High Throughput Forwarding for ICN with Descriptors and Locators

Michele Papalini, Koorosh Khazaei, Antonio Carzaniga, Daniele Rogora
Faculty of Informatics
Università della Svizzera italiana (USI)
Lugano, Switzerland

ABSTRACT

Application-defined and location-independent addressing is a founding principle of information centric networking (ICN) that is inherently difficult to realize if one also wants scalable routing and forwarding. We propose an ICN architecture, called TagNet, intended to combine expressive application-defined addressing with scalable routing and forwarding. TagNet features two independent delivery services: one with application-defined and possibly location-independent content descriptors, and one with network-defined host locators. In this paper we develop and evaluate specialized forwarding algorithms for TagNet. We then implement and combine these algorithms in a forwarding engine built on a general-purpose commodity CPU, and show experimentally that, thanks to the dual addressing, by descriptor or by locator, this engine can achieve a throughput of over 20Gbps with large forwarding tables corresponding to hundreds of millions of users.

CCS Concepts

•**Networks** → **Data path algorithms;** *Naming and addressing; Network layer protocols;*

Keywords

ICN; forwarding; algorithms; locators; content descriptors

1. INTRODUCTION

Addressing by name is a defining feature of information centric networking that poses a crucial trade-off. On the one hand, the network would serve applications better if applications were allowed to choose names to refer directly to information objects or other entities, such as servers or users, without necessarily referring to the network hosts where those entities are located. On the other hand, such application-defined and possibly location-independent addresses may aggregate poorly, which fundamentally limits the scalability of the network.

ANCS '16, March 17-18, 2016, Santa Clara, CA, USA

DOI: http://dx.doi.org/10.1145/2881025.2881032

One solution could be to restrict applications to use names with globally routable prefixes similar to DNS names. This is the approach taken, for example, by the CCN architecture. The data plane would still be more complex than IP, since it requires longest *name* prefix matching, but fortunately there are good solutions for that [7, 12, 13] and in any case the approach would improve scalability, since names would be forced to aggregate. However, from the application's perspective, this solution amounts to turning names back into network-chosen addresses. And even though the notion of ICN could be beneficial in other ways, those benefits might not be worth a radical redesign of the Internet.

We believe that a radically new network should offer a richer communication service. So our goal is to design an information centric network that allows applications to choose meaningful names, and our challenge is to engineer routing and forwarding systems that can cope with such application-chosen names. In prior work, we developed our design with a network architecture called *TagNet* that, among other things, supports application-defined addressing. In the same prior work we also proposed and evaluated a scalable routing scheme for TagNet [6]. In this paper we develop the TagNet data plane. In particular, we build a software forwarder (matcher) on inexpensive general-purpose CPUs and show that such a forwarder can sustain a throughput of over 20Gbps even with a large forwarding information base.

At the architectural level, our approach is first to disentangle two network functions typically embodied in a single name. Names are supposed to allow applications to refer to content and application-level entities, and at the same time they are the basis for routing and forwarding and therefore they are supposed to allow the network to locate and reach hosts. This double duty is problematic, so we designed a network supporting two types of addresses: one based on application-chosen and typically location independent *content descriptors*, which are intended to be expressive for applications although potentially expensive for routing; and one based on network-chosen *host locators* that are instead very efficient for routing but meaningless to applications.

In particular, we designed TagNet to support content descriptors consisting of sets of tags. For example, a music server might advertise its collection with the descriptor {*concert*, *classical*} and a music player might request a file for download with the descriptor {*classical*, *Beethoven*, *ninth*, *concert*, *360bps*}, and TagNet would forward the request *by descriptor* to the server. We then developed a routing scheme for TagNet that supports descriptors as well as very efficient locators based on a compact routing scheme

by Thorup and Zwick [10]. Now we turn to forwarding, by descriptor or locator.

Forwarding a packet using its descriptor D_p (a set of tags) amounts to finding one or more subsets of D_p (possibly the largest subset) in a descriptor FIB containing many tag sets. This is analogous to longest-prefix matching in IP forwarding or to longest name prefix matching in name-based ICN, except that *subset* matching is fundamentally more expressive and therefore also more complex than prefix matching. Our contribution is a descriptor matching algorithm that performs well in practice, especially within a network that supports efficient host locators.

We start by representing the descriptor forwarding table with a trie over which we implement two subset-matching algorithms: *Find All Subsets* and *Find Largest Subset*. We then introduce specific improvements to speed-up both algorithms. We do that by first compressing the trie and then by laying out its structure in memory so as to maximize locality in the memory access patterns of the two algorithms. We then implement and fine-tune the forwarding algorithm for the Thorup and Zwick locators, and finally we integrate descriptor-based and locator-based forwarding algorithms in our TagNet forwarding engine.

We evaluate our TagNet forwarding engine under a variety of workloads. This evaluation shows that the forwarding engine is efficient, and in particular that the descriptor-based matching algorithms perform well under different workloads, and scale well with the size of the FIB as well as with the number of threads. We also show that, thanks to the separation between descriptors and locators, the TagNet forwarding engine can achieve a high throughput, since most of the traffic can be forwarded based on extremely efficient network-defined locators. For example, in the music download flow exemplified above, except for the very first request packet routed by descriptor, all the follow-up data and request packets exchanged between the music server and the music player can be forwarded by their locators. In such mixed flows, the forwarding engine takes advantage of the excellent performance of locator-based matching, exceeding 20Gbps of throughput even with large FIBs.

2. BACKGROUND

We now briefly describe the TagNet architecture, focusing on the formulation of the problems related to forwarding. We then review existing algorithms and systems that solve the same or related problems.

2.1 TagNet: Descriptors and Locators

TagNet is an ICN architecture designed to support push and pull flows with two independent addressing and delivery methods: one with content descriptors, one with host locators (hereafter simply *descriptors* and *locators*). Descriptors are sets of tags and are chosen by applications. Locators are opaque bit strings and are chosen by the network. More specifically, the network allows an application to (1) advertise one or more descriptors $A_1, A_2, \ldots$; (2) obtain the application's own network-assigned locators (one or more); (3) send a packet addressed by descriptor B to up to k applications that advertise a matching descriptor $A_i \subseteq B$; (4) send a packet addressed by locator to the corresponding application. The fan-out limit k, which controls the delivery by descriptor (3), can be used to obtain an anycast ($k = 1$) or multicast ($k = \infty$) delivery semantics.

With these primitive operations, applications can communicate in various ways. For example, this is how a consumer C could "pull" data from a producer P in a pattern typical of ICN architectures such as CCN and NDN: producer P advertises a content object x with a descriptor D_x; consumer C obtains its own locator L_C and then sends an "interest" packet I addressed by descriptor D_I with fan-out limit $k = 1$ (i.e., anycast) and containing C's locator L_C; assuming D_I matches D_x ($D_I \supseteq D_x$), meaning that object x satisfies the interest expressed by consumer C, the network delivers the interest packet I to producer P; producer P obtains its own locator L_P and sends a "data" packet, addressed by locator L_C, containing its locator L_P and the requested object x (or a chunk of x); the network delivers the data packet to consumer C. At this point consumer C knows the locator of the producer, L_P, so C can send follow-up requests (e.g., for other chunks of data) by addressing P directly by locator.

Consumer C could also use a fan-out limit $k > 1$ to try to obtain the data from multiple producers at the same time, or to choose one among many producers. Also, switching roles, producer P could "push" content to C, and similarly a producer could also push data to multiple consumers, either directly pushing the data or by pushing a request to pull.

Descriptors are strictly more expressive than hierarchical names (as in CCN or NDN). In fact, we can emulate the semantics of a hierarchical name with a descriptor by enumerating the components of the name as separate tags. For example, the name */org/gnu/software/* is equivalent to the descriptor {*1:org*, *2:gnu*, *3:software*}. In Section 3.1.3 we describe a specific forwarding algorithm that can also emulate the semantics of longest name prefix matching.

2.2 Forwarding in TagNet

In the TagNet data plane, we need to implement a forwarding engine that supports both locator- and descriptor-based forwarding. We describe the locator-based forwarding implementation later in Section 3.3, where we give an overview of the Thorup and Zwick routing scheme. In this section we focus on descriptors.

Descriptors expressed as sets of string tags may not be ideal for network-level packet forwarding. So, in TagNet we use Bloom filters to compress descriptors into fixed-size bit vectors, which is what we then use for forwarding at the network level. Notice that an individual descriptor is a relatively small set of tags, so the corresponding Bloom filter can also be small. Notice also that the inclusion relation between two Bloom filters (bitwise) corresponds to the subset relation between the tag sets that they represent, except for a controllable small false-positive probability. Concretely, based on a conservative estimate on the typical number of tags in descriptors, we choose to use Bloom filters of 192 bits and 7 hash functions (see Papalini et. al [6] for details).

In summary, a packet addressed by descriptor carries a Bloom filter B and a fan-out limit k, and a router's FIB maps Bloom filters $A_1, A_2, \ldots$ to interfaces. With that, forwarding amounts to finding at most k entries $A_i \subseteq B$ in the FIB. This subset lookup problem is equivalent to and therefore also known as the partial matching problem.

2.3 Forwarding in ICN

The problem of forwarding in ICN has been considered mostly if not exclusively for hierarchical name-based addressing, and more specifically in forwarding interest pack-

ets in CCN or NDN. In these cases, forwarding amounts to longest prefix matching (LPM) or more specifically longest name prefix matching (LNPM).

One of the first systems for high-throughput interest forwarding was developed by Wang et al. [12] and exploits the high-parallelism of a GPU. Wang et al. implement LPM on a character trie compressed in a data structure called multi-aligned transition array (MATA). Wang et al. report a throughput of 63.52Mpps with a FIB of 10 million entries (we later refer to this FIB as 10M-CCN). However, this system also incurs high latency [7], which is a common problem for GPU-based systems.

Most ICN forwarding systems implement LNPM with a hash-table: the algorithm stores the FIB in a hash table and, for an input name of ℓ components, proceeds by first looking up the prefix of length ℓ (the whole name), then the prefix of length $\ell - 1$, and so on. On this basis, authors have built a number of variants. For example, Perino et al. use Bloom filters to group multiple prefixes [7], Wang et al. search prefixes in an order based on the distribution of prefix lengths [11], and Yuan and Crowley use a binary search on multiple hash tables organized by prefix lengths [13]. These systems achieve a high throughput in the tens of million packets per second on the 10M-CCN FIB, although notably Yuan and Crowley evaluate their system on a FIB of 1 billion entries, which is the largest FIB used so far to test an interest forwarding algorithm for ICN.

In addition to interest forwarding, CCN and NDN also require a suitable implementation of the pending-interest table (PIT) to forward *data* packets. We know of only two systems that support both forwarding functions. The first, proposed by So et al. [9], is based on hash tables and achieves 8.8Mpps throughput on mixed traffic (interest and data packets) and with FIBs of 64 million entries. Another one is BFAST by Dai et al. [2], and consists of a unified index that supports LNPM in the FIB and exact-match in the PIT and content store, and that can achieve a throughput of up to 81.32Mpps depending on the incoming traffic mix.

The *eXpressive Internet Architecture* (XIA) is a proposed new Internet architecture [3] that supports a notion of flexible addressing that, although not designed specifically for ICN, relates to forwarding in ICN. In essence, flexible addressing in XIA means that each packet may specify multiple addresses of different kinds together with a directed acyclic graph that expresses alternatives or "fallback" relations between those addresses. Based on the partial order defined by the graph, a router then selects the most appropriate address it is capable of using for forwarding.

XIA provides a nice framework within which we could realize the dual addressing of TagNet, by defining descriptors and locators as two types of addresses. However, notice that XIA and its fallback mechanism do not provide specific support for descriptors or locators. Furthermore, the duality of locators and descriptors in TagNet is rather different from, and to some extent *incompatible* with the flexible addressing of XIA. This is because locators are not intended to be fallback addresses for descriptors, or vice-versa, and even though a fallback relation may make sense in some cases, both locators and descriptors are intended to be globally routable in TagNet. Still, XIA is flexible enough to accommodate the dual addressing of TagNet, and possibly to extend it with alternative types of descriptors and locators.

2.4 Subset Matching

The systems discussed so far relate to our work because of the domain (forwarding in ICN) and the nature of the solution (algorithmic, general-purpose multi-core CPU). But notice that they solve a different combinatorial problem, namely prefix matching instead of subset matching. We now review existing results for subset matching.

The theory on subset matching is well established, although unfortunately not very useful in our case. Recall that we are given n sets $A_1, A_2, \ldots, A_n \subseteq \{1, \ldots, m\}$ and a query set $B \subseteq \{1, \ldots, m\}$ (specifically $m = 192$) and we want to find all (or up to k) sets A_i such that $A_i \subseteq B$. There are two trivial solutions: one stores the answers for all possible queries in an index, and requires $O(m)$ time but a prohibitive $O(2^m)$ space; one scans the sets linearly, which requires linear (minimal) space but also linear time $O(nm)$. The first non-trivial improvements are due to Rivest [8], one regarding the index solution that is irrelevant for us, and one based on a trie that is quite similar to the one we develop in this paper. Charikar et al. also propose two improvements over the trivial solutions [1] but their results are not useful in practice, as they either require too much memory or are too slow for a real implementation of a matching engine.

Subset matching has concrete applications in many fields, for example in networking for packet classification, and in information retrieval to search documents containing a given set of words. Most relevant for our purposes are applications in databases, where the base of sets (the FIB in our case) is typically very large. What is traditionally considered the best algorithmic solution in databases and information retrieval is an inverted index that maps each element (tag) to the list of sets the element appears in [4]. However, inverted indexes perform well when the universe of elements is large and the frequencies of individual elements are low, not when the universe is small and therefore each element appears in many sets. But this is precisely the problem we intend to solve, since the Bloom filters already reduce our universe to a small size ($m = 192$).

Perhaps the most relevant work in the database literature is a recent paper by Luo et al. [5] who propose two set-containment algorithms based on tries. The first one, called *PATRICIA trie-based signature join* (PTSJ), encodes each set of tags with a hash-based signature, and then stores the signatures in a trie. This is similar to what we do, but there are also significant differences: in essence, we use a more effective algorithm and layout, and we use more compact Bloom filters. The second algorithm, called PRETTI+, works on the actual elements of the sets (tags) and builds what amounts to an inverted index. Interestingly, Luo et al. evaluate their algorithms with real data sets. The most relevant data set, which is also more similar to our typical workload, is a collection of 3.5 million tagged photos from Flickr with an average of 5.36 tags per photo. Under this workload, PRETTI+ outperforms PTSJ with an average throughput of 17.5K queries per second on a single Intel Xeon 2.27GHz core. These results are interesting for us because we use comparable workloads with which we achieve a throughput of 140K–319K queries (packets) per second, also on a single thread, but on a collection (FIB) that is almost three times larger. Notice however that, even though Luo et al. use a hardware platform almost identical to ours, they implement their algorithms in Java while we use C++.

3. FORWARDING ALGORITHMS

In this section we present the matching algorithms we implement to realize the dual forwarding system of TagNet. We start with the algorithms for descriptor-based forwarding, which we describe in detail, and then at the end of the section we briefly present the algorithm for locator-based forwarding. We do not spend much text to describe this algorithm simply because we implement it directly from a compact-routing scheme by Thorup and Zwick [10].

3.1 Descriptor-Based Matching Algorithms

We develop two algorithms to realize descriptor-based forwarding in TagNet: *Find All Subsets* (FAS) and *Find Largest Subset* (FLS). Both algorithms operate on the same data structure that represents the FIB. We now describe this structure and then detail each algorithm. Recall that we encode descriptors (tag sets) as Bloom filters. Therefore, for the purpose of forwarding and throughout this section, the term *descriptor* refers to the Bloom filter that encodes the descriptor.

3.1.1 FIB Data Structure

We use a prefix trie to store all the descriptors in the FIB. The FIB represents a relation between descriptors and interfaces, therefore we link each full descriptor from the prefix trie to a list of output interfaces. Figure 1 shows an example of this basic data structure. To be more precise, in TagNet we route packets on multiple trees [6], so the FIB represents a relation between filters, interfaces, *and trees.* The FIB is still indexed by descriptor with a prefix trie, but each full descriptor in the trie links to a list of tree–interface pairs. This structure is only relevant for the very last phase of the forwarding algorithm, which has little if any impact on performance. Therefore, for the purpose of this paper and for simplicity, we ignore trees altogether.

Descriptor		Out ifx
Bit String	1s Pos	
1000100000	(1,5)	i_2
1010000100	(1,3,8)	i_4,i_2
0110100000	(2,3,5)	i_3
0011100010	(3,4,5,9)	i_6,i_2
0010101000	(3,5,7)	i_5,i_2
0000100100	(5,8)	i_2

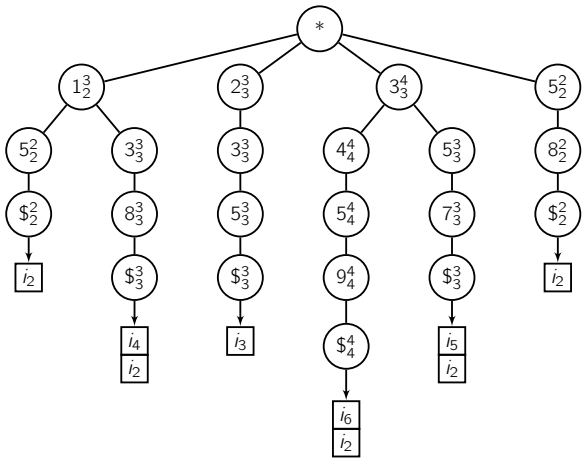

Figure 1: FIB representation using a prefix trie

Each path in the trie from the root to a leaf represents a descriptor in the FIB. In particular, the trie represents each descriptor as a sequence of positions (from 1 to 10 in the figure, from 1 to 192 in the real implementation) corresponding to the bits set to one in the descriptor (1-bits). Thus each node in the trie represents either the position of a 1-bit in one or more descriptors, or a terminator node, marked with the '$' position in the figure, that represents a full descriptor and that links that descriptor to the output list. For example, the FIB depicted in Figure 1 contains descriptor 0011100010 represented as the sequence $3, 4, 5, 9$ in the trie and associated with interfaces i_6 and i_2.

In each node we also store the maximum and minimum depths reachable through that node. Depth values exclude the terminator nodes, and therefore represent the Hamming weight of a descriptor (the number of 1-bits). In Figure 1 we write the maximum and minimum depths with superscripts and subscripts, respectively. For example, the node marked 1_2^3 is the root of a subtree representing a set of descriptors of Hamming weights between 2 and 3.

Given the descriptor trie, matching an incoming packet with descriptor B amounts to finding one or more descriptors in the trie that are subsets of B. This in essence can be done by performing a walk over the prefix trie guided by the 1-bits in B. This algorithmic structure is the basis for the two algorithms we develop: *Find All Subsets* and *Find Largest Subset*.

3.1.2 Find All Subsets

The forwarder invokes the *Find All Subsets* (FAS) algorithm to forward packets with fan-out limit $k > 1$, and in particular multicast packets with $k = \infty$. FAS takes an input packet p and the fan-out limit k, and in its basic form visits only nodes representing bit positions that correspond to a 1-bit in the packet descriptor ($p.descriptor$). The FAS algorithm is listed as Algorithm 1.

Algorithm 1 *Find All Subsets* (FAS)

Input: packet p, fan-out limit k, FIB trie *root*
Output: set of output interfaces

```
nodes_to_visit ← {root}                    // stack of nodes
out ← ∅
while nodes_to_visit ≠ ∅ and |out| < k do
  n ← pop node from nodes_to_visit
  if n is a terminator node then           // subset found
    out ← out ∪ n.interfaces               // up to |out| = k
  else if p.descriptor[n.pos] is a 1-bit then
    for all children c of node n do
      push c onto nodes_to_visit
    end for
  end if
end while
return out
```

FAS walks through the trie in depth-first order using an explicit stack of nodes to visit. When the algorithm reaches a terminator node, which means that there is a matching descriptor, the algorithm processes the list of interfaces associated with that descriptor, adding those interfaces to the set of output interfaces for the input packet. The walk terminates immediately if the set of output interfaces reaches a size greater than or equal to the fan-out limit k.

3.1.3 Find Largest Subset

When the fan-out limit is $k = 1$, which is typically used to send anycast packets such as data or service requests, the forwarder uses *Find Largest Subset* (FLS). This specialized algorithm is intended to select the *most relevant* descriptor among a potentially large number of matching descriptors. FLS is also intended to provide the semantics of longest name prefix matching with TagNet's descriptors. The FLS algorithm is listed as Algorithm 2.

Algorithm 2 *Find Largest Subset* (FLS)

Input: packet p, FIB trie *root*
Output: maximal FIB entry matching $p.descriptor$

```
nodes_to_visit ← {root}                    // stack of nodes
best ← null                  // leaf node of best match found
while nodes_to_visit ≠ ∅ do
  n ← pop node from nodes_to_visit
  if best is null or n.max_depth > best.max_depth then
    if n is a terminator node then          // subset found
      best ← n
    else if p.descriptor[n.pos] is a 1-bit then
      for all children c of node n do
        push c onto nodes_to_visit
      end for
    end if
  end if
end while
return best
```

The FLS algorithm is conceptually similar to the FAS algorithm, and therefore FLS also performs a trie walk limited to matching descriptors. The main difference is that FLS keeps track of the matching descriptor with maximal Hamming weight found during the walk (variable *best*). Thus FLS further limits the trie walk to the subtrees that may contain matching descriptors larger than the current best.

3.1.4 Basic Algorithmic Improvements

Having defined the high-level structure of the descriptor matching algorithms, we now introduce three additional pruning strategies to further reduce the complexity of the trie walk. These strategies are based on the Hamming weight of the descriptors.

The first strategy exploits the fact that we search for subsets, so the Hamming weight of the descriptor in the packet must be greater than or equal to the Hamming weight (or depth) of a matching descriptor in the FIB. Therefore, we can check the minimum depth of a node before we push the node onto the stack, and in case the minimum depth exceeds the Hamming weight of the descriptor in the packet, we can safely skip that node.

The second strategy is a refinement of the first one. Before pushing a node c onto the *nodes_to_visit* stack, we compute the number of remaining 1-bits in the input descriptor to the right of the bit position of node c ($c.pos$). We then add this number to the depth of node c in the trie, which represents the number of 1-bits already matched. The result is the maximum potential matching weight. That is, the maximum possible Hamming weight of any matching descriptor under node c. This also means that we can skip c when this maximum weight is less than the minimum depth of c. For example, consider matching input descriptor $(1, 3, 5)$ against the trie of Figure 1. In this case, the algorithm can skip the descendants of the node labeled 3_3^4 (third child of the root). In fact, the depth of node 3_3^4 is 1, and the number of 1-bits to the right of position 3 in the input descriptor is 1, since there is only one position left unchecked (position 5), which adds up to a maximum potential matching weight of 2, which is less than the minimum depth of 3 under node 3_3^4.

The third strategy is also based on the maximum potential matching weight, and is applicable to the FLS algorithm only. FLS remembers the maximal matching descriptor seen in the walk (*best*). Therefore the algorithm can safely skip subtrees that have a maximum potential matching weight lower than *best*.

3.2 Improvements for Memory Usage

Tries, like other linked data structures, are notoriously inefficient in their use and access of memory. A naïve implementation of the FIB described in Section 3.1.1 requires a lot of memory, and accesses that memory without locality and therefore inefficiently. We now describe how we engineer the FIB to make it efficient in terms of memory usage and access. In summary, we apply four transformations: (1) we permute the bits of the descriptors in the FIB according to their popularity, (2) we factor out node chains (lists of nodes with a single child) from the trie, (3) we implement the trie with a single compact vector, and (4) we lay out the nodes in the vector so as to improve locality in the search algorithms.

3.2.1 Bit Permutation

The first transformation we apply to the trie is global. In essence, we sort the bits in decreasing order of frequency. The effect is that we move the most popular 1-bits to the leftmost positions in the descriptors. This in turn increases the sharing of prefixes in the trie, which means that the trie contains less nodes and also that a trie walk crosses less nodes.

Bit Pos	Bit Freq	New Pos
1	2	3
2	1	5
3	4	2
4	1	6
5	5	1
6	0	9
7	1	7
8	2	4
9	1	8
10	0	10

Original Descriptors	Permuted Descriptors
(1,5)	(1,3)
(1,3,8)	(2,3,4)
(2,3,5)	(1,2,5)
(3,4,5,9)	(1,2,6,8)
(3,5,7)	(1,2,7)
(5,8)	(1,4)

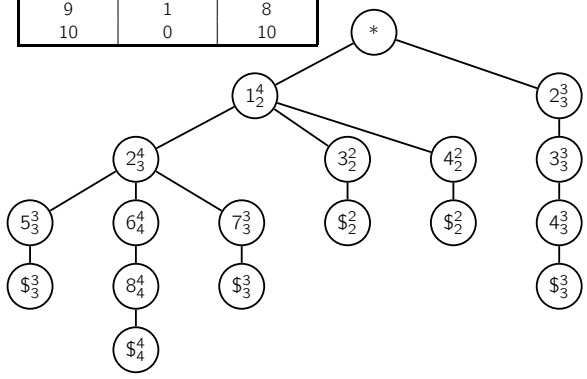

Figure 2: Trie compression with bit popularity (the original trie is the one presented in Figure 1)

Figure 2 shows the application of a bit permutation to the trie of Figure 1. The top-left table in Figure 2 shows the occurrences of each bit in the whole FIB together with the resulting bit permutation. The top-right table then shows the application of that permutation to the FIB. For example, descriptor $(2, 3, 5)$ becomes $(1, 2, 5)$, because the permuted position of 2 is 5, the new position of 3 is 2, and 5 goes to position 1. This technique is very effective and gives us an impressive compression. In order to use this technique we have to permute all the incoming queries accordingly. This overhead is linear in the number of 1-bits in the query and is negligible in practice.

Compare the new trie obtained with the permutation, represented in the lower part of Figure 2, with that of Figure 1. The permutation improves the sharing of prefixes and therefore reduces the size of the trie from 22 to 18 nodes. This improvement might seem small. However, notice that in this example the frequencies of the 1-bits in the FIBs are all small and therefore very similar, necessarily since the trie is itself very small. In large and realistic FIBs, the frequency distribution tends to be much more skewed, as can be deduced from the the popularity distributions for on-line content, which are likely to remain the same in an ICN. And that amplifies the benefits of the permutation transformation.

3.2.2 Chains Removal

To further reduce memory usage for the prefix trie, we remove chains from the data structure. With the term *chain* we indicate a sequence of nodes with a single child that amounts to a linked list of nodes. In our implementation we transform only those chains that lead to a terminator node, namely a node with position '$'. We also experimented with chain removal throughout the trie, but that proved less effective. This is because chains at the top of the trie are shared and there are far more terminal chains than intermediate ones. And crucially, operating on chains in the middle of the trie requires additional checks in the basic trie-walk algorithm that offset the cost savings of chain elimination. Conversely, terminal chains do not require additional processing in the trie walk.

Our intuition is that chains are at the same time costly in terms of memory, and also redundant for the purpose of the search algorithms. In fact, when we reach a chain we can explore only one path, meaning a single descriptor. In other words, a chain exercises the trie-walk algorithm for potentially several iterations, only to produce a simple yes/no answer: either the descriptor matches or it does not. Our approach is therefore to extract each terminal chain from the trie and to store it, together with the corresponding list of output interfaces, in a separate compact data structure that we check in the final stage of matching with a much more efficient comparison operation. Figure 3 shows an example of a terminal chain in the trie (left) and its compact external representation (right).

To remove a chain we simply move the terminator node that closes the chain to the head of the chain, and drop all the other nodes. Notice that the minimum and maximum depths are the same throughout the chain, which means that the shortened trie behaves exactly as the original trie for the purpose of the matching algorithms. We then link the terminator node to a compact vector that represents all the bit positions removed from the trie. See Figure 3 for an explanatory example.

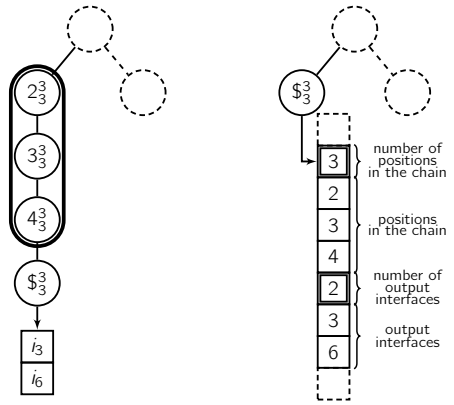

Figure 3: A chain in the trie (left) and its representation (right)

3.2.3 Vector Representation and Memory Layout

One of the main disadvantages in a naïve trie representation is the use of pointers and possibly of tables of pointers. A common way to reduce the use of pointers is first to structure the trie as a binary trie in which each node "points to" the node's first child and the node's next sibling, and then to represent this binary trie within a vector so as to remove one of the two pointers by implicitly linking a node with the following one in the vector.

Figure 4 exemplifies this method by showing the representation of the trie of Figure 2 after chain removal as a binary trie (top), and then the vector representation of the binary trie (bottom). We draw the binary trie with solid arrows to denote first-child links and with dashed arrows to denote next-sibling links.

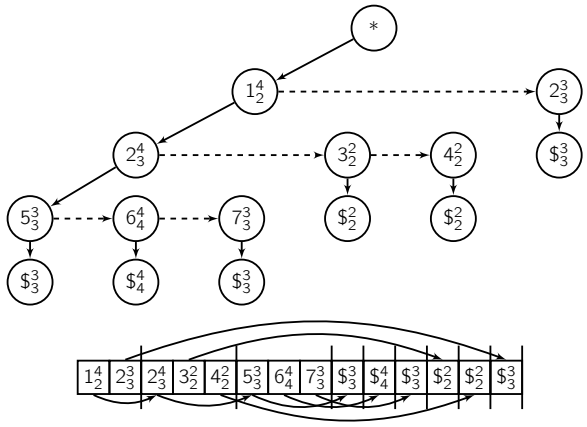

Figure 4: Trie represented as a vector

The binary trie can then be encoded as a vector in which the next-sibling links are implicit in the sequence of nodes, and the first-child links are represented as offsets in the vector. In particular, each node within the vector holds the same data as in the trie (position, minimum and maximum depths) plus an offset to point to the first child, which we depict with a solid arrow in the figure. This, however, is not enough to represent and walk through the trie, since the

walk algorithm would know how to start iterating through the children of a given node, but it would not know where to stop. So, to support the trie walk, we add a flag to a node that indicates whether that node is the last child, or in other words the last node in a contiguous sequence of siblings. In Figure 4 we indicate that a node is a last child with a vertical bar.

Having defined the vector representation of the FIB, we also adapt the search algorithms to that representation. Algorithm 3 is the specialization of the *Find All Subset* algorithm that operates on the vector FIB. The structure is the same for the *Find Largest Subset* algorithm.

Algorithm 3 *Find All Subsets* on FIB vector

Input: packet p, FIB vector *FIB*
Output: set of output interfaces

```
nodes_to_visit ← {FIB[0]}                    // stack of nodes
out ← ∅
while nodes_to_visit ≠ ∅ and |out| < k do
  n ← pop node from nodes_to_visit
  while true do          // iteration through n's children
    if n is a terminator node then          // subset found
      out ← out ∪ n.interfaces              // up to |out| = k
    else if p.descriptor[n.pos] is a 1-bit then
      push first child of node n onto nodes_to_visit
    end if
    if n is a last child then
      break out of the loop                 // go to next-node loop
    end if
    n ← next node in the FIB vector
  end while
end while
return out
```

The vector representation we just described is generally not unique (for a given tree). In fact, there are different ways to lay out the nodes in the vector that, in combination with the search algorithm, would induce different memory access patterns. In the following we study two node layouts and we compare them through a simple example.

Figure 5 shows the memory access pattern for the FAS algorithm for two different layouts of the vector. In this example we trace the nodes that the algorithm visits when matching the input descriptor $(1, 2, 4, 7)$. In particular, we depict the pointers of the trie structure (first child) with thin gray arrows below the vector, while we depict the sequence of memory accesses with bold black arrows above vector. Under each vector we also represent the evolution of the stack during the search. The values in the stack are the indexes of the vector cells indicated over each vector.

The first node-layout in Figure 5 is the same layout shown in the example of Figure 4. We obtain this layout by storing the descendants of a list of sibling nodes in the same order of the siblings, which is by increasing bit positions. Therefore we call this the *Sibling Order Layout* (SOL).

More specifically, we always store a list of siblings in increasing order of bit position. For example, we store the two sibling nodes in the first level of the trie of Figure 4, 1_2^4 and 2_3^3, at position 1 and 2, respectively. But then we have to choose how to lay out the children of these two nodes, and their children, etc. In the SOL layout we do it left-to-right and with immediate recursion, so we start by visiting the

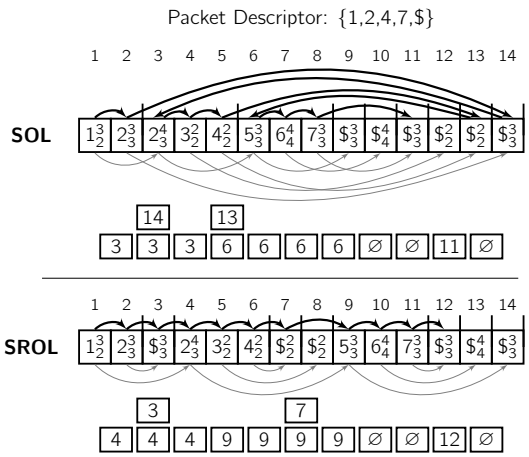

Figure 5: Different node layouts and related memory access pattern

children of the first sibling node, and then recursively their children until we lay out the entire subtree rooted at the first sibling, and then we proceed in order of bit position with the subtrees of the second sibling, the third, etc. So, as shown in the top part of Figure 5, we first add the children of node 1_2^4, at positions 3,4,5, then the entire subtree rooted at 1_2^4, and then the child of node 2_3^3 at position 14.

The second layout, shown at the bottom of Figure 5, uses the reverse ordering, and hence we call it *Sibling Reverse Order Layout* (SROL). As usual, we store sibling nodes in consecutive cells sorted by bit position. However, contrary to SOL, with the SROL layout we visit the children of sibling nodes in reverse order, starting from the last sibling node. For example (see the lower part of Figure 5) after nodes 1_2^4 and 2_3^3, we start with the children of 2_3^3, and recursively their children and the entire subtree rooted at 2_3^3, and after that we lay out the children of node 1_2^4 and its entire subtree.

In our implementation we use this latter layout (SROL), because it induces a much more linear and therefore cache-efficient memory access pattern. In fact, as shown in Figure 5, the more natural SOL layout defines an almost random memory access pattern. We first visit nodes 1 and 2, then jump to position 14, then back to position 3, and so on. Instead, the SROL layout produces a nicely sequential memory access pattern that is optimal for caching.

3.3 Locator-Based Matching Algorithm

We now provide an overview of the labeling and forwarding scheme that we use for locators. Labeling is the process by which the network assigns locators and locator FIBs to nodes. In their seminal paper on stretch-3 compact routing schemes for general graphs [10], Thorup and Zwick also propose a lesser-known but still practical compact routing scheme for trees. Since we use trees as a basic routing structure for TagNet, we adopt this Thorup and Zwick scheme for locator-based forwarding, and therefore we refer to the implementation of our locators as TZ-labels. Here we review the scheme only very briefly, without going through the details of the labeling algorithm and its associated forwarding algorithm, and we refer our readers to Section 2 of Thorup and Zwick [10].

The most compact scheme for trees by Thorup and Zwick uses both labels and FIBs of $(1 + o(1)) \log_2 n$ bits. We use a simpler and less compact version of that scheme (described in Section 2.1 of their paper) that uses $3.4 \log_2 n$ bits. Along with the labeling algorithm for that scheme, Thorup and Zwick provide a very efficient forwarding algorithm, which is what we use in TagNet and that we reproduce in Figure 6.

To forward a message toward a destination node (on a tree) each intermediate node needs only the TZ-label of the destination node (on that tree) plus its own TZ-label, which also serves as the node's FIB.

```
1  struct TZ_label {
2      uint16_t node_id;
3      uint16_t ifx_list;
4      uint16_t mask;
5  };
6  struct TZ_label my_label;
7  uint8_t k = leftmost_bit(my_label.mask);
8  uint16_t P[2] = {parent_interface, heavy_child_interface};
9  uint16_t f = largest_descendent_id;
10 uint16_t h = heavy_child_id;

12 int forward(struct TZ_label & dest) {
13     v = dest.node_id;
14     L = dest.ifx_list;
15     M = dest.mask;
16     return ((v >= my_label.node_id && v < h)
17                 ? (L >> k) & ((M >> k) ^ ((M >> k) −1))
18                 : P[v >=h && v <=f]);
19 }
```

Figure 6: Locator-based forwarding algorithm

Figure 6 indicates the local variables that we need to store at each node. A node stores its own label (*my_label*), which in turn contains the node identifier (*node_id*), a list of interfaces encoded in a bit string (*ifx_list*), and a mask used to extract them (*mask*). Each node also stores a constant k that indicates the size of the local mask in bits, the identifier f of the largest descendant, and the identifier h of the node's heavy child, which is the child through which it is possible to reach the majority of the descendants. In addition, a node stores a vector P that contains the interfaces where to forward packets for the parent node and the heavy child.

The forward function extracts all the information needed from the incoming TZ-label, and returns the output interface. Notice that this forwarding decision is taken in a single line of code that amounts to a handful of machine instructions. Using this scheme, TZ-labels computed for the AS-level network topology[1] of 42113 nodes and 118040 edges, are at most 46-bits long and can be represented by the TZ_label struct defined in the code of Figure 6.

4. EVALUATION

The algorithmic complexity of our subset matching algorithms conforms to an analysis of general partial-matching algorithms on search tries developed by Rivest [8]. In our setting, assuming a random FIB, the expected running time is $O(n^{h/m})$, where n is the size of the FIB, h is the Hamming weight of the input set, and $m = 192$ is the size of the Bloom filters we use.[2] For example, with an input set of four tags, a basic subset search would run in time $O(n^{0.15})$. In essence, our subset matching algorithms are efficient with reasonably small input sets, but their complexity grows with the size of the input set.

This analysis is indicative at a high level but does not consider algorithmic improvements, memory-usage, the specific distribution of tag sets in a realistic FIB, and many other issues that are important in practice. We therefore turn to an experimental evaluation. In particular, we evaluate the forwarding engine we developed for TagNet in order to examine the following research questions: (1) How effective are the memory compression techniques that we propose? (2) What are the differences in performance between the SOL and SORL memory layouts? (3) How scalable is our implementation in terms of number of CPU cores and FIB size? (4) What is the effective throughput of our matcher under different traffic workloads?

Our subject is a matcher written in C++ that implements the three algorithms described in Section 3, namely the FAS, which we run with fan-out limit $k = \infty$, and the FLS algorithm, both operating on a vector FIB, plus the locator-based forwarding algorithm of Thorup and Zwick. We run all our experiments on a general-purpose machine equipped with two Intel Xeon E5-2670 v3 processors, each with 12 cores and a clock frequency of 2.30GHz. The machine has 64GB of RAM.

With this subject and test-bed, we measure memory and throughput of the forwarding engine. The throughput corresponds to the total processing time of the forwarding engine, which includes the parsing of the packet header to figure out the type of address (descriptor or locator) and the fan-out limit (for descriptors), the dispatching of the packet to the appropriate algorithm (FAS, FLS, or TZ forwarding), and the execution of that algorithm. In all our experiments, we measure matching times and throughput as an aggregate measure (average) over all the packets in a workload, which is typically over one million packets. These average measurements are extremely consistent over repeated trials, therefore we never report any variance in the charts and in the tables. In fact, time measurements (averages over all packets) are so consistent that the full variability range for repeated trials (maximum minus minimum) is never more than 3% of the mean.

We begin our analysis by characterizing the data sets and the workloads we use for the evaluation, and then we continue with a series of experiments to answer our research questions.

4.1 FIB and Traffic Workloads

We test descriptor-based forwarding under three different FIBs. The first FIB workload, labeled 63M, contains more than 63 million unique descriptors. We generated this set of descriptors for a previous analysis of the scalability of the

[1]Internet AS-level topology archive (http://irl.cs.ucla.edu/topology/), data retrieved 29/06/2012.

[2]See Section 4.3 of Rivest [8]. The general complexity is $O(n^{\log_2(2-s/k)})$, where s/k is the ratio of fixed bits over the total number of bits in the partial-matching query. Since $s/k < 1$, $\log_2(2 - s/k)$ is approximately $1 - s/k$, which is the ratio of "don't care" bits, which in our case is h/m, since 1-bits in the input Bloom filter play the role of "don't care."

routing protocol we proposed for TagNet [6]. In particular, we derived the 63M FIB from the routing state generated by 500 million users for a variety of classes of applications. The second workload we use, labeled 10M, is a sample of 10 million descriptors taken from the 63M workload. The third workload, labeled 10M-CCN, is composed of almost 10 million descriptors. This workload corresponds to the FIBs used by Wang et al. to test their GPU-based matcher for CCN hierarchical names [12], and was also used to test other implementations [7, 11]. Notice that the workload by Wang et al. consists of hierarchical names, not tag-set descriptors. We therefore compile 10M-CCN by transforming hierarchical names into descriptors as shown in Section 2.1. We use this workload to show that our algorithms perform well also when they emulate the semantics of hierarchical names.

In addition to the FIB, the workload must provide a way to feed traffic into the forwarder, specifically we need to define a way to create the descriptors in the packets. So, in order to obtain valid results, we first analyze various options for the traffic mix. The analysis shows that the matching rate has a fundamentally negative impact on performance. Figure 7 shows the matching time of the two descriptor matching algorithms as a function of the matching rate, for the 10M FIB and for a single thread.

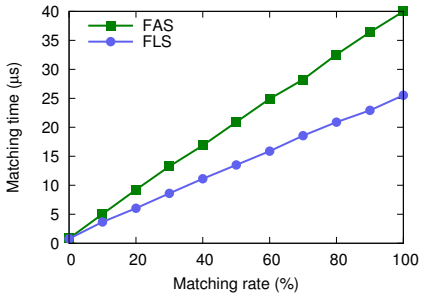

Figure 7: Effect of the matching rate on performance; matching rate is the percentage of messages that match at least one descriptor in the FIB

Notice that even a single thread can handle on average 1.2Mpps (million packets per second) with the FAS algorithm, and 1.3Mpps with FLS, when none of the packets matches a single descriptor (0% matching rate). In fact, non-matching packets can be discarded very quickly after a few checks. However, the matching time grows quickly with the matching rate, especially for the FAS algorithm. Based on these results, to be conservative, in the rest of the evaluation we use workloads with 100% of matching packets. To guarantee the high rate of matching, we generate packet descriptors by drawing descriptors from the FIB itself and then by adding extra tags to those descriptors.

The number of additional tags that we put in the packet descriptors also plays a fundamental role in the performance of our matcher, since it further increases the matching rate. And even without more matches, having more tags and therefore descriptors with higher Hamming weights means more paths to visit in the trie. Figure 8 shows the impact of the additional tags on the average matching time (in microseconds) for the FAS and FLS algorithms. In this experiment we use again the 10M FIB and we run again the

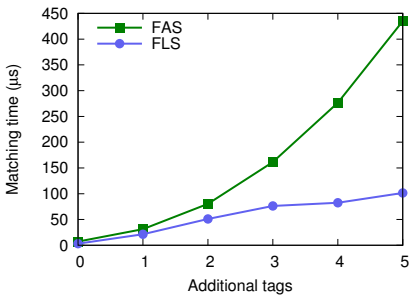

Figure 8: Effect of the descriptor size on the performance of the matcher

matcher with a single thread. It is clear from the figure that the number of tags in the packet descriptors has a very significant impact on the FAS algorithm, with the matching time growing almost exponentially with the number of additional tags, which is also consistent with the complexity analysis discussed earlier. This effect, however, is much less visible for the FLS algorithm. This is because increasing the matching rate also increases the chance of finding a good match at the beginning, which in turn allows the algorithm to skip entire sub-trees. In the remainder of our evaluation we use a workload where we introduce up to two additional tags to the packet descriptors drawn from the FIB.

4.2 Trie Compression

We now present the results of a number of experiments intended to test the effectiveness of our trie compression techniques, namely bit permutation (Section 3.2.1) and chain removal (Section 3.2.2), both in terms of memory usage and in terms of the performance of the matching algorithms.

We start by analyzing the memory usage. We always use the vector representation for the trie, but we apply different combinations of compression. Figure 9a shows the results for all the FIB workloads. The histogram shows the memory used by the trie with no compression ("no compression"), when we apply the bit permutation ("bit perm"), and when we also remove chains ("bit perm+chain"). The bit permutation is quite effective on all the workloads, with a 48% reduction in the FIB size for the 10M FIB, 42% for 10M-CCN, and 46% for 63M. Chain removal is also effective in all cases. In the end, the size of the FIB for 10M and 10M-CCN is 171MB, while it is 1.06GB for 63M.

We then look at the effect of trie compression on matching time. Figure 9b shows the matching time required by the FAS and FLS algorithms using the different trie compression techniques. We run our experiments using a single thread using the 10M workload. The results are in microseconds. We label the different compression techniques as in Figure 9a.

The results show that trie compression is not only useful to reduce the memory footprint of the FIB, but it also reduces the matching times for both FAS and FLS. The bit permutation is the most effective optimization in terms of matching time. This is due to the fact that it reduces the number of prefixes to check, which reduces the search space. Bit permutation alone yields a 29.2% performance improvement for the FAS algorithm, and 16.8% for FLS. In addition to that, chain removal also gives a good improvement, with an additional 8% for FAS and 6.2% for FLS.

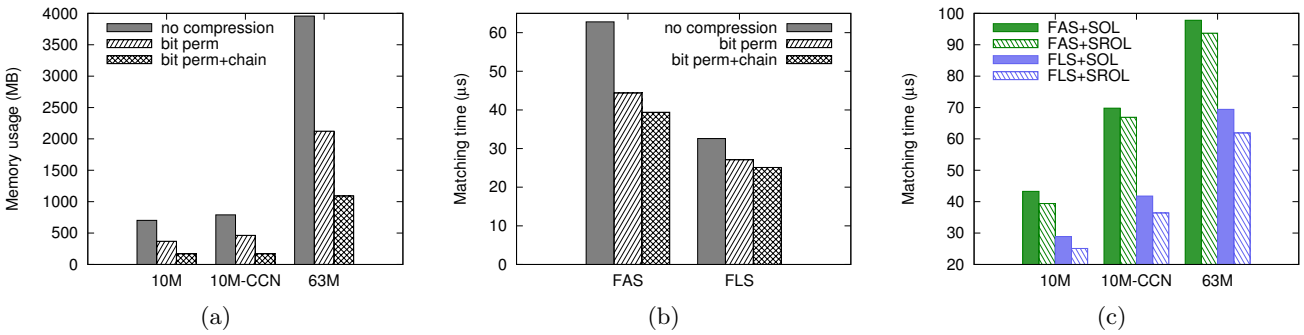

Figure 9: Memory usage for different FIBs with different types of trie compression (a); matching times with the 10M workload with different types of trie compression (b), and with different node layouts (c)

4.3 Memory Layout

We now examine the performance of the matcher in terms of average matching time using the two memory layouts discussed in Section 3.2.3. In Figure 9c we report the average matching time for the SOL and SROL memory layouts. The histogram shows the matching time required by both the FAS and FLS algorithms for all the FIB workloads. The results are for a single thread.

What is immediately clear is that the SROL layout improves performance in all cases. The relative improvements vary. In the worst case, when we use the FAS algorithm with the 10M-CCN and 63M FIBs, we gain 4.1%. However, with the FLS algorithm we always gain more than 10%, with a peak of 13.1% in the case of 10M.

These results are also particularly important to measure the latency of our matcher, since the values reported in Figure 9c represent the average latency introduced by the matcher. In particular, the latency of our implementation is the one of the SROL layout, shown in the histogram with the diagonal pattern. According to Wang et al. the latency should be lower than $100\mu s$ [12], and our implementation satisfies this requirement with all the workloads we tested.

4.4 Implementation Scalability

We now test the scalability of our implementation with respect to the number of threads and with respect to the size of the FIB.

4.4.1 Number of Threads

With modern hardware architectures, it is particularly important that algorithms be capable of exploiting parallelism. In the following experiments we test the scalability of our implementation with respect to the number of CPU threads used within the matcher. In Figure 10a we show the throughput in thousands of packets per second (Kpps) of the FAS algorithm when we vary the number of threads. The plot shows that the implementation scales almost perfectly with higher numbers of threads for all three FIB workloads. Every time we double the number of threads, we gain on average 80.8% in throughput, with a peak of 98.3%, which is very close to linear scaling. However, when we go from 16 to 32 threads, using the 63M FIB, we only obtain a 57.1% improvement. We believe this is due to the fact that our test machine has only 24 real cores, so the 32 threads run using Intel's Hyper-Threading technology. Using the FAS algorithm, our router can forward on average 499Kpps with 10M, 307Kpps with 10M-CCN, and 183Kpps with 63M.

In Figure 10b we show the multi-threading scalability of the FLS algorithm. Also in this case the implementation scales well, since on average we obtain an incremental (doubling) gain of 83.1%, with a peak of 98.4%. And again the least incremental gain (59.3%) is when we use 32 threads with the 63M FIB. The throughput in the case of FLS is almost twice the throughput of the FAS algorithm. Our matcher processes 914Kpps with 10M, 602Kpps in case of 10M-CCN and 272Kpps using 63M. The gain in the throughput is due to the fact that using this algorithm we can skip more checks and so we need less memory accesses.

Another interesting result visible both in Figure 10a and Figure 10b is that, although the 10M and 10M-CCN workloads require the same amount of memory (see Figure 9a), their performance differs significantly under the two workloads. This is due to the fact that 10M-CCN has more paths in the top part of the trie as compared to 10M. For this reason, the algorithm needs to explore more nodes, which in turn requires more memory accesses.

4.4.2 FIB Size

A crucial measure of scalability for our matcher is the ability to sustain a good throughput with large FIBs. We already have some evidence of good scalability from the results of figures 10a and 10b, since the throughput achieved with the 63M FIB is only 2.59 times lower (on average) than the throughput with the 10M FIB, even though the 63M FIB is more than 6 times larger than the 10M FIB.

To further test the scalability in the size of the FIB, we conduct an experiment in which we vary the size of the FIB, starting with 10 million entries and increasing the size up to 60 million entries in steps of 10 millions. We report the results of this experiment in Figure 10c, where we show the throughput of the matcher with the FAS and FLS algorithms. The chart clearly shows that the throughput decreases, but does so with a flattening slope, which confirms that our implementation scales well with the size of the FIB.

4.5 Performance with Mixed Traffic

So far we tested only the performance of the content-based matcher. However our matcher can also forward packets by locator and therefore benefit from their efficiency. To highlight these benefits, but also to test the performance of the matcher under more realistic workloads than the ones used

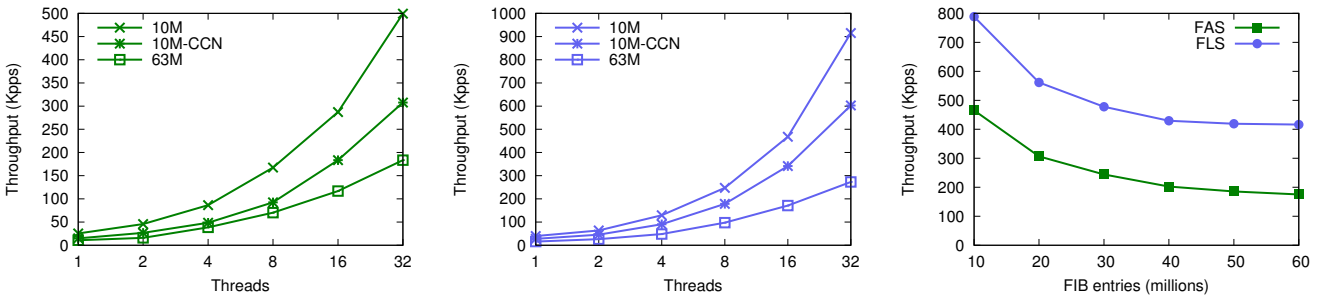

Figure 10: Throughput of FAS (a) and FLS (b) with multiple threads, and with different FIB sizes (c)

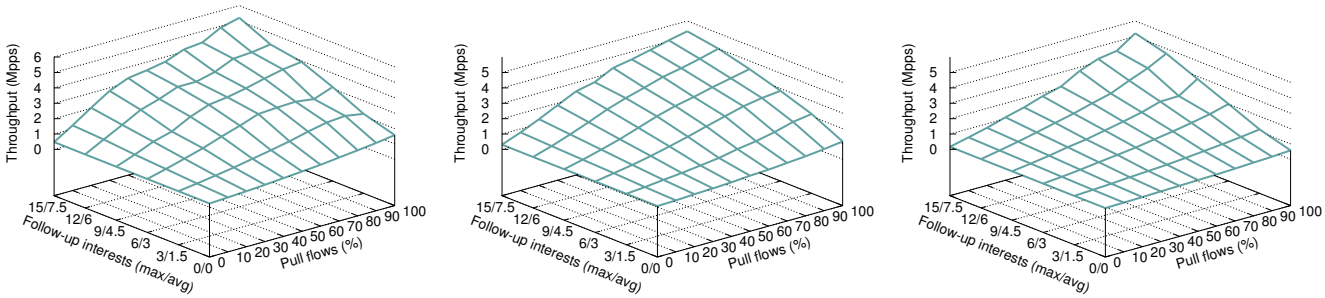

Figure 11: Throughput with different traffic mixes

so far, we create different traffic mixes. In particular, we generate a traffic mix where we vary the percentage of push and pull flows as described in Section 2.1. A push flow always consists of a single packet that we forward using the FAS algorithm ($k = \infty$). A pull flow starts with an anycast "interest" packet addressed by descriptor, and therefore forwarded with the FLS algorithm, and continues with a sequence of follow-up interest packets addressed directly to the producer by locator, and for each interest there is also a data packet forwarded by locator in the opposite direction. In each in pull flow, we choose the number of follow-up interests uniformly at random in a range between 0 and a maximum value that we set as the independent variable of our experiments.

In these experiments we measure the throughput in Mpps for the three different FIBs, varying the percentage of pull traffic and the number of follow-up interests. We display the results in Figure 11. In the charts we indicate the number of follow-up interests on the x-axis, the percentage of pull flows on the y-axis, and the measured throughput on z-axis. Notice that on the x-axis we indicate both the maximum and the average number of follow-up interests that we can have in each pull flow. The results demonstrate that we can easily forward millions of packets per second, even using short pull flows averaging only 7.5 follow-up requests. The throughput also depends on the push traffic, which is quite costly for forwarding. For example, if we consider a traffic mix where we have 80% of pull flows and 20% of push flows, we can forward 2.89Mpps with 63M, 4.27Mpps with 10M, and 3.79Mpps with 10M-CCN.

In this experiment we also estimate the throughput in bits per second. To do that we compute the average packet size for each traffic mix. We hypothesize that push packets and data packets are quite large, since they would typically carry data. In particular, for those packets we consider 1280 bytes, which is the minimum link MTU in IPv6.[3] For the first interest we consider the average size of an HTTP GET header, which is around 800 bytes according to the Google SPDY report.[4] Finally we consider follow-up interests to be small packets of 100 bytes. With these parameters, Table 1 reports the average packet size (in bytes, in the second column) for different percentages of pull traffic. In this table we consider only the case where we have a maximum number of follow-up requests equal to 15. The table reports the throughput in Mpps and in Gbps. Considering again the case where we have 80% of pull flows and 20% of push flows, our forwarder can achieve a throughput of 17.1Gbps using 63M, 22.4Gbps using 10M-CCN and 25.3Gbps with 10M.

Table 1: Throughput (Mpps/Gbps) for different traffic mixes using different FIBs

		Throughput (Mpps/Gbps)		
pull (%)	pkt (B)	10M	10M-CCN	63M
0	1280	0.47/4.8	0.31/3.1	0.17/1.8
20	835	2.04/13.6	1.30/8.7	0.74/5.0
40	775	3.30/20.5	2.44/15.2	1.39/8.7
60	751	3.70/22.2	3.29/19.8	2.08/12.5
80	739	4.27/25.3	3.79/22.4	2.89/17.1
100	731	5.20/30.5	4.33/25.3	4.2/24.6

[3]RFC2469: Internet Protocol, Version 6 (IPv6) Specification. https://tools.ietf.org/html/rfc2460

[4]SPDY: An experimental protocol for a faster web. https://www.chromium.org/spdy/spdy-whitepaper

5. CONCLUSION

We presented and evaluated a data plane for TagNet, an ICN architecture that features a dual addressing scheme and two corresponding delivery services, one based on expressive, application-defined descriptors, and one based on extremely efficient, network-defined locators. We see this dual addressing, and the algorithms we developed to support it, as an effective way or perhaps simply a first but essential step in designing a true *information centric* network.

In prior work we proposed and evaluated a routing scheme for TagNet. Here we developed the essential algorithms to realize the TagNet data plane. In particular, we developed a forwarding engine that implements specialized subset matching algorithms for descriptor forwarding and an extremely fast locator forwarding algorithm based on a compact routing scheme for trees. The forwarding engine runs on general purpose CPU, and yet it is capable of forwarding over 20Gbps of mixed traffic flows with large forwarding tables corresponding to hundreds of millions of users.

One of our most immediate plans for future research includes the development of a forwarding engine built on massively parallel hardware to increase the throughput of even the descriptor-based forwarding alone. Indeed we have a GPU-based prototype that in a preliminary evaluation can process almost one million descriptors per second with the 63M FIB and the same traffic workload we used in the evaluation presented in this paper.

6. ACKNOWLEDGMENTS

We thank Alessandro Margara and Gianpaolo Cugola for their comments an insights in many useful discussions on the subset matching problem. This work was supported in part by the Swiss National Science Foundation under grant number 200021-132565 and under grant number 200021-157164.

7. REFERENCES

[1] M. Charikar, P. Indyk, and R. Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In *Proceedings of the 29th International Colloquium on Automata, Languages, and Programming (ICALP 2002)*, pages 451–462, July 2002.

[2] H. Dai, J. Lu, Y. Wang, and B. Liu. BFAST: Unified and scalable index for NDN forwarding architecture. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 2290–2298, Apr.–May 2015.

[3] D. Han, A. Anand, F. Dogar, B. Li, H. Lim, M. Machado, A. Mukundan, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste. Xia: Efficient support for evolvable internetworking. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, pages 309–322, Apr. 2012.

[4] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison predicates. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 386–395, Aug. 1997.

[5] Y. Luo, G. H. L. Fletcher, J. Hidders, and P. De Bra. Efficient and scalable trie-based algorithms for computing set containment relations. In *31st IEEE International Conference on Data Engineering (ICDE'15)*, pages 303–314, Apr. 2015.

[6] M. Papalini, A. Carzaniga, K. Khazaei, and A. L. Wolf. Scalable routing for tag-based information-centric networking. In *Proceedings of the 1st International Conference on Information-centric Networking (ICN'14)*, pages 17–26, Sept. 2014.

[7] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue. 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 (ANCS'14)*, pages 137–148, Oct. 2014.

[8] R. L. Rivest. Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50, Mar. 1976.

[9] W. So, A. Narayanan, and D. Oran. Named data networking on a router: Fast and dos-resistant forwarding with hash tables. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'13)*, pages 215–226, Oct. 2013.

[10] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures (SPAA'01)*, pages 1–10, July 2001.

[11] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast name lookup for named data networking. In *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS)*, pages 198–207, May 2014.

[12] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang. Wire speed name lookup: A gpu-based approach. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, pages 199–212, Apr. 2013.

[13] H. Yuan and P. Crowley. Reliably scalable name prefix lookup. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'15)*, pages 111–121, May 2015.