASIC series as an example. The total TCAM per pipe in Intel Tofino 2 and Intel Tofino 3 is 10.3 Mb, with 4 pipes in total [50]. Thus, TCAM memory in Tofino 2 is 41.2 Mb, representing 5.15 MB. Such an amount is insufficient to store 4.3 million prefixes in NDN.p4 and HBM, even when we ignore the hash collision problem, since they use only TCAM to store prefixes. Their theoretical memory footprint indicates 374 MB and 77.7 MB, respectively.

As for the SRAM size, we can assume 100 MB based on the table sizes reported in [51]. Therefore, based on our results, CoFIB is the only solution that scales to 4.3 million prefixes, considering a real programmable switch. Indeed, CoFIB can store up to 4.5 million prefixes in SRAM when using a 4M dataset with P_r=1. This requires 25.5 MB of SRAM to store prefixes in f-FIB (25.5% of the total SRAM available on the chip) and 1.9 MB of TCAM to store the tables HCT, CPST, and DPST (36.8% of the total TCAM available on the chip), leaving some free memory for more prefixes.

Additionally, the tables in CoFIB are distributed across both the ingress and egress pipelines, which allows CoFIB to use all the SRAM memory available in the switch. Conversely, in NDN.p4 and HBM, all the P4 tables are placed in the ingress pipeline, wasting resources in the egress pipeline.

D. Throughput

In this section, we conducted several experiments to evaluate the CoFIB's performance in terms of throughput and the number of pipeline passes. We compare CoFIB with HBM only for the throughput since NDN.p4 does not scale to millions of prefixes. We compare CoFIB with NDN.p4 and HBM regarding the number of pipeline passes. Because no P4 implementation of FCTree is publicly available, we omit FCTree in our experiments. We aim to compare CoFIB against

solutions that provide a line rate packet processing when deployed on real programmable switches, which is the case for NDN.p4 and HBM, but not for FCTree.

We carried out four experiments, described in the following sections. The experiments were conducted on a Ubuntu 20.04 VM with 4GB RAM and four vCPUs hosted on Microsoft Windows 11 equipped with an Intel Core i5-1135G7 CPU. We have used a pre-built Vagrant image as shown in [52].

We have created a topology in Mininet [45] (Fig. 11). The switches are implemented in BMv2 [46] and, to reduce the effects of the non-deterministic processing time, all experiments were repeated ten times to get an average throughput.

We store the 0.5K canonical dataset in the CoFIB in ESw_1 in the first three experiments. In experiment 4, we store the 4M canonical dataset into the CoFIB in ESw_1. Therefore, the CPST is empty in all edge switches. Each prefix in both datasets contains a random *SwId* value that could be either 2 or 4, representing ESw_2 or ESw_3, respectively.

All experiments send *Ipkts* from the consumer node to the producers. The packets were sent to the NPC_1 at a constant frequency. To determine the maximum rate we can achieve in our scenario, we measured the lowest inter-packet time for which BMv2 experiences no packet loss when using the HBM as the baseline.

In experiments 1 and 2, we have used *tcpdump* to capture packets in both ESw2 and ESw3 through their incoming virtual interfaces, and we have merged the resulting *pcap* files using *Wireshark* to obtain the throughput. On the other hand, in experiments 3 and 4, we computed the CDF of the number of pipeline passes. Since all experiments presented in this section aim to measure the influence of CoFIB on throughput, we disabled both the CS and PIT in all switches and deployed CoFIB only at the edge switch ESw_1.

The traffic injected on ESw_1 in the first three experiments consists of 160K *Ipkts* crafted in C based on 160K canonical name prefixes extracted from the 0.5K dataset. In experiment 4, the traffic pattern consists of 4.3M packets crafted in C based on canonical prefixes from the 4M dataset. All packets were stored in a binary file in the consumer node. Each *Ipckt* contains one name prefix and an average of 150 bytes related to fields such as selectors, nonce, and guiders. In the first three experiments, the 160K *Ipkts* correspond to eight groups, each containing exactly 20K prefixes with a given number of components. In experiment 4, the 4.3M *Ipkts* are shuffled before being stored in the traffic file. Then, the *Ipkts* are read one by one from the traffic file at a fixed time interval and are sent periodically to ESw_1 through a C raw socket.

1) Experiment 1: Sorted Interest Packets: One feature of CoFIB that might impact the throughput is the number of packet recirculations during the LNPM. That leads us to a question: *What is the throughput behavior when the LNPM operation in CoFIB hits with the maximum number of named components?* To answer this question, we set up an experiment to evaluate the throughput of both HBM and CoFIB in the worst-case scenario. We have deployed CoFIB using the default linear table placement strategy.

The key idea in this experiment is to gradually increase the number of named components in the prefixes that match at CoFIB to observe the impact on CoFIB throughput when processing prefixes with the maximum number of named components. To that end, the traffic file was created with the eight groups sorted in ascending order in the number of components. Thus, the first 20K *Ipkts* are sent to ESw_1 containing prefixes with one named component, followed by 20K *Ipkts* containing prefixes with two named components, and so on up to the last 20K *Ipkts* containing prefixes with eight named components.

The maximum rate we could achieve following this methodology was around 2.5 Mbps, corresponding to an average of 1500 pkts/s. Fig. 12 outlines the throughput obtained in CoFIB and HBM. We can see an increase in bits/sec over time in both solutions, although the throughput in Ipkts/sec seems constant. We can make two observations from this result. First, the increase in bits/sec occurs because packet sizes become greater as we have name prefixes with more components, since the average payload size is the same. Second, this result indicates that throughput does not change significantly when CoFIB processes *Ipkts* whose prefixes contain the maximum number of components (time $\sim$80 seconds onward). The throughput follows a similar trend in both CoFIB and HBM.

One might argue why the throughput in HBM is similar to CoFIB, considering that HBM processes each packet in one pipeline pass. The reason is that although HBM can provide line-rate processing when deployed on a real programmable switch, the parser latency is influenced by the number of named components. Thus, parsing names with more components tends to increase the per-packet latency, which explains our experiment's slight decrease in throughput observed between 80 and 100 seconds.

In Fig. 12, when we contrast the average throughput of both solutions, we note that CoFIB's throughput is 3.86% less than in HBM. This is because CoFIB recirculates *Ipkts* multiple times in the pipeline to perform the LNPM, while in HBM and NDN.p4 the LNPM is executed in one pipeline pass. However, as the CPE algorithm prioritizes shorter prefixes over the longer ones, as shown in Table II, CoFIB stores prefixes containing few named components while the longer ones are stored in *s*-FIB. Thus, the average number of per-packet recirculations tends to be low, which will not significantly impact the CoFIB's overall performance.

2) Experiment 2: Unsorted Interest Packets: In real deployments, we expect *Ipkts* from the edge switches containing prefixes with an arbitrary number of components. As such, this experiment aims to evaluate the CoFIB's throughput in the average case (the number of components is not sorted in the *Ipkts*). Consequently, this experiment aims to answer the question: *How does the CoFIB's throughput behave in contrast to HBM in a scenario where the Ipkts contain prefixes with an arbitrary number of components*?

The maximum rate we could obtain in this experiment was around 2.6 Mbps, corresponding to an average of 1600 ptks/s. In Fig. 13 we note that the throughput in bits/sec and Ipkts/sec does not change over time in CoFIB and HBM. The throughput in bits/sec over time behaves as expected since the average packet size does not increase during the simulation. When comparing the average throughput, the CoFIB's performance

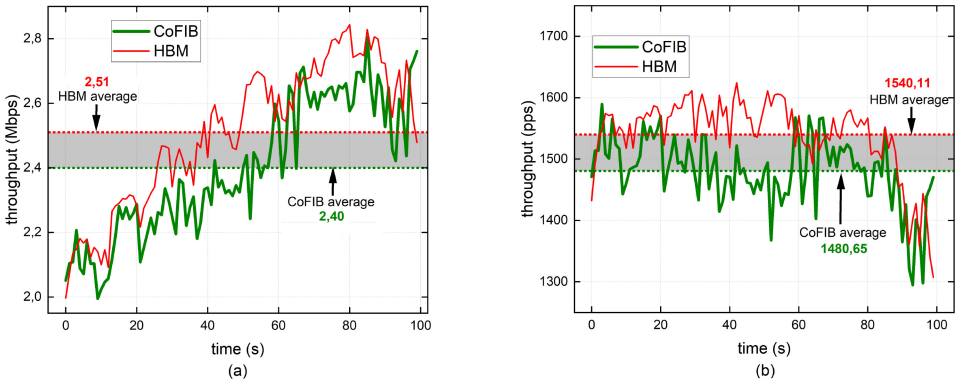

Fig. 12. Throughput in Mbps (a) and in packets per second (b) when sending *Ipkts* sorted by the number of components.

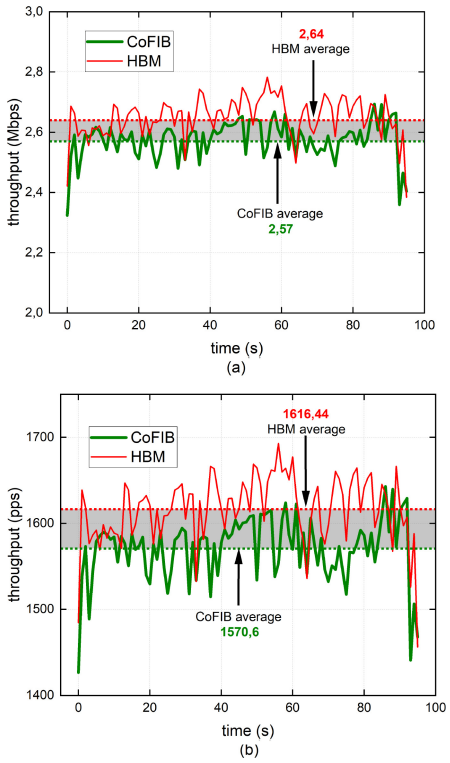

Fig. 13. Throughput when sending Ipkts randomly by the number of components.

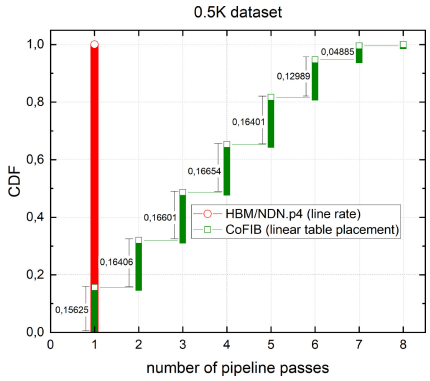

Fig. 14. CDF of the number of pipeline passes using the 0.5K dataset.

is around 2.83% less than observed in HBM. This throughput degradation is also due to packet recirculations, as in the previous experiment. However, this is better than we obtained in experiment 1, which represents 3.86% of degradation.

As the inter-packet time is constant and is set to a value sufficient to avoid packet losses, the queue at the incoming interface in ESw_1 can absorb both the new traffic and the extra traffic due to the recirculated packets. This is possible because the *Ipkts* containing prefixes with only a few named components are equally distributed over time. Moreover, since there are no huge sequences of packets that recirculate several times, the queue occupancy is less than in the previous experiment, which explains the better throughput.

It is important to mention that BMv2 recirculates packets through the same incoming port as they arrive. Such a design compromises throughput since it reduces the possibility of parallelism. On the other hand, in real programmable ASICs, it is possible to dedicate ports for recirculation to increase throughput.

We also need to consider that CoFIB is deployed only at the edge, with *Ipkts* being forwarded at the line rate in the core. On the other hand, as HBM and NDN.p4 are deployed in all switches, although they provide line-rate LNPM, parsing the TLV headers throughout the network adds some per-packet latency, compromising the throughput in contrast to CoFIB.

Overall, the results obtained from experiments 1 and 2 show that our interactive approach to perform the LNPM through packet recirculations in software does not considerably degrade end-to-end throughput. Even in the worst-case scenario, where the ESw_1 receives *Ipkts* containing prefixes that match in f-FIB using eight named components, the possibility of some packets recirculating 7 times, passing 8 times in the pipeline, did not affect the throughput significantly.

3) Experiment 3: Pipeline Passes (0.5K Dataset): Despite all the optimizations made in our simulated environment, it is known that BMv2 is not meant to be a production software switch. Besides that, several factors might impact throughput in a software switch, such as the number of table entries and CPU cores, the experimentation platform (Linux VM or a bare-metal Linux machine), and so on. Therefore, experiments 3 and 4 aim to evaluate CoFIB from another perspective, ignoring the effects of nondeterministic software processing.

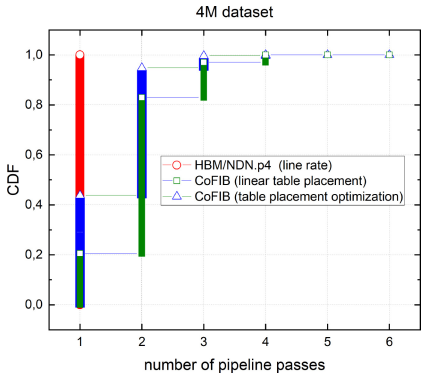

Fig. 15. CDF of the number of pipeline passes using the 4M dataset.

In this way, the main goal of experiments 3 and 4 is to evaluate CoFIB in terms of how many pipeline passes each *Ipkt* experiences during the LNPM. This approach allows us to infer throughput behavior more accurately when deploying CoFIB in a real programmable switch, since the main factor that might impact throughput in such devices is the number of packet recirculations.

In experiment 3, we sent to ESw_1 a total of 160K *Ipkts* in an arbitrary order, exactly as in experiment 2. The only difference is that here, we compute the number of pipeline passes for each packet during the LNPM. Fig. 14 presents the CDF of the number of pipeline passes when CoFIB stores the 0.5K dataset and processes the 160K packets from the traffic file. Unsurprisingly, NDN.p4 and HBM processed 100% of packets in one pipeline pass. Conversely, the number of pipeline passes in CoFIB ranges from 1 to 8 times. Once discrete values give the number of pipeline passes, the step-like CDF graph in Fig. 14 is an approximation of a discrete uniform distribution, as expected. This is because the traffic file has the same amount of packets (20K) for each possible number of components.

Interestingly, the height of each step in the graph varies as the number of pipeline passes increases, which is not supposed to happen in a step-like graph based on a discrete uniform distribution. However, such behavior is expected, as packets that passed x times in the pipeline do not necessarily contain prefixes with exactly x components. More precisely, packets that crossed the pipeline x times only might contain y components, where $x \leq y \leq 2x$ if $x \leq 4$ or $x \leq y \leq 8$ otherwise. This is because CoFIB supports prefixes with up to 8 named components and enables two component match-action operations per pipeline pass, depending on where the tables are placed.

The possibility of performing two match-action operations in one single pipeline pass is beneficial for longer prefixes. For example, although the traffic file has 20K *Ipkts* with prefixes containing eight components, only a fraction of these packets experienced exactly eight pipeline passes. This represents the worst-case scenario where only 1 component match-action occurs per pipeline pass. The CDF graph in Fig. 14 indicates that 99.561% of the packets experienced at most seven pipeline passes, which results in only 0.439% packets (lowest height among the eight steps in the graph) that experienced exactly eight pipeline passes (100% - 99.561%). This represents only 702 out of the 20K packets sent.

On the other hand, when we look at the highest height among the 8 steps in the CDF graph, we notice that this value is 0.16654 and corresponds to the number of packets that crossed the pipeline four times. Indeed, packets crossing the pipeline four times include packets containing prefixes with four components and some containing prefixes with five, six, seven, and eight components. That means the number of recirculations is reduced for some groups of packets, contributing positively to better throughput.

4) Experiment 4: Pipeline Passes (4M Dataset): Experiment 4 also aims to evaluate CoFIB in terms of pipeline passes. However, it includes the table placement optimization strategy and the 4M dataset. Thus, the traffic file injected on ESw_1 consists of the 4.3M *Ipkts* in a random order containing the prefixes from the 4M dataset with P_r=1.0.

This experiment considers the worst-case scenario. The worst-case scenario here means that all packets hitting the CoFIB will perform the match-action operations for all their components. In other words, neither CPST nor DPST reduces the number of match-action operations. Of course, in the average-case scenario, we expect that some packets will be processed at a line rate when the shapes of their prefixes do not exist in CPST and/or DPST.

In summary, we try to answer two questions through this experiment. First, *what is the reduction in the number of pipeline passes when we use the table placement optimization strategy*? Second, *at what degree the throughput might be reduced due to the packet recirculations when CoFIB stores 4.3 million prefixes in a real programmable switch*?

Fig. 15 shows the CDF of the number of pipeline passes when CoFIB stores the 4M dataset and processes the 4.3M packets from the traffic file. In contrast to the results obtained in experiment 3, we can observe the CDF graph does not approximate a discrete uniform distribution. The reason is that the average number of components in the 4M dataset, as shown in Table II, is less than in the 0.5K dataset. Moreover, as mentioned before, the CPE algorithm prioritizes shorter prefixes. This explains why the average number of pipeline passes is less than in the previous experiment.

Surprisingly, we can see that the percentage of *Ipkts* processed at line rate increases from 20.57% to 43.74% when using the table placement optimization strategy. For *Ipkts* that crossed the pipeline 2 times, the percentage increased from 82.95% to 94.77%. We can observe the same trend for packets that experienced more pipeline passes. For instance, from 96.98% to 99.65% for packets experiencing three pipeline passes and from 99.98% to 100% for packets experiencing four pipeline passes. This is significant because CoFIB can provide a throughput closer to the line rate when reducing the number of recirculations per packet.

Unfortunately, we can not fully answer our second question as we could not deploy CoFIB on a real programmable switch. However, we can provide some insights about how throughput would be affected based on the results we obtained in this experiment and some observations that can be made.

Firstly, a recent and comprehensive latency profiling study of the Tofino programmable ASIC presented in [53] suggests an average per-packet latency of 385.8*ns* when using packets with similar sizes as in our experiments. This value is

obtained when configuring the switch ports to 100G speed and considering several use case scenarios similar to what CoFIB would experience in a physical deployment. Thus, based on the results obtained in this section, we can say the latency for most *Ipkts* is bound to 771.7*ns* considering our scenario, which gives us some confidence to hypothesize that latency in CoFIB can be within the nanosecond time scale.

Secondly, it is known that the search time in TCAM is 2.7*ns* for one lookup, while in SRAM, each access time is 0.47*ns* [21]. Although CoFIB requires multiple accesses on SRAM to perform the LNPM, in our worst-case scenario, 94.77% of all *Ipckts* will have the LNPM latency bound to 1.88*ns* (0.47×2 for table lookup and 0.47×2 for extracting the ports). Therefore, if we ignore the effects of multiple header parsing and consider only LNPM latency, CoFIB outperforms NDN.p4 and HBM regarding per-packet latency since they are TCAM-based solutions with the FIB deployed in all switches.

Furthermore, to minimize the effects of packet recirculations, since CoFIB is designed to use eight output ports, we can dedicate 50% of the switch ports for loopback to gain $8\times$ more throughput, similarly to [54]. Also, because each component appears at only one position in the prefix and duplicated named components in the same prefix are not allowed when recirculating and receiving *Ipkts* at the same time in real programmable ASICs, it is possible to perform parallel table lookups since the switch has dedicated ports for recirculation.

As a consequence of the results presented in this section and the observations above, it is reasonable to suppose that, when running on a real programmable switch ASIC, the per-packet processing latency in CoFIB will remain within the nanosecond time scale given the small number of pipeline passes experienced for most packets. Thus, these observations suggest that recirculating some of the *Ipkts* will not impact the CoFIB scalability for a large-scale name dataset.

Moreover, since the proposed LNPM algorithm is designed to forward packets not to the output ports but to the edge programmable switches, the forwarding strategy is not supported by the CoFIB. However, it is possible to use P4 registers to store the port rankings required by the forwarding strategy and send the packets to the core according to such ranks instead of selecting random output ports.

V. Concluding Remarks and Future Works

This work proposes CoFIB, an NDN FIB data structure designed to run on edge programmable switches in an SDN-based network. Instead of relying only on scarce TCAM memory, CoFIB explores the SRAM memory available in both ingress and egress pipelines. In the data plane, we propose an LNPM algorithm and a table placement optimization strategy to reduce the number of packet recirculations in the pipeline. Regarding the control plane, we design data structures to produce the P4 table entries and to extract canonical prefixes from the *Routing Information Base* (RIB). Experimental and analytical results show that CoFIB scales to millions of named prefixes stored in the data plane, reducing the memory consumption up to $16.58\times$ in contrast to other FIB solutions in literature while keeping the average packet processing time on a nanosecond time scale.

Our primary goal in future work is to deploy CoFIB into a Tofino-based programmable switch to take advantage of hardware parallelism. Because f-FIB is distributed across both ingress and egress control units, the possibility of having two match-action operations per pipeline pass will scale the CoFIB to a *Tbps* data rate. Additionally, we aim to improve the LNPM algorithm by adopting the speculative match-action operation strategy to speed up the throughput without compromising memory consumption.

Regarding on-chip memory consumption, we intend to use some name prefix aggregation techniques to increase the number of name prefixes stored in the SRAM. Additionally, although the elements in the control plane is not a big concern since the memory capacity in DRAM is orders of magnitude larger than ASIC memory, in future works, we also aim to evaluate the impact of s-FIB in terms of memory footprint and throughput.

Acknowledgment

This work has been partly developed in the scope of the project EXIGENCE. EXIGENCE has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101139120. The authors acknowledge the Coordination for the Improvement of Higher Education Personnel - CAPES (ROR identifier: 00x0ma614) covered the Article Processing Charge for the publication of this research. For open access purposes, the authors have assigned the Creative Commons CC BY license to any accepted version of the article. They also thank the Goiano Federal Institute (IFGoiano), and Fundação para a Ciência e Tecnologia within the R&D Unit Project Scope UID/00319/Centro ALGORITMI (ALGORITMI/UM).

References

[1] L. Zhang et al., "Named data networking," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656887

[2] E. Newberry, X. Ma, and L. Zhang, "YaNFD: Yet another named data networking forwarding daemon," in *Proc. 8th ACM Conf. Inf. Centric Netw.*, New York, NY, USA, 2021, pp. 30–41.

[3] Z. Li, Y. Xu, B. Zhang, L. Yan, and K. Liu, "Packet forwarding in named data networking requirements and survey of solutions," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1950–1987, 2nd Quart., 2019.

[4] E. C. Rosa and F. de Oliveira Silva, "A Review on Recent NDN FIB Implementations for High-Speed Switches," in *Advanced Information Networking and Applications*, L. Barolli, F. Hussain, and T. Enokido, Eds. Cham, Switzerland: Springer Int., 2022, pp. 288–300.

[5] J. Shi, D. Pesavento, and L. Benmohamed, "NDN-DPDK: NDN forwarding at 100 Gbps on commodity hardware," in *Proc. 7th ACM Conf. Inf. Centric Netw.*, 2020, pp. 30–40.

[6] J. Takemasa, Y. Koizumi, and T. Hasegawa, *Vision: Toward 10 Tbps NDN Forwarding with Billion Prefixes by Programmable Switches*. New York, NY, USA: ACM, 2021, pp. 13–19. [Online]. Available: https://doi.org/10.1145/3460417.3482973

[7] P. Bosshart et al., "Forwarding metamorphosis fast programmable match-action processing in hardware for SDN," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2013, pp. 99–110. [Online]. Available: https://doi.org/10.1145/2486001.2486011

[8] S. Chole et al., "DRMT: Disaggregated programmable switching," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 1–14.

[9] W. Quan, C. Xu, J. Guan, H. Zhang, and L. A. Grieco, "Scalable name lookup with adaptive prefix bloom filter for named data networking," *IEEE Wireless Commun. Lett.*, vol. 18, no. 1, pp. 102–105, Jan. 2014.

[10] Z. Li, Y. Xu, K. Liu, X. Wang, and D. Liu, "5G with B-MaFIB based named data networking," *IEEE Access*, vol. 6, pp. 30501–30507, 2018.

[11] S. Signorello, R. State, J. François, and O. Festor, "NDN.p4: Programming information-centric data-planes," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, 2016, pp. 384–389.

[12] R. Miguel, S. Signorello, and F. M. V. Ramos, "Named data networking with programmable switches," in *Proc. IEEE 26th Int. Conf. Netw. Protocols (ICNP)*, 2018, pp. 400–405.

[13] O. Karrakchou, N. Samaan, and A. Karmouch, "ENDN: An enhanced NDN architecture with a P4-programmabie data plane," in *Proc. 7th ACM Conf. Inf.-Centric Netw.*, 2020, pp. 1–11.

[14] A. L. R. Madureira, F. R. C. Araújo, G. B. Araújo, and L. N. Sampaio, "NDN fabric: Where the software-defined networking meets the content-centric model," *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 374–387, Mar. 2021.

[15] O. Karrakchou, N. Samaan, and A. Karmouch, "FCTrees: A front-coded family of compressed tree-based FIB structures for NDN routers," *IEEE Trans. Netw. Service Manag.*, vol. 17, no. 2, pp. 1167–1180, Jun. 2020.

[16] X. Long, K. Huang, R. Yang, Q. Dai, and Z. Li, "Pegasus: A high-speed NDN router with programmable switches and server clusters," in *Proc. 10th ACM Conf. Inf.-Centric Netw.*, New York, NY, USA, 2023, pp. 12–18.

[17] X. Long, K. Huang, R. Yang, Q. Dai, and Z. Li, "Pegasus: A practical high-speed cross-platform NDN forwarder," *Comput. Netw.*, vol. 269, Sep. 2025, Art. no. 111474.

[18] Y. Wang et al., "Scalable name lookup in NDN using effective name component encoding," in *Proc. IEEE 32nd Int. Conf. Distrib. Comput. Syst.*, 2012, pp. 688–697.

[19] H. Dai and B. Liu, "CONSERT: Constructing optimal name-based routing tables," *Comput. Netw.*, vol. 94, pp. 62–79, Jan. 2016.

[20] S. H. Bouk, S. H. Ahmed, and D. Kim, "Hierarchical and hash based naming with compact trie name management scheme for vehicular content centric networks," *Comput. Commun.*, vol. 71, pp. 73–83, Nov. 2015.

[21] T. Song, H. Yuan, P. Crowley, and B. Zhang, "Scalable name-based packet forwarding: From millions to billions," in *Proc. 2nd ACM Conf. Inf.-Centric Netw.*, 2015, pp. 19–28.

[22] D. Saxena, S. Mahar, V. Raychoudhury, and J. Cao, "Scalable, high-speed on-chip-based NDN name forwarding using FPGA," in *Proc. 20th Int. Conf. Distrib. Comput. Netw.*, 2019, pp. 81–89.

[23] H. Wang et al., "P4FPGA: A rapid prototyping framework for P4," in *Proc. Symp. SDN Res.*, Santa Clara, CA, USA, Apr. 2017, pp. 122–135.

[24] A. Afanasyev and J. Shi, "NFD overview—Named data networking forwarding Daemon (NFD) 0.6.6-24-g1402fa1 documentation," 2019. [Online]. Available: http://named-data.net/doc/NFD/current/overview.html

[25] T. Song, T. Li, and Y. Yang, "PtCAM: Scalable high-speed name prefix lookup using TCAM," in *Proc. ACM SIGCOMM Conf.*, 2025, pp. 707–719.

[26] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue, "Caesar: A content router for high-speed forwarding on content names," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2014, pp. 137–148.

[27] F. Li, F. Chen, J. Wu, and H. Xie, "Longest prefix lookup in named data networking: How fast can it be?" in *Proc. 9th IEEE Int. Conf. Netw. Archit. Storage*, 2014, pp. 186–190.

[28] Z. Li, K. Liu, D. Liu, H. Shi, and Y. Chen, "Hybrid wireless networks with FIB-based named data networking," *EURASIP J. Wireless Commun. Netw.*, vol. 2017, no. 1, p. 54, Mar. 2017.

[29] K. Huang and Z. Wang, "A hybrid approach to scalable name prefix lookup," in *Proc. IEEE/ACM 26th Int. Symp. Qual. Service (IWQoS)*, 2018, pp. 1–10.

[30] W. Yu and D. Pao, "Hardware accelerator for FIB lookup in named data networking," *Microprocess. Microsyst.*, vol. 71, Nov. 2019, Art. no. 102877.

[31] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: Fast and DoS-resistant forwarding with hash tables," in *Proc. Archit. Netw. Commun. Syst.*, 2013, pp. 215–225.

[32] A. Ooka and H. Asaeda, "CCNX router on FPGA accelerator achieving predictable performance," in *Proc. 10th ACM Conf. Inf.-Centric Netw.*, 2023, pp. 1–11.

[33] Z. Li, K. Liu, Y. Zhao, and Y. Ma, "MaPIT: An enhanced pending interest table for NDN with mapping bloom filter," *IEEE Commun. Lett.*, vol. 18, no. 11, pp. 1915–1918, Nov. 2014.

[34] M. Mosko, I. Solis, and C. A. Wood, "Content-centric networking (CCNx) semantics," RFC 8569, IETF, Jul. 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8569

[35] M. Mosko, I. Solis, and C. A. Wood, "Content-centric networking (CCNx) messages in TLV format," RFC 8609, IETF, Jul. 2019. [Online]. Available: https://www.rfc-editor.org/info/rfc8609

[36] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "NLSR: Named-data link state routing protocol," in *Proc. 3rd ACM SIGCOMM Workshop Inf.-Centric Netw.*, 2013, pp. 15–20.

[37] P. F. Tehrani, E. Osterweil, T. C. Schmidt, and M. Wählisch, "SoK: Public key and namespace management in NDN," in *Proc. 9th ACM Conf. Inf.-Centric Netw.*, 2022, pp. 67–79.

[38] H. Khelifi et al., "Named data networking in vehicular ad hoc networks: State-of-the-art and challenges," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 1, pp. 320–351, 1st Quart., 2020.

[39] A. Tariq, R. A. Rehman, and B.-S. Kim, "Forwarding strategies in NDN-based wireless networks: A survey," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 1, pp. 68–95, 1st Quart., 2020.

[40] X. Chen, G. Zhang, and H. Cui, "Investigating route cache in named data networking," *IEEE Commun. Lett.*, vol. 22, no. 2, pp. 296–299, Feb. 2018.

[41] E. C. Rosa and F. D. O. Silva, "A hash-free method for FIB and LNPM in ICN programmable data planes," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, 2022, pp. 186–191.

[42] P. V. Mockapetris, "Domain names-implementation and specification," RFC 1035, IETF, 1987.

[43] *P416 Language Specification*, P4 Language Consortium, Palo Alto, CA, USA, May 2023. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.4.html

[44] C. Yi, J. Abraham, A. Afanasyev, L. Wang, B. Zhang, and L. Zhang, "On the role of routing in named data networking," in *Proc. 1st ACM Conf. Inf.-Centric Netw.*, 2014, pp. 27–36.

[45] "Mininet: An instant virtual network on your laptop (or other PC)—Mininet," Accessed: Mar. 11, 2024. [Online]. Available: https://mininet.org/

[46] A. Bas, A. Fingerhut, and A. Sivaraman, "The behavioral model," May 2024. [Online]. Available: https://github.com/p4lang/behavioral-model

[47] S. Timilsina, D. Pesavento, J. Shi, S. Shannigrahi, and L. Benmohamed, "Capture and analysis of traffic traces on a wide-area NDN testbed," in *Proc. 10th ACM Conf. Inf.-Centric Netw.*, 2023, pp. 101–108.

[48] C. Lorenzetti, A. Maguitman, and C. Baggio, "DMOZ 2006 dataset and its wikification." Mendeley Data. May 2019. [Online]. Available: https://data.mendeley.com/datasets/9mpgz8z257/1

[49] "Top 10 million Websites." DomCop.com. May 2021. [Online]. Available: https://www.domcop.com/top-10-million-websites

[50] "What is Intel Tofino? An easy guide to its benefits and use cases." whiteboxsolution.com., 2012. Accessed: Mar. 11, 2024. [Online]. Available: https://www.whiteboxsolution.com/blog/what-is-intel-tofino-an-easy-guide-to-its-benefits-and-use-cases/

[51] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 15–28.

[52] "P4 Developer Day 2019—Open networking foundation," 2019. Accessed: Mar. 11, 2024. [Online]. Available: https://opennetworking.org/p4-events/p4-developer-day-2019/

[53] D. Franco et al., "A comprehensive latency profiling study of the Tofino p4 programmable ASIC-based hardware," *Comput. Commun.*, vol. 218, pp. 14–30, Mar. 2024.

[54] X. Chen, "Implementing AES encryption on programmable switches via scrambled lookup tables," in *Proc. Workshop Secure Programmable Netw. Infrastruct.*, New York, NY, USA, 2020, pp. 8–14.

Eduardo Castilho Rosa received the master's degree in electrical engineering, in 2011, where he worked on mechanisms to provide quality-of-service for wireless broadband networks, and the Ph.D. degree in computer science from the Federal University of Uberlândia in 2024, working on future Internet architectures and software-defined networking. He is a Professor with the Department of Information Systems, Goiano Federal Institute (IF Goiano), Brazil. His research interests include software-defined networking, information-centric networking, and programmable data planes.

Daniel Nunes Corujo received the Ph.D. degree in communication models for the Future Mobile Internet from the University of Aveiro, in 2013, where he is an Associate Professor with the Departamento de Eletrónica e Telecomunicações. He was with Telecommunication Management Software, Nokia Siemens Networks and as an IMS Deployment Executive for the research branch of Portugal Telecom. He was the Coordinator of the Advanced Telecommunications and Networks Group with the Instituto de Telecomunicações in Aveiro, Portugal, where he worked on several EU IST projects. He is also the Project Manager of several software projects with IT-Aveiro, such as ODTONE and OPMIP, and contributes to the IEEE 802.21 and IRTF's ICNRG standardization workgroups. His current research areas in mobility mechanisms for heterogeneous networks and the future Internet.

Flávio de Oliveira Silva (Senior Member, IEEE) received the Ph.D. degree from the University of São Paulo in 2013. He is a Professor with the Department of Informatics at the School of Engineering, University of Minho, Braga, Portugal, and a Researcher with the ALGORITMI Centre. He has published and presented several papers at conferences worldwide. His research interests include future networks, the IoT, network softwarization, such as SDN and NFV, future intelligent applications and systems, cloud computing, and software-based innovation. He is a member of ACM and SBC. He is a Reviewer of several journals and a member of the TPC at several IEEE conferences.

Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - ROR identifier: 00x0ma614