On the Design and Implementation of a wire-speed Pending Interest Table

Matteo Varvello†, Diego Perino⋆, Leonardo Linguaglossa⋆

Bell Labs, Alcatel-Lucent, †USA, ⋆France
{*first.last*}@alcatel-lucent.com

***Abstract*—*Information-Centric Networking* (ICN) is a novel networking paradigm that aims at making network routers aware of the data they transfer. This new paradigm requires changes in the routers in order to support networking operations on content names at wire speed. NDN, one of the most popular ICN proposal, also suggests that routers should keep track of what content is requested and from which line-card's interface, in a data structure called *Pending Interest Table* (PIT). In this work, we set out to understand how to design a PIT that can support wire-speed. We survey the existing literature and propose few new designs; then, we evaluate numerically the design spectrum for the PIT. Finally, we implement the most promising design on a network processor and evaluate its performance. Our preliminary results are encouraging: we successfully handle a PIT with about 1 Million entries with a wire-speed of 10 Gbps.**

I. Introduction

The research community has recently proposed *Information-Centric Networking* (ICN) [4], a novel networking approach where information (or content) replace end-hosts as communication entities. ICN proposes to make network elements, such as routers, aware of the data they transfer. On the one hand, this requires more complex networking operations based on content names instead of IP addresses. On the other hand, it has several attracting features: integration of caching in network elements, native multicasting, etc.

NDN [7] is one of the most popular ICN designs (Section II). Among the many interesting features, NDN proposes to aggregate *Interests*, or content requests, when they are addressed to the same content name. This is realized by mean of the Pending Interest Table (PIT), a novel data structure in the context of router design. The PIT keeps track of what content is requested and from which line-card's interface; this ensures a single outstanding Interest in presence of concurrent content requests, and it allows to multicast the *Data* packet received as a response. An efficient design of the PIT is thus key to enable NDN (or ICN) at wire speed.

The PIT's design consists of two aspects: *placement* and *data structure*. Placement refers to where in a router the PIT should be implemented. Data structure refers to how PIT entries should be stored and organized to enable efficient operations. Despite CCNx, NDN's prototype, currently implements the PIT as a central hash-table, many recent works [6], [9], [15] show that this solution quickly becomes a bottleneck as we add more line-cards to a router. Accordingly, these works propose new PIT placements as well as novel data structures.

This work focuses on the design and implementation of a PIT that can support wire-speed (cf. Section III). We start by discussing the goals one needs to keep in mind when designing and implementing a PIT. Then, we set out to understand which placements and data structures are possible. Finally, we evaluate all placements and data structures numerically, while evaluating a specific design via a prototype that we have realized on a 10 Gbps network processor [2].

We identify in the literature three PIT placements: *input only* [9], [15], *output only* [6], and *input-output* [6]. The labels indicate the a content router's linecard where the PIT resides: either in input, output, or both. We also make a contribution by proposing to place the PIT on a third party line-card "delegated" for a set of content names; accordingly, we refer to this placement as *third party*. We also identify the following data structures either directly proposed for the PIT or that we select as promising solutions: linear-chained hash-table (LHT) [8], d-left open-addressed hash-table (DHT) [8], counting Bloom filters (CBF) [9], [15] and encoded name prefix trie (ENPT) [6].

We evaluate numerically the different PIT placements along with the data structures (Section IV-A). With respect to the placement, the third party solution is the most promising: it requires a single probe to the data structure to realize all PIT operations (insert, update and delete), while enabling support for timers (necessary to delete entries as they expire), multipath, correct Interest aggregation and loop detection. The drawback is an additional switching operation in the router's central switch in order to delegate each PIT's operation. With respect to the data structure, we conclude that DHT is the best solution for the PIT: it can be coupled with all PIT's placements, while supporting the highest number of packets per second for both average and worst case scenarios.

Based on the indications from the numerical evaluation, we have implemented a DHT-based PIT on a network processor (Sec IV-B), a software-programmable device optimized for networking applications widely used on router's line-cards [1]. The main outcome of the early experimental evaluation is that a DHT-based PIT can handle traffic up to 10 Gbps while storing up to 1 Million entries, confirming on real hardware the numerical results.

II. Named Data Networking

A content in NDN consists of a sequence of chunks, each addressed with a hierarchical human-readable name, e.g., `/NOMEN/PAPERS/PaperA.pdf/chunk0`. A user requests content by sending several *Interest* packets addressed to the name of each chunk that composes the desired content. Interests are forwarded toward content sources using longest prefix matching (LPM) computed over a set of content prefixes stored in a Forwarding Information Base (FIB). Multipath is supported by forwarding Interest on multiple interfaces. In order to prevent loops, Interests contain a random nonce value. The Interest propagation leaves a trail of "bread crumbs" that a chunk follows to reach back the original requester, thus realizing symmetric routing. These bread crumbs are also used to aggregate Interests for the same chunk, naturally realizing multicast at the network layer. Finally, content routers can temporarily store chunks in a Content Store (CS), and serve them in case of future requests.

To realize symmetric routing and multicasting, NDN uses a *Pending Interest Table* (PIT). The PIT keeps track of the interfaces from where chunks have recently been requested and yet not served. A PIT's entry is the tuple <content_name, list_interfaces, list_nonces, expiration>. The content name indicates for which chunk there is at least a pending Interest; list_interface contains the set of interfaces where at least an Interest addressed to content_name was received; list_nonces contains the set of nonces extracted from the pending Interests; expiration refers to the time when the entry will expire, and it is the sum of the time when the first Interest was received and a settable timeout value. Lookups in the PIT are done using LPM on the content_name as in the FIB.

Three operations can be performed on the PIT: *insert*, *update* and *delete*. The insert operation is used when a new Interest is received: first, we use the content name extracted from the Interest to lookup the PIT and verify whether an entry associated to this content is already present or not. If not, we complete the insert operation by creating a new entry; otherwise, if the entry is not expired the insert becomes an update operation. The update operation requires to further verify that the Interest nonce is not contained within list_nonces, in which case a loop is detected and no entry is added. If not, if the interface from where the Interest was received is contained in list_interfaces, no further action is needed; otherwise, the new interface is added to list_interface. The delete operation is used in two scenarios: when an entry in the PIT expires and when a Data is received. The delete consists of a lookup operation to identify the correct entry, coupled with a low level delete that depends on the data structure used. To summarize, all PIT operations require the same number of probes to the PIT's data structure.

III. Design Space

This Section explores the design space for the PIT. We start by discussing the requirements of a PIT's design. Then, we set out to understand to which extent the current proposals for PIT's *placement*, *i.e.,* "where" in a router the PIT should be implemented, satisfy such requirements; we also present a novel placement that, to the best of our knowledge, has not yet been proposed. Finally, we analyze pros and cons of popular *data structures*, *i.e.,* how PIT entries are stored and organized, when used to implement the PIT.

A. Requirements

Frequent operations – Since Interest are smaller than Data, the "load" on the PIT can be defined as the fraction of traffic which is composed by Interests. Such load drives the frequency of PIT's operations. Let's assume a wire speed of 40 Gbps, Interest packet with a size of 80 bytes and Data packet with a size of 1,500 bytes. As a worst case, we consider a load equal to 100% where no Data is available in response of the Interests; in this case, the wire is saturated by Interests and PIT's operations peak at 60 Million operations per second. However, in a more realistic scenario where Data are correctly transmitted as responses to the Interests, flow balance or a load of 50%, the frequency of PIT's operations reduces to 6 Millions per second.

Deterministic operation time – In a content router, multiple packets are processed in parallel to hide memory access time and increase throughput. However, packets also have to be processed in a specific order to guarantee protocol correctness; it follows that a non deterministic operation time would require input queues to buffer packets, causing processes to idle while waiting for others to terminate. It is thus very important that a PIT design achieves deterministic operation time.

Matching algorithm – NDN suggests to use LPM to perform lookups in the PIT. As highlighted in [16], exact matching on PIT enables the basic NDN functionalities without loss of generality, while gaining in speed and simplicity. Due to the potential high frequency of PIT's operations, in this work we also assume PIT's lookup is performed using exact matching and not LPM; we recognize that more complex access strategies to the PIT can improve NDN performance [5], and we plan to address this issue as future work.

Timer support – PIT's entries are deleted after a timeout to avoid the PIT's size to explode over time. A timeout also enables protection against simple attacks that could overflood a router's PIT [14]. It follows that each PIT's entry has to be associated to a timer, and mechanisms to detect timer expiration and to purge expired entries are needed. The presence of timers increases the rate of delete operations; when the load equals 100%, the deletion rate matches the insertion rate, e.g., 60 Million operations per second in a 40 Gbps link.

Potential large state – The PIT size can be estimated as λ*T, where λ refers to the wire speed and T is the time each entry lives in the PIT. In presence of flow balance, we can assume that PIT entries would not last for more than 80 ms, *i.e.,* an average Internet latency [3]. It follows that the PIT would contain no more than about 250 thousand entries even when λ = 40 Gbps. Conversely, in the worst case each entry would expire and thus last in the PIT for as long as the timeout. If we assume a timeout with value between 500 ms and 1 second, the PIT can contain between 30 and 60 Million entries.

Distributed design – In NDN [7], the PIT is designed as a centralized data structure. A high-speed router distributes data structures that operate at wire speed among its line-cards in order to avoid a central bottleneck. It is thus recommended that each line-card deploys its own PIT. However, moving from a central PIT to multiple decentralized PITs can be quite challenging in order maintain correct Interest aggregation, loop detection, and multipath support.

B. Placement

We assume a content router composed by $\mathcal{N}$ line-cards interconnected by a switch fabric. For simplicity, we logically separate the line-cards between input and output. For the ease of explanation and without loss of generality, we assume absence of the CS. We focus on the following placements: *input only*, *output only*, *input-output* and *third party*.

Input-only – Originally proposed in [9], [15], it indicates that a PIT should be placed at each input line-card. Accordingly, an Interest creates a PIT entry only in the PIT of the line-card where it is received. When corresponding Data returns at an output line-card, it is broadcasted to all input line-cards where a PIT lookup indicates whether the Data should be further forwarded or not. This placement enables multipath, but it lacks loop detection and correct Interest aggregation, as each PIT is only aware of local list_interfaces and list_nonces. Most importantly, this placement requires $\mathcal{N}$ PIT lookups in presence of returning Data, which is a serious bottleneck.

Output-only – Originally proposed in [6], it indicates that the PIT should be placed at each output line-card. Accordingly, an Interest does not create a PIT entry at the input line-card where it is received, but at the output line-card where it is forwarded, selected using LPM in the FIB. This approach allows aggregating Interests received at different line-cards, but it shows limitations in case of multipath. When an Interest received at line-card *i* is forwarded to two output line-cards, *j* and *k*, the returning Data is forwarded twice by line-card *i*; in fact, the Interest creates two entries in PIT_j and PIT_k, respectively, and there is no way for line-card *j* and *k* to detect whether the Data was already received at the other line-card. Similarly, assume an Interest received at line-card *i* is sent to line-card *j* and a second Interest for the same Data is received at line-card *l* but sent to line-card *k*. In theory, the first Data received, either on line-card *j* or *k*, should satisfy both Interests; in practice, two Data are needed with this PIT's placement. Finally, the output-only placement requires a FIB lookup per Interest, even when a previous Interest for the same content was already received. Last, loops cannot be detected as each output PIT is only aware of the local list_nonces.

Input-output – This placement was originally discussed in [6] but dismissed in favor of the output-only placement. It indicates that the PIT should be placed both in input and output. Accordingly, an Interest creates a PIT entry both at the input line-card where it is received and at the output line-card where it should be forwarded. Compared to the output only placement, it has two benefits: no unnecessary FIB lookups and duplicated packets in presence of multipath. However, the input-output placement also suffers from the latter multipath issue as it also requires two Data in order to serve Interests received at different line-cards that were forwarded to different output line-cards. Also, loops cannot be detected for the same reasons as above. A minor problem is that a Data triggers two lookup operations: in the PIT of the line-card where it is received, and in the PIT(s) of the line-card(s) from where it was originally requested. The latter issue is discussed in [6] as the main motivation to dismiss this placement.

Third party – The third party placement indicates that a PIT should be placed at each input line-card as in the input-only placement. However, when an Interest for a content A is received at a line-card it is "delegated" to a third party line-card, here the name. This third party line-card is selected as $j = contentID mod \mathcal{N}$, where $\mathcal{N}$ is the number of line-cards in the router and contentID is the hash (e.g., CRC-32) of the content name, or $H(A)$. Accordingly, the PIT at j aggregates all PIT entries for A independently of the input line-card where an Interest for A was received. No PIT at the output is needed; as Data is received, the output line-card identifies j by performing $H(A) mod \mathcal{N}$. This placement enables both multipath and loop detection as the third party line-card acts as an aggregation point. For example, when two Data are expected at two different output line-cards the first Data received is always forwarded to the third party line-card where it consumes each pending Interest. It follows that as the second Data is received and forwarded to the third party line-card, no PIT entries will be available anymore. In addition, compared to the input-output placement it only requires a single lookup per PIT's operation. The drawback of the third party placement is that it generates an additional switching operation for both Interest and Data. Such increase in switching operations can be absorbed by additional switch fabrics as commonly done in commercial routers [13].

C. Data structure

In the literature, three data structures have been proposed for the implementation of the PIT: *counting Bloom filter* [15], [9], *hash-table* [11], [16], and *name prefix trie* [6]. If not otherwise noted, we assume that the PIT contains n tuples <content name, list_interfaces, list_nonces, expiration>. In the following, we overview each data structure in detail while discussing its strengths and weaknesses.

Counting Bloom filter (CBF) – A CBF is a data structure for membership queries with no false negative probability and tunable false positive probability. Compared to a classic Bloom filter, CBF enables deletion using a counter per bit. In [9], [15], the authors propose to use a CBF to implement the PIT. A CBF-based PIT only stores a footprint of each PIT's entry, *i.e.,* available or not, which realizes great compression. The drawback is the presence of false positives that generate wasted Data transmissions. Also, a CBF-based PIT can only be coupled with the input-only placement since the compression of its entries loses the information contained in list_interfaces which requires to lookup PITs at each input line-card in order to determine where a Data should be forwarded. Finally, a

CBF-based PIT cannot detect loops and support timers, as nonce values and timestamps are lost in the compression as well. The memory footprint of a Bloom filter is $S = \frac{-kn}{log(1-p^{\frac{1}{k}})}$, where k is the number of hash functions, and p the false positive probability. However, a CBF requires $k^{'} \cdot S$ memory, where $k^{'}$ denotes the size of a counter. For a CBF-based PIT, we assume $k = 8$, $k^{'} = 5$ and $p = 0.1\%$.

Hash-table – It is a data structure that maps keys to values. In [11], [16], the authors suggest to implement the PIT using a hash-table where a content name is used as key and its corresponding PIT's entry is used as a value. Compared to the CBF-PIT, a PIT based on a hash-table can be coupled with all placements, and it can detect loops (if the PIT placement supports it as well) and support timers. These features come at the expense of a larger memory footprint compared to CBF.

In theory, a PIT based on a hash-table can perform all operations with a single memory access. In practice, multiple accesses are needed in presence of collisions, *i.e.,* when multiple keys map to the same bucket. A classic hash-table uses chaining, *i.e.,* a list per bucket, to handle collisions. Chaining guarantees that PIT operations are accomplished in $2+\frac{\alpha}{2}$ memory accesses on average, where $\alpha = \frac{n}{m}$ and m refers to the number of buckets. However, when collisions happen, up to $O(\frac{log(n)}{log(log(n))})$ accesses (assuming n=m) are needed, which can severely hurt the required determinism.

Several approaches exist to improve upon the classic hash-table with chaining [8]. Multiple choice hash-tables, as d-left hashing, are data structures where d hash functions ($d \geq 2$) are used: each entry is hashed d times and added to the less loaded bucket among the d identified. This strategy trades increased complexity and average access time, computation of d hashing functions and d probes to the data structure, with lower collision probability, which in turn reduces the number of memory accesses in the worst case, e.g., $O(\frac{log(log(n))}{d\phi_d})$ (assuming n=m) where ϕ_d is the asymptotic growth rate of the d-th order Fibonacci numbers. Open-addressed hash-tables are another solution where every bucket stores a fixed number of items; the size of a bucket is limited by the amount of data that can be read with a single access to the memory. It follows that even in presence of collisions a single memory access is enough. The drawback is a larger memory footprint compared to the previous hash-tables.

Based on this discussion, we propose the following solutions to implement a PIT based on a hash-table: *linear-chaining hash-table* (LHT), and *open-addressed d-left hash-table* (DHT). We assume every PIT tuple is 48+l bits long: l bits for the content name, 16 bits for expiration, 16 bits for list_nonces, 16 bits for list_interfaces. A LHT entry requires 32 additional bits to store the CRC of the content name and 32 bits to store a pointer to the next element of the chain, summing up to 112+l bits per entry. For DHT, we consider $d = 2$ and we set the maximum number of items per bucket equal to the longest possible chain; it follows that DHT is over-dimensioned w.r.t. the amount of elements to be stored. The number of memory accesses per operation is $d+1 = 3$ if the d sub-tables are accessed sequentially (DHT), and 2 if the d sub-tables are accessed in parallel (DHTp). In addition to the 48+l bits of the PIT tuple, a DHT-based PIT entry requires 32 bits to store the CRC of the content name, *i.e.,* 80+l bits. As DHT is over-dimensioned, it is possible to reduce its memory footprint by removing the content name from the PIT tuple, and replace it with a pointer, 32 bits, achieving a total of 112 bits. This requires to store a list of content names in a separate data structure that also accounts in the total memory footprint.

In order to support deletion, we propose a lazy mechanism. Specifically, we propose to remove an expired entry only when this entry is accessed by another PIT operation. In such case, LHT requires an additional write to the memory in order to rearrange pointers; this additional write is not needed with DHT as there are no pointers to maintain.

Name prefix trie – It is an ordered tree used to store/retrieve values associated to "components", set of characters separated by a delimiter; for example, `NOMEN` is a component in e.g., `/NOMEN/PAPERS/PaperA.pdf/chunk0` and the delimiter is `/`. The name prefix trie supports LPM, and exact matching as a subset of it. The *Encoded Name Prefix Trie* (ENPT) [6] reduces the memory footprint of a name prefix trie by encoding each component to a 32-bits integer called "code". The drawback is that this compression requires to introduce a hash-table to map codes to components. The ENPT-based PIT described in [6] does not specify any mechanism to detect loops and remove PIT entries with expired timers; however both operations can be accomplished assuming the usage of the tuple described above as a PIT entry. To do so, we simply have to add to each PIT's entry the code associated to the content_name. Similarly, the lazy deletion mechanism discussed above can be used to remove entries when needed.

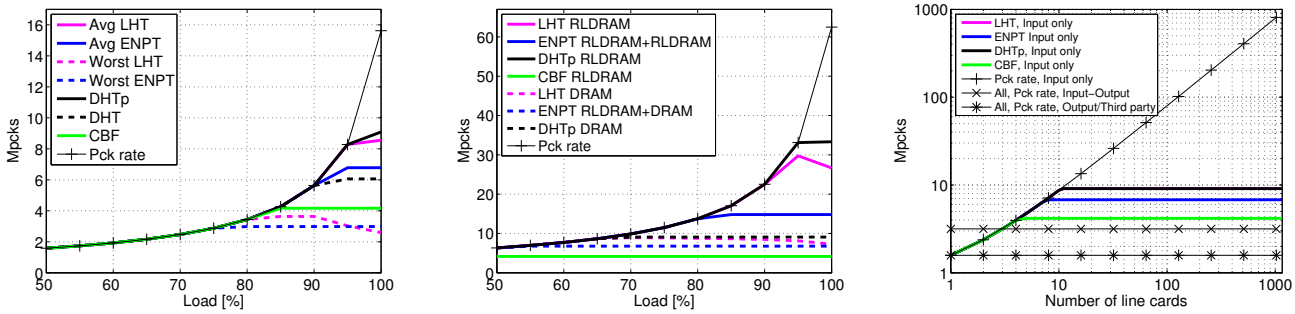
In a ENPT-based PIT, each operation starts at the root of the trie and proceeds iteratively along the tree until a leaf node is reached or it is not possible to further proceed: it follows each PIT operation requires a number of accesses to memory that is linear with the number of components in a content name. Recall that a ENPT-based PIT also require an external hash-table to store PIT tuples: it follows that the memory footprint of a ENPT-based PIT is the size of the ENPT plus this hash-table. Since no details is further provided on which hash-table should be used in [6], we assume either LHT or DHT for the reasons discussed above. Finally, two additional accesses to memory are required to retrieve/update/remove an element from the hash-table once the node in the ENPT is found.

IV. Evaluation

In this section, we first numerically evaluate the different placements and data structures for PIT. Then, we select the most promising data structure and implement it on a Cavium Octeon network processor [2]. We use this implementation to evaluate PIT's performance on real hardware.

A. Numerical

We start by comparing the memory footprint of the following data structures for PIT: counting Bloom filter (CBF),



(a) 10 Gbps line-card. (b) 40 Gbps line-card. (c) 10 Gbps line-card.

Fig. 1. Numerical evaluation ; Interest packet size = 80 Bytes ; Data packet size = 1,500 Bytes.

# of entries	62K	252K	8M	30M
CBF	584 (584) KB	2.3 (2.3) MB	71 (71) MB	273 (273) MB
LHT	2.1 (4.1) MB	8.8 (16) MB	265 (500) MB	1 (1.9) GB
DHT(p)	3.9 (5.8) MB	15.6 (23) MB	483 (717) MB	1.8 (2.7) GB
ENPT	N.A.	2 (2) MB	30 (30) MB	N.A.

TABLE I
MEMORY FOOTPRINT ; CBF, LHT, DHT(P) AND ENPT ; N=[100K,1M,10M] ; L=[20,50]B (VALUES FOR 50B ARE IN BRACKETS).

linear-chained hash-table (LHT), d-left open-addressed hash-table with sequential (DHT) and parallel access (DHTp) to sub-tables, and encoded name prefix trie (ENPT). Our goal is to understand which memory should be used for each data structure. Remember that today's largest on-chip memory, SRAM, has a size of 4.25 MB and access time of 1 ns; off-chip memory can be SRAM (access time 4 ns, size 25 MB), RLDRAM (access time 15 ns, size 250 MB) or DRAM (access time 55 ns, size 10-100 GB) [11].

Table I compares the memory footprint of all data structures as a function of the number of PIT entries n and length of the content names l. For LHT, DHT, DHTp and CBF we derive the numbers using the formulas presented in Section III-C; for ENPT, in absence of analytical formulation we use values derived from their experimental evaluation [6]. Please note that DHT and DHTp have the same memory footprint. We set n=[62K,252K,8M,30M]: <62K;8M> and <252K;30M> are the values for n derived assuming a load of 50,100% (flow balance and worst case) at 10 and 40 Gbps, respectively (cf. Section III-A); we set l=[20,50] Bytes as in [6], and we refer to these lengths as "short" and "long". For minimum number of entries, n=62k, all data structures fit on on-chip memory, though DHT(p) only fit if we assume short content names. For n=252k, only CBF and ENPT can be stored on on-chip memory, instead. Overall, no data structure fits on on-chip memory when we assume n=8-30 Millions, worst cases at 10 and 40 Gbps. Since the PIT is a critical element of a content router, we believe it should be dimensioned to support a worst case; accordingly, we conclude that ENPT and CBF fit on RLDRAM while LHT and DHT(p) only fit on DRAM.

We now compute for each <data structure, memory> pair how many packets per second it can handle. As CBF does not support timers, to be fair we do not consider additional probes required to purge expired entries (cf. Section III-C). We assume Interest packets of 80 bytes and Data packets of 1,500 bytes carried over a 10 Gbps link. For LHT and ENPT, we also differentiate between "average" and "maximum" cases: for LHT, this refers to the average and maximum chain length, while for ENPT this refers to average and maximum number of components per content name. For the other data structures, we do not differentiate as they are not impacted by chaining or by number of components per content name. We set the maximum number of component to 15 as in the traces used in [6]. Figure 1(a) shows the number of packets each solution can handle as a function of load; the Figure also shows as a baseline the number of packets per second, both Interest and Data, received at the line-card (Pck rate).

Figure 1(a) shows that in a flow balanced scenario, load=50%, all data structures can handle the target packet rate of 1.5 Mpcks. As the load increases, each data structure can only serve a fraction of the packet rate: for example, ENPT sustains on average a load up to 90% with no penalties, whereas DHTp handles a load up to 95%. LHT also sustains a load of 95% on average. However, in the maximum case LHT and ENPT only handle a load up to 80 and 75%, respectively. CBF does slightly better as it sustains a load up to 85%. From this analysis, we conclude that DHT and DHTp have the best performance as they sustain the highest load, and thus largest number of packets per second, while providing determinism, *i.e.,* average performance that matches the maximum case one.

Figure 1(b) investigates the performance of each data structure over a 40 Gbps link. For LHT and ENPT, we only consider the average case, while for DHT we only consider its parallelized version, DHTp. In addition, we speculate a scenario where LHT, DHTp and ENPT's hash-table are implemented over RLDRAM; despite such large RLDRAMs do not exist today (cf. Table I), the rationale is that they might exist in the future. As expected, Figure 1(b) shows that RLDRAM largely improves the performance of each data structure. For example, DRAM-based DHTp peaks at about 9 Mpcks (better visible in Figure 1(b)) which in a 40 Gbps link is only enough to sustain a load of 65%; instead, RLDRAM-based DHTp can still handle a load of 95% even in a 40 Gbps link, *i.e.,* about 32 Mpcks. Similarly, all-RLDRAM-ENPT sustain a load up to 80%, whereas when DRAM is used for its hash-table ENPT, as CBF, cannot even sustain the flow balanced scenario.

	Memory	Mpcks <avg,min>	Timer	Loop	Placement
LHT	DRAM	(8.5,3.6)	Yes	Yes	All
DHT	DRAM	(6.1,6.1)	Yes	Yes	All
DHTp	DRAM	(9,9)	Yes	Yes	All
CBF	RLDRAM	(4.2,4.2)	No	No	Input-only
ENPT	RLDRAM	(6.7,3)	Yes	Yes	All

TABLE II
COMPARISON OF DATA STRUCTURES FOR THE PIT

Table II summarizes the results and analysis of the PIT data structures. We conclude that DHT and its optimization DHTp are the best data structures for PIT; in fact, they both support all required features as well as possible placements, while achieving highest speed for both average and worst cases.

We finally evaluate the PIT's placements while coupled with the data structures. Figure 1(c) (log-log scale) shows the number of packets per second each line-card has to process as a function of the number of line-cards and PIT placement. We consider a flow balanced scenario in a 10 Gbps link, *i.e.,* 1.5 Mpcks on the wire. Figure 1(c) shows that the input only placement requires each line-card to process a number of packets that grows exponentially with the number of line-cards. This is expected as every Data packet has to be broadcasted to all other line-cards. It follows that a maximum of 4, 8 and 10 line-cards can be supported if the PIT is implemented using CBF, ENPT, and LHT/DHTp, respectively. Conversely, for all remaining PIT's placements the amount of packets to process is independent from the number of line-cards. It is worth noticing that the input-output placement doubles the number of packets each line-card has to perform, whereas output-only and third party placement does not add any additional burden to the line-cards.

Based on these results and the discussion in Section III-B, we conclude that the third party placement has the best performance: in fact, it requires a single probe to the data structure to realize all PIT operations, while enabling support for timers, multipath, Interest aggregation and loop detection.

B. Implementation

This section evaluates a proof of concept DHT-based PIT that we implemented on a Cavium Octeon Network processor (NP), a software-programmable device optimized for networking applications, that is widely used on router line-cards [1]. Our reference NP board is based on a Cavium Octeon Plus CN5650 12 cores 600 MHz network processor [2] equipped with 48KB of L1 cache per core, 2MB of shared memory, 4 GB of off-chip DRAM memory, and an SFP+ 10GbE.

We do not implement the optimized version of DHT, DHTp, as our reference NP is equipped with only one memory controller that manages all memory banks, thus not allowing parallel access to the off-chip DRAM. In addition, we do not implement any further hash-table optimization technique, as bucket prefetching or summary bloom filters [12], [8], or ad-hoc solutions for multi-core architectures,e.g., [10]. We plan to investigate these solutions as future work.

We evaluate our proof of concept for a DHT-based PIT assuming a load=50-80% at 10 Gbps, with a PIT size of 62K and 1M entries respectively. We generate Interest packets of 80 bytes and Data packets of 1,500 bytes; we assume short content names (20 Bytes) that are inserted at the beginning of the IP payload. With only 3 cores our prototype successively handles all received packets in the flow balanced scenario, *i.e.,* load=50% and 1.5 Mpcks, while 8 active cores are required when the load goes up to 80%, *i.e.,* 3.4 Mpcks.

V. CONCLUSIONS AND FUTURE WORK

Named Data Networking (NDN) is today's most complete solution for Information-Centric Networking, a novel networking paradigm centered around information or content. In this work, we focus on the design and implementation of the Pending Interest Table (PIT), the table where Interest packets, or content requests, are aggregated to enable multicasting as well as symmetric routing, two core features of NDN. We make the following contributions. First, we define a spectrum of candidate designs for PIT, focusing on its placement within a router as well as on its data structure. Then, we numerically evaluate each design with respect to PIT's requirements. Finally, we implement the most interesting design on a 10 Gbps network processor and evaluate its performance. The main outcome of the evaluation is encouraging, as our PIT's prototype can handle about 1 Million entries assuming a wire speed of 10 Gbps. In the near future, we plan to extend our proof of concept implementation to a full-fledged prototype.

ACKNOWLEDGMENTS

This work has been partially funded by the French national research agency (ANR), CONNECT project, under grant number ANR-10-VERS-001.

REFERENCES

[1] Alcatel-lucent fp3. http://www.alcatel-lucent.com/products/fp3/.
[2] Cavium. http://www.cavium.com/.
[3] Meridian. http://www.cs.cornell.edu/.
[4] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. A Survey of Information-Centric Networking. 2011.
[5] G. Carofiglio, M. Gallo, and L. Muscariello. Joint hop-by-hop and receiver-driven interest control protocol for content-centric networks. In *ICN'12*, Helsinki, Finland, 2012.
[6] H. Dai, B. Liu, Y. Chen, and Y. Wang. On pending interest table in named data networking. In *ANCS'12*, Austin, Texas, USA, 2012.
[7] V. Jacobson, D. K. Smetters, J. D. Thronton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Network Named Content. In *CoNEXT'09*, Rome.
[8] A. Kirsch, M. Mitzenmacher, and G. Varghese. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*, pages 181–218. Springer, 2010.
[9] Z. Li, J. Bi, S. Wang, and X. Jiang. Compression of pending interest table with bloom filter in content centric network. In *CFI'12*, Austin, Texas, USA, 2012.
[10] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: a cache-partitioned hash table. *SIGPLAN Not.*, 47(8):319–320, Feb. 2012.
[11] D. Perino and M. Varvello. A reality check for content centric networking. In *ICN'11*, Toronto, Canada, Aug. 2011.
[12] W. So, A. Narayanan, D. Oran, and Y. Wang. Toward fast ndn software forwarding lookup engine based on hash tables. In *ANCS'12*.
[13] M. Varvello, D. Perino, and J. Esteban. Caesar: a content router for high speed forwarding. In *ICN'12*, Helsinki, Finland, 2012.
[14] M. Xie, I. Widjaja, and H. Wang. Enhancing cache robustness for content-centric networking. In *INFOCOM'12*, Orlando, FL, USA.
[15] W. You, B. Mathieu, P. Truong, J.-F. Peltier, and G. Simon. DiPIT: a distributed bloom-filter based PIT table for CCN nodes. In *ICCCN'12*, July 2012.
[16] H. Yuan, T. Song, and P. Crowley. Scalable NDN forwarding: Concepts, issues and principles. In *ICCCN'12*, 2012.