NDN-DPDK: NDN Forwarding at 100 Gbps on Commodity Hardware

Junxiao Shi
National Institute of Standards and Technology
Gaithersburg, MD, USA
junxiao.shi@nist.gov

Davide Pesavento
National Institute of Standards and Technology
Gaithersburg, MD, USA
davide.pesavento@nist.gov

Lotfi Benmohamed
National Institute of Standards and Technology
Gaithersburg, MD, USA
lotfi.benmohamed@nist.gov

ABSTRACT

Since the Named Data Networking (NDN) data plane requires name-based lookup of potentially large tables using variable-length hierarchical names as well as per-packet state updates, achieving high-speed NDN forwarding remains a challenge. In order to address this gap, we developed a high-performance NDN router capable of reaching forwarding rates higher than 100 Gbps while running on commodity hardware. In this paper we present our design and discuss its tradeoffs. We achieved this performance through several optimization techniques that include adopting better algorithms and efficient data structures, as well as making use of the parallelism offered by modern multi-core CPUs and multiple hardware queues with user-space drivers for kernel bypass. Our open-source forwarder is the first software implementation of NDN to exceed 100 Gbps throughput while supporting the full protocol semantics. We also present the results of extensive benchmarking carried out to assess a number of performance dimensions and to diagnose the current bottlenecks in the packet processing pipeline for future scalability enhancements. Finally, we identify future work which includes hardware-assisted ingress traffic dispatching, dynamic load balancing across forwarding threads, and novel caching solutions to accommodate on-disk content stores.

CCS CONCEPTS

• **Networks → Routers**; **Network performance analysis**; *Network layer protocols*; *Point-to-point networks.*

KEYWORDS

Named data networking, Information centric networking, NDN forwarder, Software router, Packet forwarding engine, High-speed forwarding, Network performance, Kernel bypass, Commodity hardware, Performance benchmarking

ACM Reference Format:
Junxiao Shi, Davide Pesavento, and Lotfi Benmohamed. 2020. NDN-DPDK: NDN Forwarding at 100 Gbps on Commodity Hardware. In *7th ACM Conference on Information-Centric Networking (ICN '20), September 29–October 1, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3405656.3418715

ICN '20, September 29–October 1, 2020, Virtual Event, Canada
2020. ACM ISBN 978-1-4503-8040-9/20/09.
https://doi.org/10.1145/3405656.3418715

1 INTRODUCTION

Named Data Networking (NDN) is a new networking protocol with a data-centric communication architecture based on retrieval of named content rather than packet delivery between hosts [9, 39]. It is one of the most prominent instances of Information Centric Networking (ICN). Communication in NDN is receiver-driven: a consumer sends an Interest packet carrying the desired content name, the network uses this name to forward the request toward a producer or an in-network cache, and eventually a Data packet is returned to the consumer on the reverse path. A fundamental component in a NDN network is the *forwarder* (or router) that implements NDN's communication model following the behavior described by Shi [29, Chapter 3]. Accordingly, an NDN router must perform name-based lookups of potentially large tables using variable-length hierarchical names and simultaneously update its internal state on a per-packet basis. This makes wire-speed NDN forwarding challenging to achieve. At the same time, a number of scientific and other data-intensive applications [1, 4, 23, 27, 28] are hampered by the lack of such high-speed capability.

In this paper we present the design of **NDN-DPDK**, a high-performance NDN forwarder capable of achieving a throughput of over 100 Gbps while running on commodity x86 hardware. NDN-DPDK adopts several architectural optimizations ranging from better algorithms and data structures to reduced kernel and system call overhead, which was made possible by leveraging the fast user-space packet processing framework Data Plane Development Kit (DPDK) [17]. DPDK is available for many common 10/100 Gbps Ethernet adapters and provides a set of libraries to accelerate packet processing tasks, such as ring buffers, memory pools, and thread management. This enables our forwarder to receive and transmit packets directly from user space without going through the Linux kernel. Additionally, NDN-DPDK takes full advantage of the parallelism offered by modern multi-core CPUs.

Our open-source codebase, available on GitHub[1], possesses several unique features that advance the state of the art in NDN software routers:

- First, to the best of our knowledge, NDN-DPDK is the first complete implementation of a high-speed NDN forwarding engine on real hardware. Previous attempts [10, 11, 16, 22, 30, 33, 36] either focused on a subset of the data plane functions, did not support the full NDN protocol and name matching semantics, prioritized modularity and flexibility over performance, or relied on simulations rather than actual implementation.

[1]https://github.com/usnistgov/ndn-dpdk

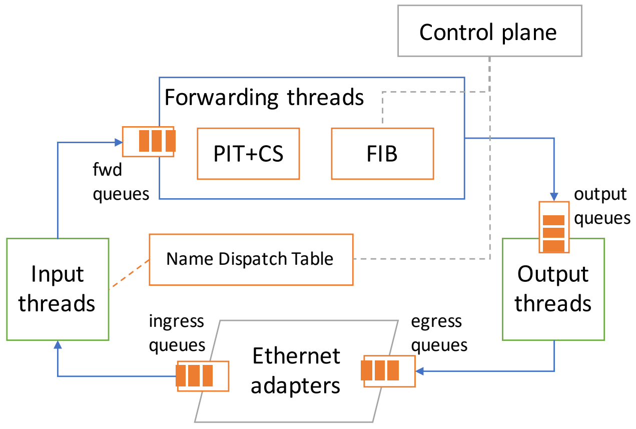


Figure 1: High-level view of NDN-DPDK's architecture.

- Second, since NDN allows for the Interest name to be a prefix of the Data name, we propose a novel approach for fast prefix matching between Interests and Data. An important ingredient of this approach is the PIT token, a small hop-by-hop header field added to each packet to accelerate PIT lookups.
- Third, in order to efficiently support prefix matching in NDN's Content Store, we developed a new solution based on indirect entries. Without such solution, an Interest carrying a non-exact name would have to be answered by the producer, reducing the effectiveness of in-network caching.

The rest of this paper is organized as follows. Section 2 provides an overview of NDN-DPDK's design. The forwarder's input and output stages are described in section 3, where packet dispatching for Interest and Data is also covered. Sections 4 and 5 introduce the main data structures and the details of their implementation. Support for NDN's forwarding strategies is discussed in section 6. The results of extensive performance benchmarks are presented in section 7. Finally, we review previous publications related to ICN forwarding in section 8, and conclude the paper in section 9, where we also list our future work.

2 DESIGN OVERVIEW

The forwarding plane of NDN-DPDK adopts a multi-threaded architecture (fig. 1). Each packet is processed in three stages:

(1) The *input stage* receives a packet from a network interface, decodes it, and dispatches it to a forwarding thread according to a Name Dispatch Table (NDT).
(2) The *forwarding stage* applies the NDN forwarding rules to the packet; this stage includes the traditional FIB (Forwarding Information Base), PIT (Pending Interest Table), and CS (Content Store) components, as well as the forwarding strategies.
(3) The *output stage* prepares outgoing packets and passes them to a network interface for transmission.

This architecture allows the forwarder to make use of all available CPU cores and process several packets in parallel.

2.1 Memory Pools and NUMA Sockets

NDN-DPDK uses DPDK's *mempool* library to preallocate most of its data structures in 1 GiB hugepages. This eliminates the unpredictable latency of calling `malloc()` on the packet processing hot path, and reduces the complexity of handling runtime memory allocation failures. These memory pools are also pinned to physical memory pages and cannot be swapped out by the kernel, which ensures consistent access latency.

Most server-grade machines adopt a Non-Uniform Memory Access (NUMA) design. Under NUMA, each CPU, memory DIMM, and PCIe peripheral belongs to one *NUMA socket*. A CPU can access its local memory (memory located on the same NUMA socket) faster than non-local memory (memory located on a different NUMA socket). Using DPDK's *Environment Abstraction Layer* (EAL) library, NDN-DPDK pins its threads to specific CPU cores, and allocates most data structures used by a thread in NUMA-local memory. This minimizes memory access latency for these data structures.

Similarly, a PCIe Ethernet adapter also belongs to a NUMA socket. All packets received on a given adapter are stored in a memory pool local to that adapter's NUMA socket. Thus, NDN-DPDK assigns the input and output threads serving each interface to CPU cores located on the same NUMA socket. Even so, during normal operations it is inevitable to access packets across NUMA socket boundaries: for instance, when an incoming packet is dispatched to a forwarding thread on another NUMA socket, or when the egress interface happens to be on a different NUMA socket.

2.2 Sharded Data Structures

A classical NDN forwarder has three main data structures (or *tables*): the FIB guides Interest forwarding toward the producer, the PIT gets Data back to the consumer, and the CS provides in-network caching. NDN-DPDK has multiple forwarding threads and they all need to access these three tables. Concurrent access, however, requires thread safety, which would increase the design complexity and reduce performance.

Instead, NDN-DPDK tries to avoid sharing the tables as much as possible: each forwarding thread has a private instance of the PIT and the CS, and a (partial) copy of the FIB. The first two do not require any cross-thread access, thus they are implemented using non-thread-safe data structures, while the FIB still needs to be updated from the control plane and therefore employs a low-overhead *Read-Copy-Update* (RCU) synchronization mechanism (section 4). This approach also enables allocating all three tables for each thread in NUMA-local memory in order to minimize the access latency, as explained in section 2.1.

2.3 Internal Packet Queues

Each forwarding thread receives the packets dispatched to it by the input stage via a set of three FIFO queues, one for each packet type: Interest, Data, Nack. The queues are provided by DPDK's *ring* library, which implements a ring buffer with a fixed capacity and lockless enqueue/dequeue operations. Packets are taken from any one of these queues in *bursts* instead of one at a time. Burst dequeuing amortizes the overhead of the ring buffer bookkeeping operations and reduces the number of CPU instruction cache misses, because multiple packets of the same type are processed together, typically following the same code path.

Initial testing suggested that the forwarding stage can easily become a bottleneck in configurations with few forwarding threads. To alleviate this bottleneck, NDN-DPDK adopts a CoDel-based [15] queue management algorithm between the input stage and the forwarding stage. If the minimum queuing delay stays above *target* (5 ms by default) for *interval* milliseconds (initially 100 ms), the forwarding thread inserts a *congestion mark* [25] in the next packet, prompting the consumer to slow down.

The forwarder also prioritizes Data over Interests, by dequeuing fewer packets from the Interest queue than from the Data queue at each iteration. This is because dropping a Data packet would not only waste the resources already spent on processing the corresponding Interest, but also cause the PIT entry to linger until its expiration, while losing an Interest is less harmful.

2.4 Life of a Packet

An incoming frame is received by an input thread, which parses it according to the NDN Packet format [19] and recognizes it as an Interest (section 3.1). The packet is then assigned to a forwarding thread based on its name (section 3.2) and is placed on a queue that goes to that thread. The forwarding thread dequeues the Interest. It first queries the PIT and the CS (section 5), but, assuming that this node did not recently receive any Interest or Data with the same name, no match is found in either table. Therefore, the thread creates a new PIT entry and records the ingress interface as a downstream node. Next, it performs a FIB lookup (section 4) to determine which forwarding strategy should be making forwarding decisions and the potential next hops. The strategy is invoked and decides to forward the Interest to a particular next hop (section 6.1). The forwarding thread finally passes the Interest to an output thread for transmission (section 3.4).

When a Data packet arrives, the input thread determines which forwarding thread previously handled the corresponding Interest based on a hop-by-hop header field in the Data packet (section 3.3), and passes the packet to that thread. The forwarding thread first locates the PIT entry that can be satisfied (section 5.2). It then checks which downstream nodes have expressed matching Interests and passes copies of the Data packet to output threads for transmission (section 3.4). It also notifies the forwarding strategy that the Interest has been satisfied, so that the strategy can make better decisions in the future (section 6). Finally, the Data packet is cached in the CS and the PIT entry is deleted.

3 FACE I/O AND PACKET DISPATCHING

NDN-DPDK is optimized for 10/100 Gbps Ethernet adapters attached to a PCIe bus. Using DPDK's *poll mode* drivers, NDN-DPDK can send and receive packets directly from user space without going through the operating system kernel. This significantly improves performance by eliminating the overhead of system calls and interrupt handling.

An NDN *face* is a generalization of the concept of network interface, on which NDN packets can be transmitted and received. NDN-DPDK supports only point-to-point faces, unlike other NDN

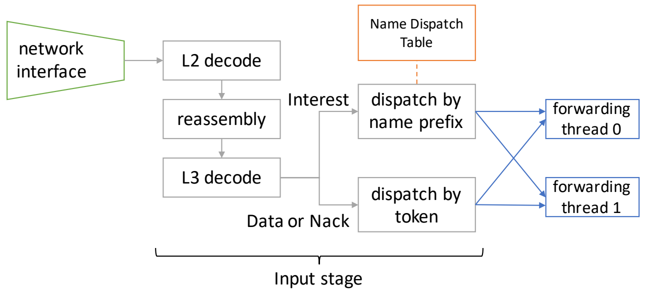


Figure 2: Input thread procedure.

forwarders, and deliberately does not support multicast communication on a single face. This directly follows from NDN-DPDK's primary use case in high-performance computing and data-intensive science [35], where the network is closely managed and peers are administratively configured, thus multicast-based discovery is not needed. Furthermore, to keep the face system simple, NDN-DPDK currently supports only Ethernet faces and cannot tunnel over IP or any other protocols[2]. The user can create one or more faces on a given Ethernet adapter; each face is distinguished by a different remote MAC address and optionally a VLAN ID.

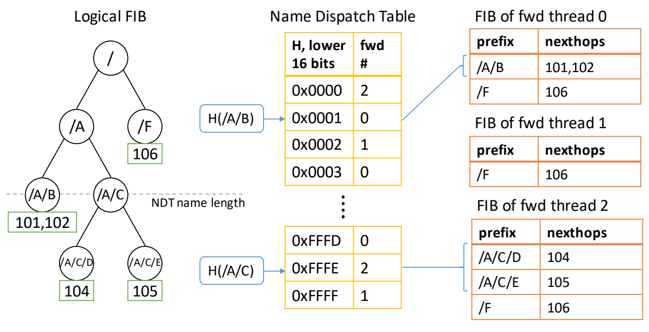
3.1 Input Stage

The forwarder's input pipeline, depicted in fig. 2, starts with receiving frames from an Ethernet adapter. As mentioned before, there can be multiple faces on the same adapter, but each face must have a distinct remote MAC address or VLAN ID. The adapter is configured to steer frames belonging to each face into a different receive queue. An *input thread* is connected to one or more of these receive queues and is responsible for decoding each incoming frame with a three-step procedure: (1) strip the Ethernet header and any VLAN tags; (2) decode as an NDNLP packet [21], performing reassembly if needed; (3) continue decoding the reassembled NDN network-layer packet and classify it as Interest, Data, or Nack.

The decoding routines store information about a parsed packet along with the packet buffer itself, in a private area reserved in the buffer header. This avoids the need for a separate memory allocation, which improves throughput but consumes more memory on a per-packet basis. Moreover, NDN-DPDK's decoding routines are optimized to serve the forwarder and do not have to accommodate the needs of consumer or producer applications. This leads to several design differences compared to the parsers that can be found in any general-purpose NDN library. For example:

- For Data, decoding stops just before the `Content` element, because the forwarding algorithms do not need to know the packet's payload or its signature.
- For similar reasons, the decoder ignores an Interest's `ApplicationParameters` element and everything that comes after it.
- For the `MetaInfo` element in a Data packet, the decoder reads only the `FreshnessPeriod` field, because other fields do not affect forwarding.

[2]Better tunneling support is planned for a future version.



Name Dispatch Table

H, lower 16 bits	fwd #
0x0000	2
0x0001	0
0x0002	1
0x0003	0
⋮	
0xFFFD	0
0xFFFE	2
0xFFFF	1

FIB of fwd thread 0

prefix	nexthops
/A/B	101,102
/F	106

FIB of fwd thread 1

prefix	nexthops
/F	106

FIB of fwd thread 2

prefix	nexthops
/A/C/D	104
/A/C/E	105
/F	106

Figure 3: Name Dispatch Table and FIB partitions.

3.2 Dispatching Interests by Name

After packet decoding, the input thread dispatches each network-layer packet to a forwarding thread. For an Interest, a prefix of the name determines which forwarding thread will process the packet. This dispatching algorithm is name-based so that two Interests with the same name end up in the same forwarding thread, which ensures the effectiveness of Interest aggregation. By using a name prefix instead of the entire name, Interests with a common prefix go to the same forwarding thread, which enables the forwarding strategy to collect measurements on a per-prefix basis.

As we will see in section 6, forwarding strategies operate at the FIB entry granularity. In theory, Interest dispatching could follow the FIB entries, which would fulfill the two goals above. However, the FIB is a fairly complex data structure and performing a full lookup in the input stage would be too slow. Therefore, the dispatching algorithm employs a much simpler method to determine the granularity: it takes the first k components of the Interest name as prefix. If the Interest has fewer than k name components, the entire name is used. The value of k should be chosen such that the resulting prefixes are shorter than most FIB entries, to keep forwarding strategy measurements effective, but also long enough that there is a sufficient number of distinct prefixes for load balancing among forwarding threads. NDN-DPDK sets $k = 2$ in its default configuration, but this parameter can be tuned according to the characteristics of the incoming traffic.

After determining the name prefix, the dispatching algorithm queries the *Name Dispatch Table* (NDT), a table unique to the NDN-DPDK forwarder (fig. 3). The input thread dispatches the Interest to the queue leading to the forwarding thread identified by the NDT entry. In order to be as simple as possible and maintain a predictable lookup speed, the NDT is not a name-indexed data structure but a linear vector of 2^b entries ($b = 16$ by default), where each entry contains a forwarding thread identifier. The lookup algorithm computes the SipHash [2] over the name prefix, takes the lower b bits of the hash value as an index into the vector, and returns the forwarding thread identifier in that entry. This allows an NDT lookup to complete in $O(1)$ time complexity.

3.3 Dispatching Data by Token

Data must be dispatched to the same forwarding thread that processed the Interest. As described in section 2.2, each forwarding thread has a private instance of the PIT. Therefore, information about a pending Interest is available only in the forwarding thread that forwarded the Interest and only that thread is able to correctly process a Data in reply to the forwarded Interest.

Although we could perform another prefix-based dispatch on the Data name, the name dispatching algorithm breaks down in one specific case. The NDN protocol allows name discovery: an Interest may be satisfied if its name is a prefix of the Data name and the Interest contains the `CanBePrefix` flag. For example, Data `/A/B/1` can satisfy an Interest with name `/A` and `CanBePrefix=1`. Applying the name dispatching algorithm in section 3.2 and observing that the Interest name in this case has fewer than two components, we can see that the Data would be dispatched under the `/A/B` prefix, while the Interest would be dispatched under the `/A` prefix. If the NDT entries corresponding to these prefixes map to two different thread identifiers, the Data would go to a forwarding thread that has no knowledge about the Interest.

To solve this problem we introduce the *PIT token*, an 8-byte hop-by-hop field carried in the NDNLP header of each packet [21]. Downstream nodes attach a PIT token when transmitting an Interest. Upstream nodes are expected to attach the same token when replying to the Interest with a Data or Nack packet. NDN-DPDK uses a few bits of this token to encode the forwarding thread identifier. When a forwarding thread transmits an Interest, it puts its own identifier in the PIT token. When Data comes back, the input thread can simply read that part of the PIT token and dispatch the Data to the correct forwarding thread.

For a Nack packet, either dispatching method would work correctly. We chose the token dispatching method as it is more efficient.

3.4 Output Stage

The output stage controls the transmission of outgoing packets. For each outgoing network-layer packet, it performs NDNLP fragmentation (if needed), prepends an Ethernet header, and enqueues the resulting link-layer frames for transmission on the Ethernet adapter. In the current version of NDN-DPDK the workload of the output threads is light. However, it will likely increase in the future as we plan to implement more advanced congestion control and queue management schemes in the output stage.

4 FIB STRUCTURE AND LOOKUP

The FIB is a read-mostly table. The forwarding threads perform a *Longest Prefix Match* (LPM) on the FIB to determine where to send each incoming Interest. On the other hand, FIB updates are seldom needed and only occur in response to a management command from the control plane.

NDN-DPDK's FIB design is inspired by So et al. [30]. In their design, the FIB entries are stored in a hash table keyed by the name prefixes. They also propose a 2-stage LPM lookup algorithm:

(1) The FIB has a fixed parameter M, such that the majority of FIB entry names have fewer than M components.
(2) When inserting a FIB entry whose name has more than M components, a virtual entry is inserted at depth M that indicates the maximum FIB depth under its prefix.
(3) Given an Interest, an LPM on the FIB starts with the M-component prefix of the Interest name.

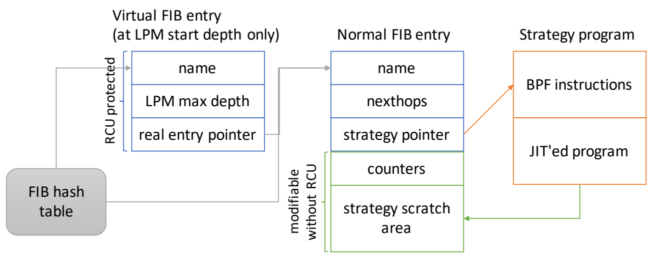


Figure 4: Structure of normal and virtual FIB entries.

(4) If this lookup finds a virtual entry, the LPM is restarted at the maximum FIB depth indicated in the virtual entry.
(5) Otherwise, the LPM continues toward shorter prefixes until a normal FIB entry is found.

In NDN-DPDK, we adopted the same FIB structure (fig. 4) and the same 2-stage lookup algorithm. However, our multi-threaded architecture requires thread safety, which we achieved with two techniques.

First, we use the Userspace Read-Copy-Update (URCU) library [3, 13] to allow both FIB queries from the forwarding thread and FIB updates from the management thread to take place simultaneously. A benefit of RCU is that its read-side overhead is minimal, which matches well with the read-mostly nature of the FIB. We use the quiescent-state-based flavor of RCU because it has the smallest overhead. This flavor requires every thread to periodically indicate a quiescent state; for a forwarding thread, this occurs before processing each burst of packets. The hash table implementation comes from URCU's lock-free resizable RCU hash table, with the resize functionality disabled to provide more predictable performance. FIB entries are allocated from a DPDK memory pool instead of the default `malloc()` memory allocator.

Second, each forwarding thread is given its private FIB instance. Each FIB instance contains only the name prefixes served by the forwarding thread. Compared to having a single FIB shared among all forwarding threads, this approach ensures the FIB entries are allocated on the same NUMA socket as the forwarding thread, avoiding memory access across NUMA boundary. Moreover, it allows the forwarding strategy (section 6) to store collected measurements on the FIB entry itself, without needing a separate measurements table.

5 COMBINED PIT AND CONTENT STORE DESIGN

The PIT and the Content Store are both read and modified frequently in the data plane, specifically:

(1) The CS is queried for every incoming Interest to check if it can be satisfied by cached Data.
(2) If not, the Interest is forwarded and a PIT entry must be inserted, unless one already exists.
(3) Upon receiving a Data packet, the PIT is queried to find which pending Interest(s) can be satisfied.
(4) When a PIT entry is satisfied, it must be erased and a CS entry inserted to cache the Data.
(5) If the CS is full, it may need to evict some entries to make room for the new Data.

Observing that item 4 often deletes a PIT entry and inserts a CS entry at the same name prefix, So et al. [30] propose combining the PIT and the CS into a single hash table, so that a satisfied PIT entry can be replaced with a CS entry without incurring the cost of a second table lookup. NDN-DPDK adopts this design and merges PIT and CS into the *PIT-CS Composite Table* (PCCT).

However, [30] is intended for the CCNx protocol [14], which differs from NDN in the Interest-to-Data matching rules. Both NDN and CCNx allow a Data[3] packet to satisfy an Interest if they have the same name. NDN additionally allows a Data to satisfy an Interest if the Interest name is a prefix of the Data name and the Interest carries the `CanBePrefix` flag. Moreover, NDN Interests can carry a forwarding hint that, when present, should be used in place of the Interest name to determine the forwarding path. These major protocol differences make our PCCT design inevitably different from the one described in [30].

Given the frequent updates in both PIT and CS, a thread-safe PCCT shared across all forwarding threads could easily become a bottleneck. We decided early on that each forwarding thread should have its own private instance of the PCCT. Contrary to the FIB case, the control plane does not need to interact with either the PIT or the CS. Therefore, we can implement the PCCT using non-thread-safe data structures, which are typically faster than their thread-safe counterparts.

5.1 Logical Structure

The overall structure of the PCCT is a combination of three data structures:

- A DPDK *mempool* to allocate PCCT entries from.
- A *name hash table* for name-based lookups. This reuses the SipHash values already computed during packet dispatching (section 3.2).
- A *token hash table* to find what PIT entries can be satisfied by incoming Data, using the PIT token carried on the Data packet (section 5.2).

Logically, each PCCT entry contains a name, a chosen forwarding hint, two PIT entries, and one CS entry (fig. 5). The name is required, but all other fields are optional.

An entry is identified by the combination of its name and the chosen forwarding hint. When a forwarding thread processes an Interest that carries forwarding hints, it performs FIB lookups using those hints, and chooses the first hint that matches a FIB entry. The PIT entry created from that Interest and the CS entry for its reply Data are then placed on a PCCT entry with the chosen forwarding hint. Having the latter as part of the PCCT entry identifier logically isolates the PIT and the CS for each forwarding hint into different partitions. This mitigates a well-known cache poisoning attack caused by forwarding hints and makes NDN-DPDK the first NDN forwarder to *securely* support forwarding hints.

Each PCCT entry can contain up to two PIT entries with the same name and chosen forwarding hint. Per the NDN protocol [19], an Interest may carry the `CanBePrefix` and/or `MustBeFresh` flags

[3]"Data" is NDN terminology; the CCNx equivalent is "Content Object".

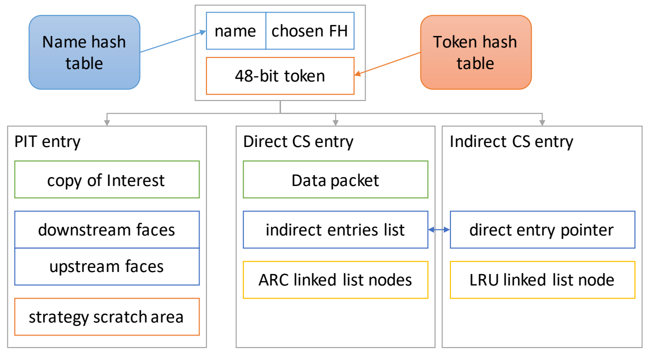


Figure 5: PIT-CS Composite Table.

that affect Interest-to-Data matching in forwarding and caches. However, the protocol is vague on how the PIT should aggregate Interests with the same name but different flags. We argue that two Interests that differ only in the `CanBePrefix` flag can be aggregated because the presence of this flag widens the set of Data that can be matched. On the other hand, two Interests that differ in the `MustBeFresh` flag cannot be aggregated, because forwarding them with the flag set would reject non-fresh Data that would otherwise satisfy the Interest that did not have the flag, while forwarding them without the flag could incorrectly satisfy the `MustBeFresh` Interest with non-fresh Data. Hence, we decided to store up to two PIT entries in each PCCT entry, one with `MustBeFresh` and one without.

5.2 PIT Lookup by Data

As mentioned before, the NDN protocol allows prefix match between Interest and Data. When a Data packet arrives, the forwarder would have to perform an LPM lookup on the PIT to determine which pending Interests can be satisfied. This could become a performance bottleneck or even an attack surface because the Data name can have many components. It is infeasible to apply the 2-stage lookup algorithm (section 4) to the PIT because the overhead of maintaining M would be too high. Instead, we propose a different approach based on the *PIT token*, a short hop-by-hop header field already introduced in section 3.3 for Data packet dispatching. Here, we extend its usage to accelerate PIT lookups.

Whenever a forwarding thread inserts a PIT entry, it allocates a *PCCT entry token* to the enclosing PCCT entry and adds it to the token hash table. Then, every outgoing Interest created from this PIT entry will carry the PCCT entry token inside its PIT token field. When a Data packet comes back, the forwarding thread can quickly locate the PCCT entry in $O(1)$ time via the token hash table. It is still necessary to verify that the Data indeed satisfies the Interest through name comparison, to prevent attacks from forged tokens.

5.3 Prefix Matching in the Content Store

Being a hash table, the PCCT only supports exact match queries using a key that consists of an Interest/Data name and a chosen forwarding hint. Thus, if an Interest name is a prefix of the Data name and the Interest carries the `CanBePrefix` flag, the exact match algorithm cannot retrieve the Data from the hash table. This severely limits the effectiveness of in-network caching, because any Interest carrying a non-exact name would have to be answered by the producer instead of being satisfied from the CS.

To address this issue, NDN-DPDK introduces the concept of *indirect CS entry* to provide partial support for prefix matching. An indirect CS entry is a special CS entry named after an Interest and containing a pointer to a *direct CS entry*. By contrast, a direct CS entry is a regular CS entry named after the Data, and contains the bits of the Data packet itself.

When the forwarder receives a Data in reply to an Interest with non-exact name, it inserts two entries into the CS: a direct entry with the Data name and the Data packet, and an indirect entry with the Interest name and a pointer to the direct entry. This allows CS matching with a prefix name, under the assumption that the consumer application consistently uses the same prefix (or a small subset of prefixes) to perform name discovery. If a future Interest with the same name as the previous Interest arrives, an exact match lookup in the CS using that Interest name will find the indirect entry, from where we can follow the pointer and retrieve the direct entry and the cached Data packet. Conversely, if an Interest with a different name arrives, even if it matches the Data, the forwarder will not be able to find the cached Data because an indirect CS entry for that Interest name does not exist.

6 FORWARDING STRATEGIES

The forwarding strategy is a component that controls various aspects of the Interest forwarding behavior. The strategy decides where to forward an incoming Interest when it cannot be satisfied by the local CS. It also decides whether to perform any corrective actions when a Nack is received. These decisions are based on inputs such as the next hops in the matching FIB entry, the downstream and upstream records in the PIT entry, as well as any collected measurements on recent data retrievals by Interests sharing a common prefix.

Experience with early NDN deployments has shown that different applications need different Interest forwarding behaviors. This provides a strong motivation to support multiple forwarding strategies with different decision making algorithms, and to dynamically choose a forwarding strategy based on application needs and network environments. As outlined by Jacobson et al. [9], the basic idea is for each FIB entry to contain a program, written for an abstract machine specialized to forwarding choices, that determines how to forward Interests. NDN-DPDK realizes this vision using *extended BPF* (eBPF) [5, 18], an evolution of the original Berkeley Packet Filter (BPF) [12].

6.1 Strategy Program

Each strategy is an eBPF program, i.e., a list of low-level instructions, such as load/store, arithmetic, and comparison operators, that can be executed by the eBPF virtual machine. In addition, the program can call a few higher-level routines that are provided to the strategies by the core forwarding engine. These include setting a timer, sending the current Interest on the specified face, and responding to an Interest with a Nack.

The strategy program exports a main function that is invoked, or *triggered*, when:

- An Interest packet arrives and cannot be satisfied with cached Data. The strategy will have to decide how to forward it.
- A Data packet arrives and satisfies an Interest. This trigger allows the strategy to collect path measurements, such as round-trip time and satisfaction ratio.
- A Nack packet arrives and does not fall under one of the simple cases that are automatically handled by the forwarding plane itself. The strategy can then decide if any corrective actions must be taken.
- A timer previously scheduled by the strategy expires.

Both FIB and PIT entries contain a writable *scratch area* for the strategy to store its state and record measurements. The FIB entry scratch area (fig. 4) is suitable for information related to a whole namespace, while the PIT entry scratch area (fig. 5) is suitable for information related to a specific pending Interest. Other than these areas, the strategy is able to inspect the current packet and has read-only access to a subset of fields in the FIB and PIT entries.

6.2 Strategy Selection and FIB Updates

NDN-DPDK associates a forwarding strategy to every FIB entry. When an Interest arrives and cannot be satisfied by the CS, the forwarding plane performs a FIB lookup, and the matched FIB entry determines not only the potential next hops but also which strategy should handle the Interest. This design is in line with [9], except that the strategy eBPF program is not stored in the FIB entry itself, but referenced by the FIB entry, so that the same strategy may be used by multiple FIB entries without storing duplicate copies. Data and Nacks are always handled by the same strategy that processed the Interest. In case the strategy or the FIB entry was changed while the Interest was pending, the forwarder handles the returning Data/Nack packet using a built-in fallback procedure and no strategy is triggered.

After a FIB update, the strategy has to restart with an empty scratch area, because it would be infeasible to migrate the data in the scratch areas during FIB updates. Indeed, although FIB entries are RCU-protected (section 4), the FIB entry scratch area is not. While this allows the strategy to modify the scratch area without going through the relatively expensive write-side RCU procedure, it also means that the control plane thread cannot safely copy the scratch area contents from the old FIB entry to the new one.

7 PERFORMANCE EVALUATION

We conducted extensive testing of NDN-DPDK in order to determine its performance characteristics under a variety of workloads. In particular, we measured the *aggregate forwarding rate*, in terms of bps (bits per second) and pps (Data packets per second), and the *per-packet forwarding latency*, in microseconds. Note that our pps metric accounts only for the Data packets because they are carrying the application content. The total number of packets (Interests and Data) actually forwarded by NDN-DPDK is at least twice[4] the reported amounts.

[4]We say "at least twice" and not "exactly twice" because in the rare case of packet loss, the consumer must retransmit the Interest. These retransmissions are not included in our statistics since they do not affect the final application-layer goodput.

In all the experiments described below, the forwarder is running on a Supermicro 6039P-TXRT server equipped with dual Intel Xeon Gold 6240 CPUs (18 cores at 2.60 GHz, with Hyper-Threading disabled), 256 GB of 2933 MHz memory in four channels (64×1 GB hugepages have been allocated to NDN-DPDK on each NUMA socket), and Mellanox ConnectX-5 100 Gbps Ethernet adapters. The operating system is Ubuntu Linux 18.04, with DPDK v19.11 and NDN-DPDK commit 34f561f4ef0e5790d4999107dcbb4c2eab82af66. The forwarder node is connected to two traffic generators, one on each Ethernet port, via direct attach copper cables. The traffic generators emulate a producer application and a number of consumers requesting content from the producer. Each consumer instance expresses Interests under a given name prefix and employs a congestion control algorithm similar to TCP CUBIC [24]. The Interest names consist of five distinct parts:

(1) The producer prefix.
(2) The consumer thread ID.
(3) The consumer node name followed by a random number that changes with each execution.
(4) The placeholder component `/127=Y` repeated as many times as necessary (possibly zero) to make the total Interest name length equal that required by the experiment scenario.
(5) The segment number.

An example Interest name is `/C/0/B_77378826/127=Y/127=Y/35=%07%C3`. The producer responds to each Interest with a Data packet, either of the same name or with an additional suffix name component `/127=Z`. The Data packet signature is neither generated by the producer nor verified by the consumer, although a signature field of proper length but with a fictitious value is present in every Data packet.

All experiments share a common "base configuration" consisting in: 8 forwarding threads, 4 components in Interest and Data names, 1000 bytes of application payload in every Data packet, 2^{16} NDT entries (see section 3.2), FIB start depth (the M parameter in section 4) set to 8, and a maximum CS capacity equal to 2^{15} entries per forwarding thread (see section 5). In each experiment we vary some of these parameters in order to assess their impact on the overall performance of the forwarder.

The forwarding rate reported for each benchmark is the arithmetic mean of 10, 50, or 150 consecutive runs (depending on the experiment), executed after a warm-up run whose results are discarded. Each run lasts 60 seconds. The forwarding latency is measured for each packet, from the moment it enters the forwarder's input stage to when it is dequeued by the output thread and handed over to the network adapter for transmission.

7.1 Forwarding Threads and Name Length

This benchmark demonstrates how NDN-DPDK's performance scales with the number of CPU cores assigned to the packet forwarding tasks. As described in section 2, each CPU core used by NDN-DPDK is entirely dedicated to running one and only one thread, hence we will use the terms "core" and "thread" interchangeably. Given that the number of input and output threads is constrained by how many network cards are installed on the system, we can only vary the number of forwarding threads, in order to distribute the incoming traffic among a larger or smaller amount of CPU cores.

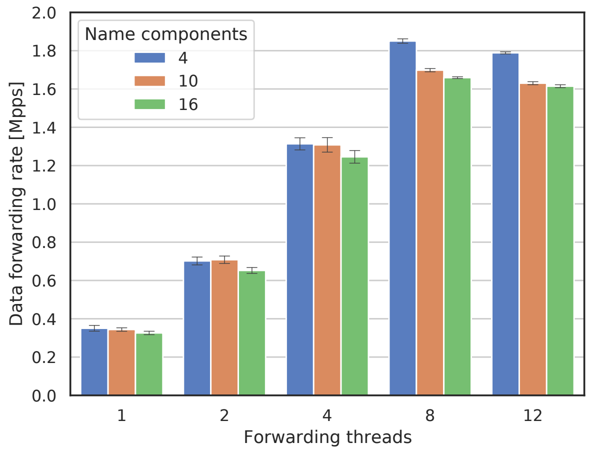


Figure 6: Mean and standard deviation of the throughput with different numbers of forwarding threads and name components.

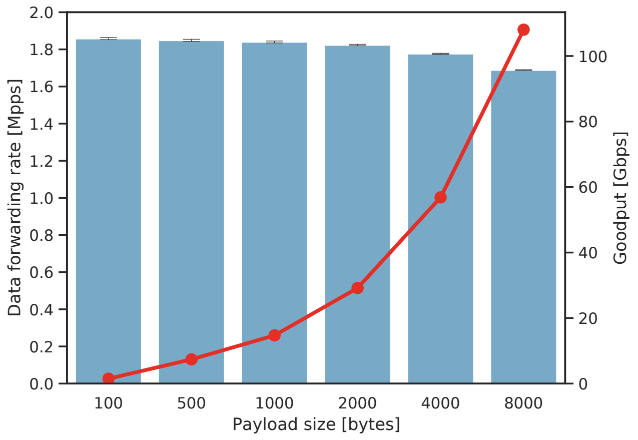


Figure 7: Mean and standard deviation of the throughput in Mpps (bars) and Gbps (line) with varying Data payload sizes.

Figure 6 shows that the throughput grows almost linearly up to 4 forwarding threads, then slows down but still improves up to 8 threads, where we reach a peak rate of about 1.84 Mpps (million Data packets per second). Further increasing the number of threads beyond that does not help performance, in fact the throughput slightly declines with 12 threads, as more cores on the same CPU are competing for hardware resources. A deeper analysis, not reported here for lack of space, revealed that with 8 or more forwarding threads, the input stage of the forwarder's pipeline becomes the bottleneck. We plan to eliminate this bottleneck in a future version of NDN-DPDK.

In fig. 6 we also illustrate the effect of the Interest name length, expressed in number of name components, on the overall forwarding rate. Longer names make the forwarder marginally slower, up to 10 % in the worst case, and the slowdown is more pronounced with 8 and 12 forwarding threads, i.e., when the input stage becomes the bottleneck. This can be explained by the fact that it is the input thread that parses the Interest name (section 3.1) and the time complexity of the decoding algorithm is linear in the number of name components.

7.2 Data Payload Length

The application-layer payload is carried inside NDN Data packets in a field called `Content`. As detailed in section 3.1, NDN-DPDK never decodes or otherwise accesses this field, because it is completely opaque to NDN routers and does not affect any forwarding decision. Therefore, we expect that varying the Data payload length will have very limited impact on the forwarder's performance. The benchmark results in fig. 7 indeed confirm our intuition: the difference between best-case and worst-case pps forwarding rate never exceeds 10 %.

This experiment also allows us to determine the maximum aggregate throughput in bits per second (bps) that can be delivered by NDN-DPDK between two network adapters: 108 Gbps, with 8000 bytes of content per packet. This number represents the application-layer *goodput*, thus excluding all Ethernet headers, Interests/Data names, Data signatures, and so on.

7.3 Scalability of FIB Lookup

We tested the scalability of NDN-DPDK's FIB lookup algorithm (section 4) with up to 1 million entries (name prefixes). We can notice from the table below that the number of FIB entries has no impact on the forwarding rate, while the Interest latency is only minimally affected.

	Fwd. rate (kpps)		Interest latency (µs)	
FIB entries	Mean	σ	Median	95th percentile
10^4	1840	5.59	90	227
10^5	1835	4.92	92	234
10^6	1839	4.42	97	249

7.4 Content Store Capacity and Hit Ratio

One major feature of NDN-DPDK is the built-in support for a limited form of prefix matching in the Content Store, i.e., the ability to return, in some cases, a cached Data packet that has a longer name than the incoming Interest. However, this type of lookup is more expensive than a simple exact match algorithm due to the additional indirection and the greater number of PCCT entries that need to be consulted, as explained in section 5.3. With this benchmark we tried to quantify the performance difference between exact and prefix match. In order to trigger a meaningful number of CS hits, we added a second consumer node to the experiment topology, connected to the forwarder through a third Ethernet port. The second consumer starts fetching content 100 ms after the first one.

Figure 8 shows the aggregate throughput results of 150 runs, together with the linear regression line obtained via Theil-Sen estimation [26, 32]. Overall, prefix matching is 15 % to 17 % slower than exact matching when the CS capacity is set to 2^{17} entries per thread, and between 10 % and 18 % slower with a capacity of 2^{20} entries. We can also see from the figure that the forwarding rate is positively correlated with the hit ratio. This is because satisfying

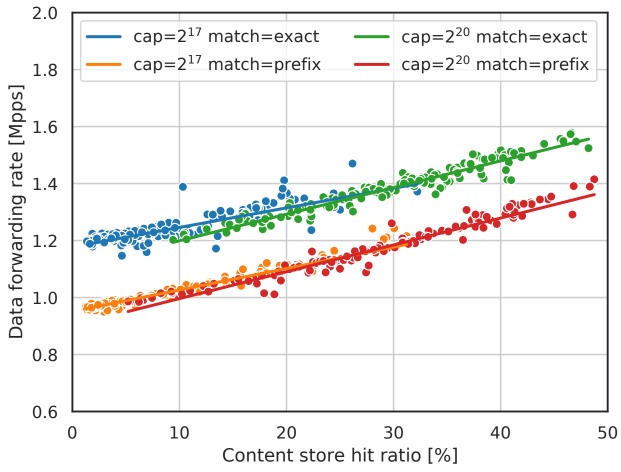


Figure 8: Forwarding rate vs. hit ratio with exact and prefix match and different Content Store capacities.

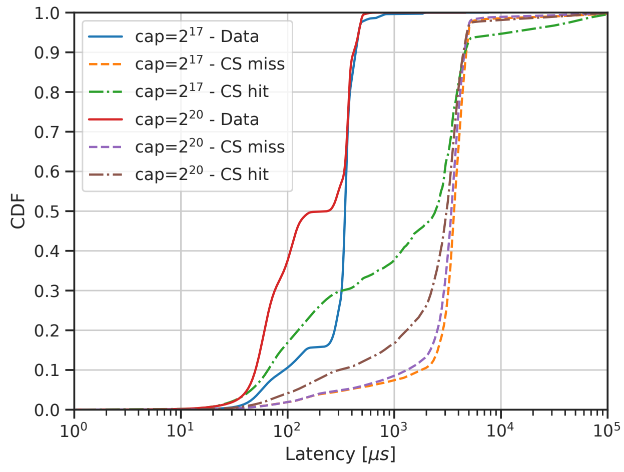


Figure 10: Cumulative distribution function of the forwarding latency with exact name matching.

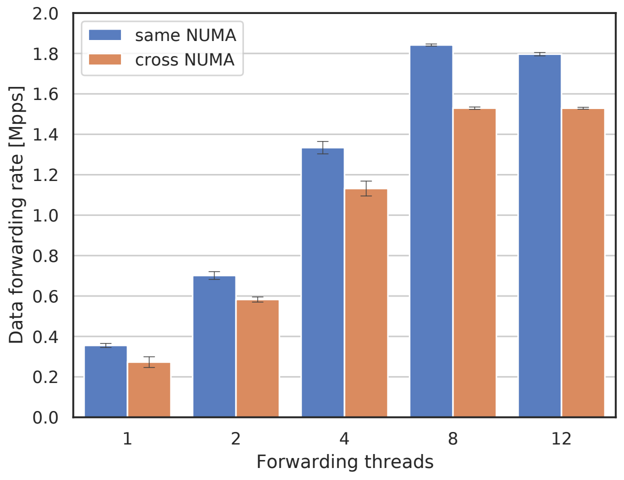


Figure 9: Mean and standard deviation of the throughput under same-NUMA and cross-NUMA memory accesses.

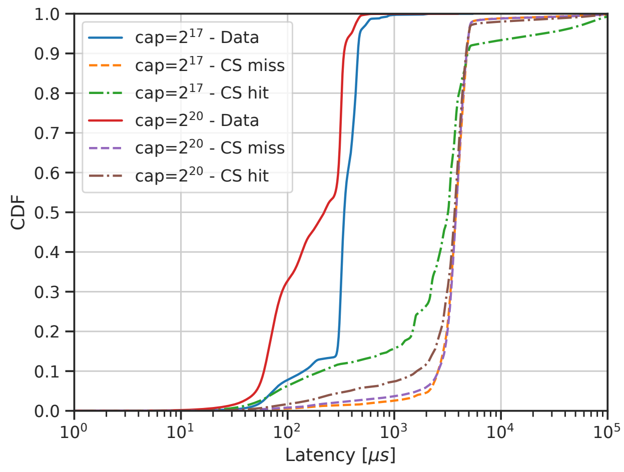


Figure 11: Cumulative distribution function of the forwarding latency with prefix name matching.

more Interests from the cache means that fewer packets will have to be further processed through the pipeline stages.

We also looked at the processing latency of each of the three main code paths that an incoming packet can take: *CS miss* (an Interest that is forwarded upstream), *CS hit* (an Interest that is answered with a Data packet from the local cache), and *Data* (a Data packet that is inserted into the CS and then forwarded downstream). The latency distributions are plotted in figs. 10 and 11 for the two scenarios of exact and prefix matching.

7.5 Impact of Nonlocal Memory Access

As mentioned in section 2.1, accessing memory on another NUMA socket incurs a higher latency compared to accessing local memory. In this benchmark, we measured the impact of cross-NUMA memory accesses on NDN-DPDK's performance. We ran the same test twice: first between two network cards installed on the same NUMA node as the forwarding threads, then we moved one of the cards to the other NUMA node, thereby forcing all traffic to require non-local memory accesses in either the Interest or the Data direction. The experiment was repeated 50 times for each scenario. The results in fig. 9 confirm that repeatedly crossing the NUMA boundary can significantly degrade the packet processing throughput, up to 20 % slower with 8 forwarding threads.

8 RELATED WORK

The research community has studied numerous performance aspects of the NDN and CCN forwarding models in great detail [11, 31, 33, 34, 36–38]. However, most of these works focus on just one or few facets of the problem, such as scalable FIB lookup, efficient name encoding, distributed PIT architectures, and hierarchical Content Stores, but only a handful of papers propose a complete design for a high-speed name-based forwarding engine.

Caesar [16] is the first full implementation of a content-centric router with a design based on hash tables. Its data plane runs on a specialized hardware platform consisting of four 10 GbE line cards and takes advantage of several hardware-specific accelerations. The FIB is distributed across the line cards and, in addition to a hash table, it uses a prefix Bloom filter, which requires hardware assistance to be beneficial. *Caesar* does not support prefix matching between Interest and Data names and relies on a custom packet format with a few fixed-length header fields to expedite parsing, therefore it cannot easily be extended to handle NDN semantics.

Another notable router design centering on hash tables is proposed by So et al. [30]. NDN-DPDK took several key ideas from this paper, enhancing them to provide additional features and support the more powerful NDN name matching semantics, as explained in the previous sections. Among them, the 2-stage LPM algorithm, the PIT partitioning scheme, and the unification of the PIT and CS data structures. A similar FIB lookup scheme is also described by Fukushima et al. [6]; however, this approach is effective only on FIB entries that are leaf nodes in the name hierarchy tree.

Kirchner et al. [10] present two open-source implementations of their software CCN router *Augustus*: a standalone monolithic forwarding engine based on DPDK and running on general-purpose hardware, and a modular prototype built with the Click framework. Their solution is able to reach a throughput of 10 Mpps, but it suffers from several shortcomings that preclude its practical deployment in NDN networks. For instance: (1) It implements a custom ICN packet format that contains a fixed-length header that complicates future evolutions of the protocol, in violation of NDN's "universality" design principle [20]. (2) Similarly to *Caesar*, it can perform only an exact match between Interest and Data packets. (3) To take advantage of hardware dispatching, *Augustus* encapsulates ICN packets in IPv4 packets and configures the adapter's Receive Side Scaling (RSS) in such a way that the hash result depends only on the source IPv4 address. To ensure that a returning Data packet is dispatched to the thread that has the corresponding PIT entry, *Augustus* requires the IPv4 source address field to contain the CRC32 hash of the name. This technique, while functionally similar to NDN-DPDK's PIT token, requires neighboring routers to agree on a hash function before they can communicate. (4) The FIB is not designed to be thread-safe, thus rendering FIB updates impossible while the forwarder is handling data plane traffic.

Other approaches that use hop-by-hop state carried between Interest and Data to accelerate or eliminate PIT lookups have been proposed in CCN-GRAM [7] and ADN [8].

9 CONCLUSION AND FUTURE WORK

This paper describes NDN-DPDK, our implementation of a high-performance NDN forwarder on commodity server hardware. NDN-DPDK is built upon a number of novel ideas, such as the introduction of the PIT token for efficient Interest-Data prefix matching, the introduction of indirect CS entries for efficient CS prefix matching, resulting in more effective in-network caching, and the secure support for NDN's forwarding hints. Initial benchmarks demonstrate that NDN-DPDK is capable of sustaining 1.8 Mpps, or a corresponding 108 Gbps when 8 kB Data packets are used.

Lessons learned from the benchmarking effort helped us identify future work needed for performance improvements in NDN-DPDK. As we observed, the input thread becomes the bottleneck when there are 8 or more forwarding threads. We are exploring potential design changes to allow attaching multiple input threads to the same Ethernet adapter. At the same time, we hope that hardware acceleration can speed up packet dispatching by sending the vast majority of incoming packets directly to a forwarding thread, bypassing the input thread bottleneck and ensuring that the packet buffers are allocated from a memory pool on the same NUMA socket as the forwarding thread. This goal can be achieved progressively:

(1) If the network card's RSS supports matching at arbitrary offsets in the Ethernet frame, Data and Nack can be dispatched directly, by configuring RSS to read the PIT token field. This will reduce the work of the input thread to just processing Interests and fragmented packets.
(2) Dispatching Interests will likely require eBPF/P4 hardware or FPGA-based solutions. One idea is to download a copy of the NDT into the hardware accelerator, which will then (partially) decode the incoming NDN packets and perform NDT lookups with the Interest names, using the algorithm in section 3.2. At this point, only fragmented packets need to be processed by software input threads.

Our roadmap for future releases also includes: (1) Expanding the Content Store capacity by caching some packets in persistent memory (e.g., Intel Optane) or NVMe disk storage. These devices are more cost effective and energy efficient than traditional RAM, and would potentially allow a forwarder to have caching capacity in the order of a few terabytes. However, their slower access speed requires novel multi-tier caching algorithms. (2) Automatic load balancing among forwarding threads by dynamically adjusting the NDT entries to spread the forwarder's workload. (3) VXLAN tunnelling with hardware offloads. (4) Continuous in-depth performance profiling, to guide further optimizations in memory access (e.g., CPU cache prefetching) and data structure design (e.g., hash table collision resolution algorithms). (5) Implementation and benchmarking of in-forwarder cryptographic processing, such as implicit digest for Data packets. (6) New forwarding strategies with enhanced capabilities.

DISCLAIMER

Any mention of commercial products or reference to commercial organizations is for information only; it does not imply recommendation or endorsement by NIST, nor does it imply that the products mentioned are necessarily the best available for the purpose.

REFERENCES

[1] Mohammad Alhowaidi, Byrav Ramamurthy, Brian Bockelman, and David Swanson. 2017. The case for using content-centric networking for distributing high-energy physics software. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2571–2572.
[2] Jean-Philippe Aumasson and Daniel J Bernstein. 2012. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*. Springer, 489–508.
[3] Mathieu Desnoyers and Paul E. McKenney. [n.d.]. *Userspace RCU Implementation*. Retrieved August 31, 2020 from https://liburcu.org/
[4] Chengyu Fan, Susmit Shannigrahi, Steve DiBenedetto, Catherine Olschanowsky, Christos Papadopoulos, and Harvey Newman. 2015. Managing scientific data with named data networking. In *Proceedings of the Fifth International Workshop on Network-Aware Data Management*. 1–7.

[5] Matt Fleming. 2017. A thorough introduction to eBPF. *LWN.net* (2017). https://lwn.net/Articles/740157/

[6] Masaki Fukushima, Atsushi Tagami, and Toru Hasegawa. 2013. Efficiently looking up non-aggregatable name prefixes by reducing prefix seeking. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 340–344.

[7] JJ Garcia-Luna-Aceves and Maziar Mirzazad Barijough. 2016. Content-centric networking using anonymous datagrams. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. IEEE, 171–179.

[8] José Joaquin Garcia-Luna-Aceves. 2017. ADN: An information-centric networking architecture for the Internet of Things. In *Proceedings of the second international conference on internet-of-things design and implementation*. 27–36.

[9] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. 2009. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. 1–12.

[10] Davide Kirchner, Raihana Ferdous, Renato Lo Cigno, Leonardo Maccari, Massimo Gallo, Diego Perino, and Lorenzo Saino. 2016. Augustus: a CCN router for programmable networks. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. 31–39.

[11] Rodrigo B Mansilha, Lorenzo Saino, Marinho P Barcellos, Massimo Gallo, Emilio Leonardi, Diego Perino, and Dario Rossi. 2015. Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. 59–68.

[12] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, Vol. 46.

[13] Paul E. McKenney, Mathieu Desnoyers, and Lai Jiangshan. 2013. User-space RCU. *LWN.net* (2013). https://lwn.net/Articles/573424/

[14] M. Mosko, I. Solis, and C. Wood. 2019. Content-Centric Networking (CCNx) Semantics. RFC 8569 (Experimental). https://doi.org/10.17487/RFC8569

[15] K. Nichols, V. Jacobson, A. McGregor (Ed.), and J. Iyengar (Ed.). 2018. Controlled Delay Active Queue Management. RFC 8289 (Experimental). https://doi.org/10.17487/RFC8289

[16] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, and Roger Boislaigue. 2014. Caesar: A content router for high-speed forwarding on content names. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*. 137–148.

[17] DPDK Project. [n.d.]. *Data Plane Development Kit*. Retrieved August 31, 2020 from https://www.dpdk.org/

[18] IO Visor Project. [n.d.]. *eBPF: extended Berkeley Packet Filter*. Retrieved August 31, 2020 from https://www.iovisor.org/technology/ebpf

[19] Named Data Networking Project. [n.d.]. *NDN Packet Format Specification, version 0.3*. Retrieved August 31, 2020 from https://named-data.net/doc/NDN-packet-spec/0.3/

[20] Named Data Networking Project. [n.d.]. *NDN Protocol Design Principles*. Retrieved August 31, 2020 from https://named-data.net/project/ndn-design-principles/

[21] Named Data Networking Project. [n.d.]. *NDNLPv2: NDN Link Protocol, version 2*. Retrieved August 31, 2020 from https://redmine.named-data.net/projects/nfd/wiki/NDNLPv2

[22] Named Data Networking Project. 2018. *NFD Developer's Guide*. Technical Report. NDN-0021, Revision 10. https://named-data.net/publications/techreports/ndn-0021-10-nfd-developer-guide/

[23] Duncan Rand, Simon Fayer, and David J Colling. 2015. Possibilities for named data networking in HEP. In *Journal of Physics: Conference Series*, Vol. 664. IOP Publishing, 052031.

[24] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. 2018. CUBIC for Fast Long-Distance Networks. RFC 8312 (Informational). https://doi.org/10.17487/RFC8312

[25] Klaus Schneider, Cheng Yi, Beichuan Zhang, and Lixia Zhang. 2016. A practical congestion control scheme for named data networking. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. 21–30.

[26] Pranab Kumar Sen. 1968. Estimates of the regression coefficient based on Kendall's tau. *Journal of the American statistical association* 63, 324 (1968), 1379–1389.

[27] Susmit Shannigrahi, Chengyu Fan, and Christos Papadopoulos. 2018. Named data networking strategies for improving large scientific data transfers. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE, 1–6.

[28] Susmit Shannigrahi, Christos Papadopoulos, Edmund Yeh, Harvey Newman, Artur Jerzy Barczyk, Ran Liu, Alex Sim, Azher Mughal, Inder Monga, Jean-Roch Vlimant, et al. 2015. Named data networking in climate research and HEP applications. In *Journal of Physics: Conference Series*, Vol. 664. IOP Publishing, 052033.

[29] Junxiao Shi. 2017. *Named Data Networking in Local Area Networks*. Ph.D. Dissertation. The University of Arizona. http://hdl.handle.net/10150/625652

[30] Won So, Ashok Narayanan, and David Oran. 2013. Named data networking on a router: Fast and DoS-resistant forwarding with hash tables. In *Architectures for Networking and Communications Systems*. IEEE, 215–225.

[31] Junji Takemasa, Yuki Koizumi, and Toru Hasegawa. 2017. Toward an ideal NDN router on a commercial off-the-shelf computer. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*. 43–53.

[32] H Thiel. 1950. A rank-invariant method of linear and polynomial regression analysis, Part 3. In *Proceedings of Koninalijke Nederlandse Akademie van Weinenschatpen A*, Vol. 53. 1397–1412.

[33] Matteo Varvello, Diego Perino, and Leonardo Linguaglossa. 2013. On the design and implementation of a wire-speed pending interest table. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 369–374.

[34] Yi Wang, Keqiang He, Huichen Dai, Wei Meng, Junchen Jiang, Bin Liu, and Yan Chen. 2012. Scalable name lookup in NDN using effective name component encoding. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 688–697.

[35] Edmund Yeh, Ran Liu, Yuanhao Wu, Volkan Mutlu, Yuezhou Liu, Harvey Newman, Catalin Iordache, Raimondas Sirvinskas, Justas Balcas, Susmit Shannigrahi, Chengyu Fan, and Craig Partridge. 2019. SANDIE: SDN-Assisted NDN for Data Intensive Experiments. In *SC19 Network Research Exhibition*.

[36] Wei You, Bertrand Mathieu, Patrick Truong, Jean-François Peltier, and Gwendal Simon. 2012. Dipit: A distributed bloom-filter based pit table for ccn nodes. In *2012 21st International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 1–7.

[37] Haowei Yuan and Patrick Crowley. 2014. Scalable pending interest table design: From principles to practice. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2049–2057.

[38] Haowei Yuan, Tian Song, and Patrick Crowley. 2012. Scalable NDN forwarding: Concepts, issues and principles. In *2012 21st International Conference on computer communications and networks (ICCCN)*. IEEE, 1–9.

[39] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. 2014. Named Data Networking. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 66–73.