Named Data Networking with Programmable Switches

Rui Miguel
LASIGE, Faculdade de Ciências
Universidade de Lisboa
Lisbon, Portugal
rmiguel@lasige.di.fc.ul.pt

Salvatore Signorello
SnT
University of Luxembourg
Luxembourg, Luxembourg
salvatore.signorello@uni.lu

Fernando M. V. Ramos
LASIGE, Faculdade de Ciências
Universidade of Lisboa
Lisbon, Portugal
fvramos@ciencias.ulisboa.pt

***Abstract*—The Internet today is mainly used for distributing content, in a fundamental departure from its original goal of enabling communication between endpoints. As a response to this change, Named Data Networking (NDN) is a new architecture rooted on the concept of naming *data*, in contrast to the original paradigm based on naming hosts. This radical architectural shift results in packet processing in NDN to differ substantially from IP. As a consequence, current network equipment cannot be seamlessly extended to offer NDN data-plane functions. To address this challenge, available NDN router solutions are usually software-based, and even the highly-optimised designs tailored to specific hardware platforms present limited performance, hindering adoption. In addition, these tailor-made solutions are hardly reusable in research and production networks. The emergence of programmable switching chips and of languages to program them, like P4, brings hope for the state of affairs to change. In this paper, we present the design of an NDN router written in P4. We improve over the state-of-the-art solution by extending the NDN functionality, and by addressing its scalability limitations. A preliminary evaluation of our open-source solution running on a software target demonstrates its feasibility.**

***Index Terms*—Programmable data-planes, Information-Centric Networking, Named-Data Networking, P4.**

I. Introduction

The Internet was originally designed to connect and route traffic between specific end-points over different packet switching networks. Today, the Internet's original architectural design serves a massive unstructured distribution of diverse content, such as hypertext, images, and videos. Yet, the Internet architecture is better-suited for the earlier use-case. By consequence, developers struggle to build efficient distributed systems on top of a network stack that is mostly connection-oriented. Information-Centric Networking (ICN) [1] approaches (e.g., Content-Centric Networking [2], or Named Data Networking [3]), propose to solve this problem with clean-slate architectures tailored for content distribution.

The Named Data Networking (NDN) architecture features a network-layer based on named data. The forwarding of named data in NDN requires a substantially different packet processing logic from the one available on today's IP-oriented network equipment. Therefore, the development of an NDN router faces two main problems. First, the algorithmic challenges of the complex packet processing logic to be performed at line-speed. Second, the lack of adequate platforms to implement the algorithmic solutions. Most existing implementations are software-based, limiting their performance. State-of-the-art solutions (detailed in Section II) achieve throughputs in the order of the tens of millions of packets per second, which is orders of magnitude slower than a hardware router. In addition, these highly-optimised solutions target specific systems, making them difficult to port. We argue that the lack of an NDN router with performance equivalent to its IP counterpart and the lack of portability of existing solutions have been two of the fundamental barriers for the wide-spread adoption of NDN.

The emergence of fully programmable switches [4]–[6], along with the availability of high-level programming languages such as P4 [7], now enable researchers and practitioners to investigate and deploy new packet forwarding architectures and network protocols. This (r)evolution in merchant switching hardware makes it possible to overcome the difficulties of building a high-performance NDN router. These switching chips are able to process several billion packets per second [5], [6], which is orders of magnitude higher throughput than existing NDN solutions are capable of. The use of the P4 language and the increasing number of compilers available facilitates portability. The same P4 program can be compiled to different targets, making it possible to run the same solution, with minor to no modification, in different software and hardware switching systems.

In this paper, we leverage the advances in programmable networking hardware and present the design and implementation of an NDN router in P4. Our design is driven both by the empirical analysis of ICN name-sets and by the observation of the fast-memory limitations of commodity switches. We are not the first to write a P4 program for NDN. The first attempt was made by Signorello et al. [8]. We improve over this solution by considering all NDN functionality (including the Content Store, not considered in [8]), and by addressing the scalability issues of the previous FIB design. We also provide a performance evaluation on a software target. The results demonstrate the feasibility of our design and highlight the overheads, shedding light on some of the optimisations to be devised as part of future work.

The rest of the document is organized as follows. We provide the reader with the necessary background and describe the related work in §II. Next, we outline the main challenges

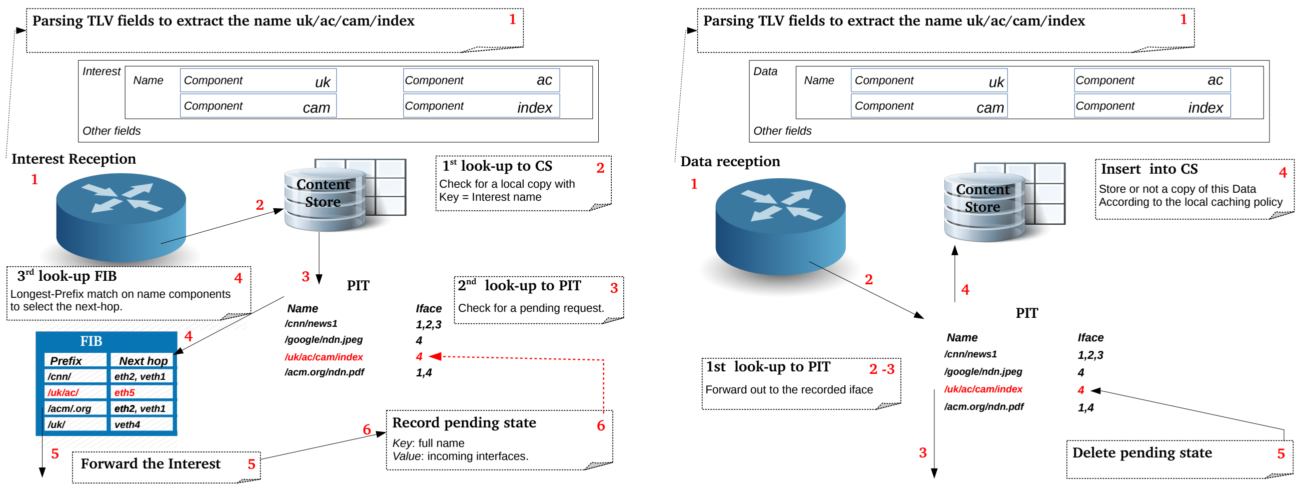


Fig. 1. Illustration of the packet processing operations performed at Interest (1a) and at Data reception (1b).

to describe the NDN forwarding plane in P4, and detail the design and implementation of our solution in §III. Finally, we present preliminary results in §IV, and conclude with our plans for future work in §V.

II. Background & Previous Work

In NDN, hosts and routers exchange two kinds of named packet: Interest and Data. An Interest carries a content request, while a Data carries the content itself and a signature bounding the content to the name. Several uniquely-named packet-sized content chunks are used to transfer contents which do not fit into a single Data packet. Interests are forwarded upstream by name to one or several content sources and leaves trail in traversed routers. Those traces are then followed and consumed by potential Data traveling downstream.

Naming. Interests and Data carry hierarchical, human-readable, variable-length names. The hierarchy in NDN names enables i) the naming schema to be scaled to large networks by routing on aggregated prefixes, ii) applications to define different name structures which, however, stay opaque to the network layer. An NDN name, e.g., 'uk/ac/cam/index', is made of several substrings delimited by '/' and called (name) components. The first part of a name, 'uk/ac', is referred to as the *prefix*. Prefixes are used to forward Interests by a name-based routing plane. The second part of the name, 'cam/index', uniquely identifies a resource under a certain name.

Data-plane components and operations. An NDN router is equipped with the following main functional blocks. The *Forwarding Information Base (FIB)* table maps variable-length names to one or many outgoing interfaces. A FIB lookup is done every time an Interest needs to be forwarded. FIB entries are inserted by name-based routing protocols running in the control-plane. The *Pending Interest Table (PIT)* records information about forwarded Interests. PIT entries are created by and consumed by Interests and Data, respectively. An entry is removed from the PIT if no Data has consumed it after a certain expiration time. The *Content Store (CS)* is the router local cache, which may opportunistically store Data to serve further Interests for the same content.

The forwarding behavior of an NDN router is illustrated through the example in Fig. 1. At packet reception, of either an Interest or a Data, the first step is to extract the name. Afterwards, when it is an Interest (Fig. 1a) the router checks the Content Store for the named content. If a local copy is available, that Data is forwarded downstream. Otherwise, the router looks-up the PIT to find any pending Interest for the same content. If a PIT entry exists, that entry is updated, and the Interest is dropped. If not, a PIT entry is created, and the router looks-up the FIB to determine the next-hop for the received Interest.

When a Data packet is received (Fig. 1b), the router looks-up the PIT to check whether the Data is solicited or not. If no PIT entry exists for this name, the Data is considered unsolicited and so is dropped. Otherwise, if a PIT entry exists, the packet is multicast to all the listed interfaces. The router may also decide to cache a copy of the Data, depending on its caching policy.

The typical way to deal with unanswered Interests (e.g., due to entry expiration, a full PIT, or packet loss), consists in consumers re-issuing the request. More sophisticated forwarding strategies have also been proposed to deal with this issue [9].

A. Related Work

Most of the related work on the design of NDN routers addresses the algorithmic challenges of a specific functional block of the data-plane. Only a few provide a complete architectural design, which is the focus of our short survey.

Software designs generally leverage the state-of-the-art algorithms and data-structures for NDN packet forwarding which better exploit the capabilities of the underlying general purpose hardware platform. The software content-centric router Augustus [10] is a multi-threaded user-space application which carefully exploits the capabilities of the underlying x86 architecture. Specifically, this solution has extensive support for parallelism in the hardware, uses DPDK's fast packet I/O in user-space, optimizes data placement in memory, and makes a meticulous use of the x86's instruction set to achieve good performance. As a result, Augustus is able to saturate a 10Gbit/s link and achieve 10 million packets per second (MPPS) throughput with very small packet sizes. The work by Takemasa et al. [11] shows analytically that reducing the number of DRAM accesses is of foremost importance to design fast NDN packet forwarding on commercial off-the-shelf computers. By proposing two prefetch techniques that partially hide the DRAM latency, their design achieves a throughput of 40 MPPS on a single computer.

Hardware designs use specialized networking hardware to build high-performance NDN packet processing engines. Varvello et al. [12] proposed the design of Caesar, the first content-centric forwarding engine supporting name-based forwarding at high speed. In this paper the authors presented only a preliminary numerical evaluation to validate their design choices. Afterwards, the authors built a prototype of Caeser, presented in [13], using a μTCA chassis and line cards equipped with a network processor. Their evaluation shows that each of its 4 line cards was able to sustain up to 10 Gbps. The work in [14] proposed the first complete design of an NDN forwarding engine implemented on a commercial router. This NDN data plane was entirely written in software, on an Intel Xeon-based line card in the Cisco ASR 9000 router. With the aim of improving both the security and the performance of the solution, the authors considered various design options for hash table design and hash function choice, lookup algorithms for the FIB, and multi-core parallelization. The proposed solution achieves 8.8 MPPS. As main take-away, the throughput of any of these software- and hardware-based solutions is still orders of magnitude below what the new generation programmable switches can sustain. They are also specific to their target systems, lacking portability.

The closest work to ours is the one by Signorello et al. [8]. The authors present a preliminary implementation of NDN in P4. Our work seconds the research vision presented in this paper, and improves the proposed NDN router design in two main ways. First, by addressing the scalability issues of the proposed FIB design. Second, by extending the NDN functionality, including the content store and multicast-capability. In addition, we present a preliminary evaluation of our solution.

III. Architecture

Describing the forwarding behavior of an NDN router in P4 poses interesting challenges. This is mainly due to the P4 language syntax to be oriented to describe standard protocol header formats and packet processing. Conversely, the NDN protocol is based on a highly-nested variable-length packet format and requires unconventional operations. Therefore, the following NDN-specific operations and components can not be easily expressed in P4: a) a variable-length nested packet structure; b) a mass storage module to cache Data packets (the Content Store); c) a stateful data-plane with per-packet update requirements; d) longest prefix match on several variable-length header fields.

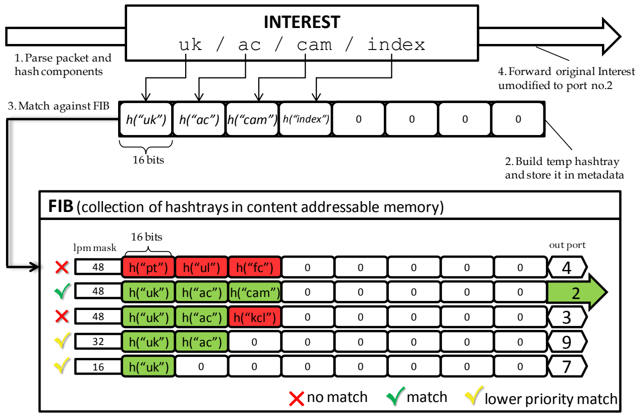
Our design addresses all the above points, improving on each of them over the seminal NDN.p4 work [8]. Importantly, many of the improvements were only enabled by the new P4 language specification [15] (namely, the parser and the content store). We thus argue our work to be an instance demonstrating the favorable progress of the language. More precisely, our solution based on P4-16 features the following improvements (over [8]): i) a more concise and flexible parser description; ii) the caching of data packets in a Content Store implemented as an *extern* module; iii) the multicast of Data packets; iv) a memory efficient and portable FIB implementation. Below we detail how these four improvements have been achieved.

Parser. All the fields in Interest and Data packets are encoded with variable-length Type-Length-Value (TLV) blocks. In theory, Type and Length follow a variable encoding. In practice, the Type encoding can be safely assumed to be fixed since all the NDN code types in use can be encoded in one byte. In contrast, the Length encoding can vary from 1 to 8 bytes, according to the TLV size. Moreover, some TLVs contain actual values, while other contain further nested TLVs. For example, the name field consists of an outmost TLV container whose value is made of several TLV fields, one per each name component (as illustrated logically in Fig. 1).

Since header definitions in P4 can contain only one variable length field, this mandates different header types for those TLVs containing actual values and for those TLVs working as containers and having a variable Length encoding. Overall, the NDN packet format mandates 5 different header types to describe fields in Interest and Data. As result of this specific header definition, the parser code contains many redundant parser states where no header extraction is featured and only the length encoding of a TLV block is determined.

The P4-16 specification introduces some language constructs which we have used to limit the verbosity of the parser description and ease the processing done in the control flow. First, the *header union* and the *sub-parser* routine enable a more concise parser description than the one proposed in [8]. The former allows the definition of a single header union type with all the possible encoding options for the Length. The latter enables redundant code to be consolidated in a single function definition, which is called every time the nature (either container or actual Value) of TLV has to be determined. Second, in P4-16 the parser can hash the components during the parsing stage and store the hashes in metadata. This feature eliminates the need for ad-hoc tables and functions to be introduced in the main control flow [8] to perform the same operation.

Forwarding Information Base (FIB). The FIB table is looked-up at Interest reception to take forwarding decisions. The main challenge in the design of this functional block in P4 is to build a table that maps variable-length strings to output ports at line rate. Unfortunately, P4's match-action tables cannot perform matches on variable-length fields. Therefore, names have to be mapped to fixed-size fields and novel



INTEREST
uk / ac / cam / index
1. Parse packet and hash components
2. Build temp hashtray and store it in metadata
3. Match against FIB
4. Forward original Interest umodified to port no.2
16 bits
FIB (collection of hashtrays in content addressable memory)
lpm mask
out port
no match
match
lower priority match

Fig. 2. The hashtray built from the Interest matches the second FIB entry through LPM.

techniques for matching on name components are required.

Our FIB design hashes each component in the Interest name and stores it in a fixed-size metadata field. Our FIB entries are thus encoded as *hashtrays* (and associated masks).

Definition 1. A **hashtray** is a data structure constructed from a name of Max components. It is divided in Max blocks, with each block i containing the result of the hash of component i.

The mask associated with a hashtray indicates the number of valid blocks in the respective FIB entry. A regular longest prefix match (LPM) is performed against our FIB table containing hashtrays. This enables high throughput in hardware, by taking advantage of TCAM parallelism.

Our FIB structure and matching algorithm are better illustrated through the example in Fig. 2. In the figure we present the configuration used in our implementation, with (1) the target supporting the forwarding of up to 8-component long Interests; and (2) the hashtray being built by using a 16-bit hash function. As a result, the FIB is made of hashtrays which are 128 bits long (i.e., each hashtray is composed of eight 16 bit-long blocks).

In the example, the prefix 'uk/ac/cam' has 3 components. So, the mask of the respective hashtray is equal to $3\times16 = 48$. A temporary hashtray is built from the components in the incoming Interest with name 'uk/ac/cam/index' and stored in a 128-bit metadata field. In this case, the metadata contains a $4 \times 16 = 64$ bit-long chain of the name components hashes, that is, $h(uk)$, $h(ac)$, $h(cam)$ and $h(index)$. The computed hashtray is then matched against the FIB. The entry with the longest mask/prefix, i.e., 'uk/ac/cam', is selected.

To address name collisions, we are currently considering two solutions, as part of on-going work. The first is to use a collision-resistant hash function. The problem is that the block size would need to be increased, at the expense of memory. To avoid this limitation, we are also investigating a hash-chained design. Specifically, the use of double hashing, matching in two tables with different hash functions, and linking the two matches by means of meta data. Our FIB design and implementation is strongly driven by two empirical observations: one about the composition of the prefix names in publicly available ICN name-sets, and one about the availability of fast memories, and their sizes, in commodity switches. First, by analyzing the Cisco's name-set at [16] we have found an average number of 6-7 components per content name. In the same name-set, the percentage of content names made of more than 9 components (8 for the prefix name plus 1 for the content identifier) is very low ($\ll 10\%$)[1]. This analysis suggests that routes for the prefix names in today's available name sets for the ICN can be mapped to our FIB. Second, by fixing the size of a hashtray to 128 bits, we guarantee that this design can be easily ported to a large variety of commodity networking hardware. These appliances already feature hardware support in terms of fast memories (SRAM and TCAM) to implement tables that match on keys of similar sizes for other purposes, such as LPM on IPv6 addresses, or Access Control Lists. Longer names can be supported by increasing the size of the hashtray (e.g., a 256-bit long hashtray enables doubling the number of name components, and is still within the range of existing hardware [4]). Even longer names can be relayed to the control-plane. As long names are very rare, we do not expect this to become a bottleneck in practice. Importantly, the FIB is meant to contain aggregated name prefixes, and in our analysis we pessimistically considered prefixes of the finest granularity.

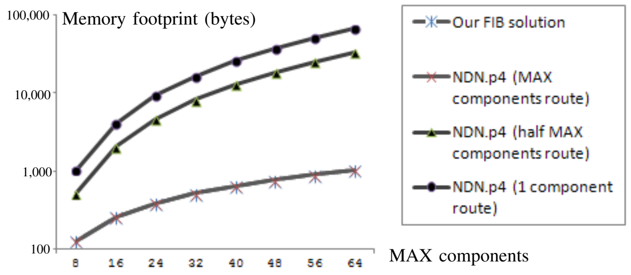


Fig. 3. Memory footprint of a FIB entry.

Finally, our FIB design improves over the one proposed in NDN.p4 [8], scaling better in terms of the memory space required per entry. In the FIB design of NDN.p4 the number of table entries for a single FIB entry was proportional to the maximum number of components *in processed packets*. Furthermore, the priority among the entries was defined by the control plane, and a rule-set modification could potentially require a re-order of the priorities of the existing entries. Instead, our design requires a single table entry per FIB route, with the priority of each entry encoded in the mask (as in typical LPM), so no re-ordering is required if the set of entries changes over time. Fig. 3 compares the required amount of memory of the two FIB designs as a function of the number of components. As can be seen, our solution scales much better. As the number of name components present in the

[1]A more detailed description of our statical analysis is available at https://github.com/signorello/ICN_nameSets.

packet increases, the memory footprint of NDN.p4 becomes impractical.

Pending Interest Table and Content Store. We implement the PIT in P4 registers, which represent the target's fast on-chip memories meant to hold generic state across packets. In our PIT design, registers store bitmasks. Each bitmask is as large as the target's number of ports. Each jth bit in the mask corresponds to the interface j. When an Interest packet is received from an interface f, the following steps are executed. First, the hash of the full Interest name is computed and used as an index to access a register cell: $h(name) = i$. Second, the bitmask B stored at i is retrieved. The new value for B is then computed by setting the bit for the interface f in the bitmask: B= B bit-OR 2^f. Finally, the new bitmask B is stored back in the register cell at i.

At Data reception, the bitmask is retrieved from the PIT by repeating the first two steps. Then, the Data is multicast to all the interfaces recorded in the bitmask. Finally, the PIT entry is deleted by clearing the relative bitmask and, depending on the policy, the Data may be cached.

The maximum number of PIT entries is dictated by the target's available memory space for registers and by its number of ports. According to the latest trend of SRAM size in ASICs (see Table 1 in [17]), commercial switching chips feature up to 100 MB of fast on-chip memory today. Hence, if we consider 64 ports targets, the design can sustain up to 1 million entries. As explained in Section II, PIT entries' lifetime is short – in practice it is set greater than an average Internet round trip time, translating to a few seconds. The short entries' lifetime reduces the worst case demand for PIT space.

In our implementation, we have modified the BMv2 software target to read the bitmask and multicast the packet copies accordingly. For forwarding devices that do not support this multicast behavior natively, the multicast operation could be implemented in P4. However, it should be noted that such a solution would require recirculating a Data packet N times at worst, where N is the size in bits of the bitmask, reducing throughput by a factor of N.

We initially set to use the extern functionality of the P4_16 language to define the Content Store module in our NDN router design. However, the reference software target at the time, BMv2-ss, had little to no support for it. Given this restriction, the software logic for the **Content Store** module has been implemented directly into the software target.

IV. Evaluation

For preliminary evaluation and functionality tests of our design we use the software switch BMv2 simple_switch [18] on an OptiPlex 3020 with a quad-core 3,20 GHz CPU and 4GB of RAM running Ubuntu 16.04. The test environment consists of a custom packet generator and sniffer interconnected by the bmv2-ss target running our NDN router written in P4.[2] As today's available P4 targets do not fully support the latest language feature-set, we could not consider all features in this evaluation. Specifically, for all the experiments reported in this section, PIT and FIB operations are performed on every Interest packet, while the Content Store is disabled.

[2]Our P4 program and all the complementary software used for testing are available at https://github.com/netx-ulx/NDN.p4-16.

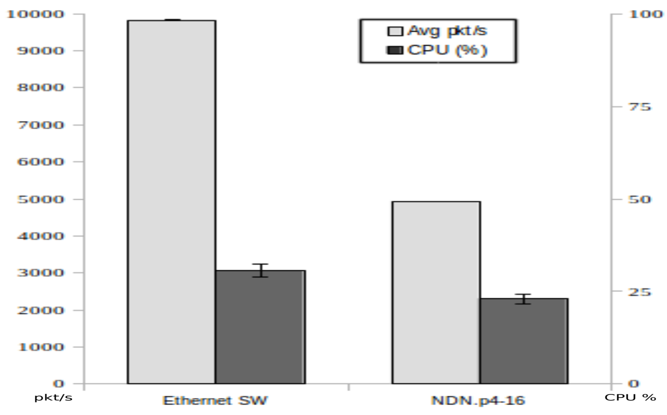


Fig. 4. Performance comparison between our router and a P4 Ethernet switch.

Throughput. We first compared a P4 Ethernet switch against our NDN router in number of packets processed per unit of time. The packet processing logic of the Ethernet switch forwards incoming packets out to a different interface and replaces source and destination MAC addresses. For this experiment, we measured the highest packet generation frequency for which bmv2 experiences no packet loss. These throughput values, reported in Figure 4, correspond to the average over several rounds of tests where the packet generator emits 30 consecutive bursts of packets with the same frequency.

The Ethernet switch achieves an average close to 10k pkt/s, while our NDN router manages to process around 5k pkt/s. This lower throughput is demonstrative of the overall higher cost of packet processing in NDN, motivating the need for specific optimisations in the target, something left as future work. The difference in the CPU usage consumed by bmv2 when running the two P4 programs was relatively low in the different rounds of this experiment. Nevertheless, the slightly lower use of the CPU by the NDN router seems to indicate that the switch could indeed be optimised for this specific use case.

Latency. This experiment evaluates the latency introduced by the parser, the PIT and the FIB to the total Interest packet processing time. We measure the latency values in two experiments where the Interest packets carry names with different number of name components.

Our generator issues bursts of 50,000 Interests at a constant frequency which does not cause any packet loss to bmv2. We have measured the latency as the difference between the timestamps measured at input and output interfaces of bmv2.

The baseline impact of the bmv2 software to any packet forwarding logic is isolated as follows. A minimal P4 program does no parsing/deparsing nor ingress/egress processing. It only forwards packets towards a fixed output interface. The

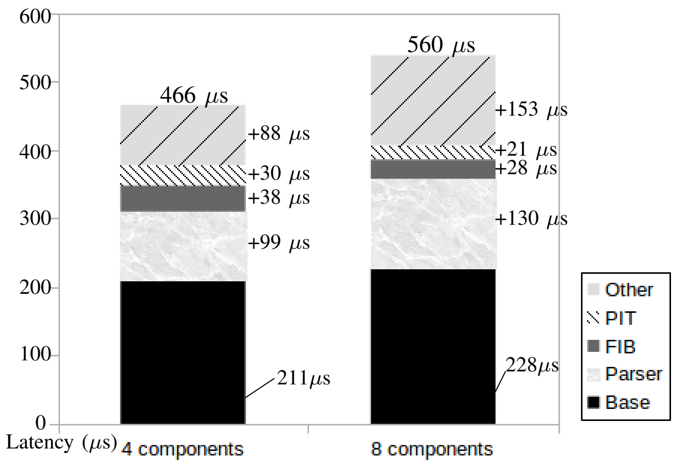


Fig. 5. Weights of the main blocks of our P4 NDN router program over the bmv2 overhead baseline.

measured latency for this program is considered to be the pure bmv2 overhead and is shown at the base of the columns in Figure 5 (211μs and 228μs for the experiments using 4 and 8 name components, respectively).

We used customizations of our NDN program to assess the impact of every functional block in isolation. Therefore, the latency blocks reported in each column correspond to iterated experiments where only one functional block is active at turns. The FIB block corresponds to an experiment where only the FIB table call is performed, meaning the time required to build the hashtray is reported in the 'Other' block.

Overall, this experiment outlines that the parser is the main responsible for the latency overhead in packet processing. Moreover, the latency introduced by this functional block is affected by the number of components, as expected. By contrast, the difference in the latency measured by the FIB and the PIT blocks in the two different experiments lays within the measured standard deviation. Bmv2 performs the operations for parsing and building the hashtray sequentially. Given the nature of these tasks, we anticipate that introducing parallelism will provide a significant increase in performance.

V. Discussion and Future Work

In this paper we presented the design, implementation, and a preliminary evaluation of an NDN router written in P4.

As next step, we will evaluate our solution on programmable networking hardware. Specifically, we are preparing experiments to run it on a testbed that includes different P4 targets, including one Barefoot Tofino switch [5] and a hardware switch using the 100Gb/s NetFPGA SUME platform [19]. With this setup we plan to evalute the performance of two programs: one written in P4-14 (for the Tofino switch), and the P4-16 version we presented here (using the P4-NetFPGA compiler [20]). As our design took the specifics of hardware into consideration (namely, making use of the TCAM), we expect our solution to exhibit high performance.

We also plan to investigate the implementation of packet forwarding engines for other similar ICN architectures. We are intrigued by at least two questions. First, we would like to understand what new hardware primitives would be useful to have in programmable hardware (and P4) to better support different ICN protocols. Our initial experience with NDN indicates that new features for parsing and matching on variable-length names would be useful, namely to facilitate parallel processing and reduce code repetition. Second, we wonder if certain ICN protocols and packet formats are already better served by existing programmable switching hardware. For example, a different packet encoding for the NDN naming scheme prepends a list of offsets to the name to identify the beginning of every single component. This may facilitate optimizations of the parser stage in existing programmable hardware.

Acknowledgments. This work was supported by FCT through funding of the uPVN project, ref. PTDC/CCI-INF/30340/2017, and LASIGE Research Unit, ref. UID/CEC/00408/2013.

References

[1] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, 2012.

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *CoNEXT'19*.

[3] L. Zhang *et al.*, "Named data networking," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 66–73, 2014.

[4] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *SIGCOMM'13*.

[5] Barefoot tofino. [Online]. Available: https://barefootnetworks.com/technology/#tofino

[6] Broadcom tomahawk ii. [Online]. Available: https://www.broadcom.com

[7] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 87–95, 2014.

[8] S. Signorello, J. Francois, O. Festor, and R. State, "Ndn. p4: Programming information-centric data-planes," in *IEEE NetSoft*, 2016.

[9] C. Yi, A. Afanasyev, L. Wang, B. Zhang, and L. Zhang, "Adaptive forwarding in named data networking," *ACM SIGCOMM CCR*, vol. 42, no. 3, pp. 62–67, 2012.

[10] D. Kirchner, R. Ferdous, R. L. Cigno, L. Maccari, M. Gallo, D. Perino, and L. Saino, "Augustus: a ccn router for programmable networks," in *ACM ICN'16*.

[11] J. Takemasa, Y. Koizumi, and T. Hasegawa, "Toward an ideal ndn router on a commercial of-the-shelf computer," in *ACM ICN'17*.

[12] M. Varvello, D. Perino, and J. Esteban, "Caesar: A content router for high speed forwarding," in *ACM ICN'12*.

[13] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue, "Caesar: A content router for high-speed forwarding on content names," in *ANCS'14*.

[14] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: Fast and dos-resistant forwarding with hash tables," in *ANCS'13*.

[15] The P4_16 Language Specification, version 1.0.0. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[16] Content Name Collection. [Online]. Available: www.icn-names.net/

[17] 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 *SIGCOMM'17*.

[18] The behavioral model, a.k.a. the p4 software switch, repository. [Online]. Available: https://github.com/p4lang/behavioral-model

[19] Netfpga. [Online]. Available: https://netfpga.org

[20] P4-netfpga. [Online]. Available: https://github.com/NetFPGA/P4-NetFPGA-public/