Caesar: A Content Router for High-Speed Forwarding on Content Names

Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, Roger Boislaigue

Caesar: A Content Router for High-Speed Forwarding on Content Names

Diego Perino, Matteo Varvello
Bell Labs, Alcatel-Lucent
first.last@alcatel-lucent.com

Leonardo Linguaglossa
INRIA
first.last@inria.fr

Rafael Laufer, Roger Boislaigue
Bell Labs, Alcatel-Lucent
first.last@alcatel-lucent.com

ABSTRACT

Internet users are interested in content regardless of its location; however, the current client/server architecture still requires requests to be directed to a specific server. Information-centric networking (ICN) is a recent vein that relaxes this requirement through the use of name-based forwarding, where forwarding decisions are based on content names instead of IP addresses. Despite previous name-based forwarding strategies have been proposed, almost none have actually built a content router. To fill this gap, in this paper we design and prototype a content router called *Caesar* for high-speed forwarding on content names. Caesar introduces several innovative features, including (*i*) a longest-prefix matching algorithm based on a novel data structure called *prefix Bloom filter*; (*ii*) an incremental design which allows for easy integration with existing protocols and network equipment; (*iii*) a forwarding scheme where multiple line cards collaborate in a distributed fashion; and (*iv*) support for offloading packet processing to graphics processing units (GPUs). We build Caesar as an enterprise router, and show that every line card sustains up to 10 Gbps using a forwarding table with more than 10 million content prefixes. Distributed forwarding allows the forwarding table to grow even further, and to scale linearly with the number of line cards at the cost of only a few microseconds in the packet processing latency. GPU offloading, in turn, trades off a few milliseconds of latency for a large speedup in the forwarding rate.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Designs**]: Network communications, Store and forward networks; C.2.6 [**Internetworking**]: Routers

General Terms

Design; Implementation; Experiments.

Keywords

ICN; forwarding; router; architecture.

ANCS'14, October 20–21, 2014, Los Angeles, CA, USA.

ACM 978-1-4503-2839-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2658260.2658267.

1. INTRODUCTION

Internet usage has significantly evolved over the years, and today is mostly centered around location-independent services. However, since the Internet architecture is host-centric, content requests still have to be directed towards an individual server using IP addresses. The translation from content name to IP address is realized through different technologies, e.g., DNS and HTTP redirection, which are implemented by several systems, such as content delivery networks (CDN) [1] and cloud services [2].

Information-centric networking (ICN) offers a radical alternative by advocating name-based forwarding directly at the network layer, *i.e.,* forwarding decisions are based on the content name carried by each packet [3]. Names provide routers with information about the forwarded content, which enables functionalities, such as caching or multicasting, as network-layer primitives. In particular, the use of hierarchical names [4] also allows efficient route aggregation and makes mechanisms to translate content names into IP addresses unnecessary.

At the core of the ICN architecture is a network device called content router, responsible for name-based forwarding. Building a content router is challenging because of two major issues [5, 6]. First, due to the ever-increasing availability of content, the size of the forwarding tables are expected to be from one to two orders of magnitude larger than current tables. Second, content names may be long, having a large number of components as well as many characters per component, which makes several previous hardware optimizations proposed for fixed-length IP prefixes ineffective [7].

In this paper, we address these ICN challenges and introduce *Caesar*, a content router compatible with existing protocols and network equipment. Caesar's forwarding engine features three key optimizations to accelerate name lookups. First, its name-based longest-prefix matching (LPM) algorithm relies on a novel data structure called *prefix Bloom filter* (PBF). The PBF is introduced to achieve high caching efficiency by exploiting the hierarchical nature of content prefixes. Second, a fast hashing scheme is proposed to reduce the PBF processing overhead by a multiplicative factor. Finally, Caesar takes advantage of a cache-aware hash table designed with an efficient collision resolution scheme. The goal is to minimize the number of memory accesses required to find the next-hop information for each packet.

Based on the proposed design, we implement the data plane of Caesar using a μTCA chassis and multiple line cards equipped with a network processor. In its basic design, Caesar maintains a full copy of the forwarding information base (FIB) at each line card. For each received packet, our name-based LPM algorithm runs independently at the input line card, and an Ethernet switch then moves the packet to the output line card following the forwarding decision made. To support large FIBs, we extend Caesar with a dis-

tributed forwarding scheme where each line card stores only *part* of the FIB and collaborate with each other to perform LPM. A second extension is also implemented to further increase Caesar's forwarding speed by offloading name-based LPM to a graphics processing unit (GPU), if required.

We evaluate Caesar using our full prototype and a commercial traffic generator that uses synthetic and real traces for the content prefixes and requests. Our main finding is that every line card of Caesar is able to sustain up to 10 Gbps with 188-byte packets and a large FIB with 10 million prefixes. Distributed forwarding over line cards sharing their FIB is shown to allow the the forwarding table to increase linearly with the number of cards at the cost of 15% rate reduction and a few microseconds of additional delay. The GPU extension is also shown to outperform previously proposed designs for the same hardware.

The remainder of this paper is organized as follows. In Section 2, we provide some background and an overview of the work closely related to Caesar. Section 3 presents the design of our name-based LPM algorithm. Section 4 then introduces the implementation of Caesar, while its extensions are presented in Section 5. We evaluate Caesar's performance in Section 6 and, in Section 7, we discuss the design and implementation of additional content router features. Finally, Section 8 concludes the paper.

2. RELATED WORK

This section summarizes the work related to Caesar. Section 2.1 provides a brief background on fundamental NDN concepts, such as name-based longest prefix matching (LPM). Section 2.2 then overviews the state of the art in name-based LPM, and Section 2.3 presents the related work on content router design.

2.1 Background

Caesar uses the hierarchical naming scheme proposed by NDN to address content [4, 8, 9, 10]. In this scheme, each content has a unique identifier composed of a sequence of strings, each separated by a delimiting character (e.g., `/ancs2014/papers/paperA`). We refer to this identifier as the *content name*, and to each string in the sequence as a *component*. For delivery, content usually has to be split into several different packets, which are identified by appending an extra individual component to the original content name (e.g., `/ancs2014/papers/paperA/packet1`). For scalability, content routers only maintain forwarding information for *content prefixes* that aggregate several content names into a single entry (e.g., `/ancs2014/papers/`$\star$).

A content router uses name-based longest prefix matching (LPM) to determine the interface where a packet should be forwarded. Name-based LPM consists in selecting from a local forwarding information base (FIB) the content prefix sharing the longest prefix with a content name. Although the concept is similar to LPM for IP, name-based LPM faces serious scalability challenges [5, 6]. An ICN FIB is expected to be at least one order of magnitude larger than the average FIB of current IP routers. In addition, several hardware optimizations that take advantage of the fixed length of IP addresses are not possible in ICN due to the variable length of content names and prefixes.

2.2 Name-Based LPM

Motivated by these challenges, a few techniques have recently been proposed for name-based LPM. Wang *et al.* [11] propose to use name component encoding (NCE), a scheme that encodes the components of a content name as symbols and organize them as a trie. Due to its goal of compacting the FIB, NCE requires several extra data structures that add significant complexity to the lookup process, and result in several memory accesses to find the longest prefix match. NameFilter [12] is an alternative name-based LPM algorithm employing one Bloom filter per prefix length, similarly to the solutions proposed in [13, 14] for IP addresses. For lookup, a d-component content name then requires d lookups in the different Bloom filters. This approach has two intrinsic limitations. First, it cannot handle false positives generated by the Bloom filters, and thus packets can eventually be forwarded to the wrong interface. Second, it cannot support a few important functionalities, such as multipath routing and dynamic forwarding.

In a different approach, So *et al.* [6, 8] implement LPM using successive lookups in a hash table. Instead of using the longest-first strategy (*i.e.,* lookups start from the longest prefix), the search starts from the prefix length where most FIB prefixes are centered, and restarts at a larger or shorter length, if needed. The approach bounds the worst case number of lookups, but cannot guarantee constant performance bounds.

Different from previous work, we reduce the problem of name-based LPM to two stages (cf. Section 3). The first stage finds the *length* of the longest prefix that matches a content name. This stage is accomplished by a Bloom filter variant engineered for content prefixes, which guarantees a constant number of memory accesses. The second stage consists of a hash table lookup to find the output interface where a packet should be forwarded to. This last stage only requires a single lookup with high probability, detects false positives, and supports enhanced forwarding functionalities.

2.3 Content Router Design

To date, the work in [6, 8] is the only previous attempt to build a content router. In this work, the content router is implemented on a Xeon-based Integrated Service Module. Packet I/O is handled by regular line cards, while name-based LPM is performed on a separate service module connected to the line cards via a switch fabric. Real experiments show that the module sustains a maximum forwarding rate of 4.5 Mpps (million packets per second). Simulations without packet I/O show that the the proposed name-based LPM algorithm handles up to 6.3 Mpps.

Different from [6, 8], Caesar supports name-based LPM directly on I/O line cards in order to reduce latency, increase the overall router throughput, and enable ICN functionalities without requiring extra service modules. Real experiments show that, using a single line card based on a cheaper technology than [6, 8], Caesar achieves a comparable throughput (Section 6). Finally, Caesar also allows line cards to share the content of their FIB in order to support the massive FIB expected in ICN, and supports GPU offloading to speed up the forwarding rate (Section 5).

3. NAME-BASED LPM

In this section, we introduce our two-stage name-based longest prefix matching (LPM) algorithm used in the forwarding engine of Caesar. Section 3.1 and 3.2 describe the prefix Bloom filter (PBF) and the concept of block expansion, respectively. Both are used in the first stage to find the length of the longest prefix match. Then, Section 3.3 explains the fast hashing scheme proposed to reduce the hashing overhead of the PBF. Finally, Section 3.4 describes the hash table used in the second stage as well as the optimizations introduced to speed up the lookups.

3.1 Prefix Bloom Filter

For the first stage of our name-based LPM algorithm, we introduce a novel data structure called *prefix Bloom filter* (PBF) and use it as an oracle to identify the length of the longest prefix match. The PBF takes advantage of the semantics in content prefixes to

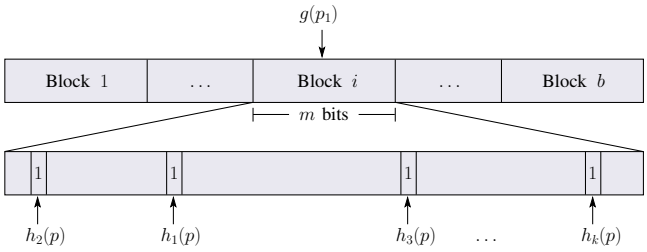


Figure 1: Insertion of a prefix p into a PBF with b blocks, m bits per block, and k hash functions. The function $g(p_1)$ selects a block using the subprefix p_1 with the first component, and bits $h_1(p), h_2(p), \ldots, h_k(p)$ are set to 1.

find the longest prefix match using a single memory access, with high probability.

The PBF is a space-efficient data structure composed of several blocks, where each block is a small Bloom filter with the size of one (or few) cache line(s). Each content prefix in the FIB is inserted into a block chosen from the hash value of its first component. During the lookup of a content name, its first component identifies the unique block, or cache line(s), that must be loaded from memory to cache before conducting the membership query.

Figure 1 shows the insertion of a content prefix p into a PBF composed of b blocks, m bits per block, and k hash functions. Let $p = /c_1/c_2/\ldots/c_u$ be the u-component prefix to be inserted into the PBF, and let $p_i = /c_1/c_2/\ldots/c_i$ be the subprefix with the first i components of p, such that $p_1 = /c_1$, $p_2 = /c_1/c_2$, and so on. A uniform hash function $g(\cdot)$ with output in the range $\{0, 1, \ldots, b-1\}$ is used to determine the block where p should be inserted. The hash value $g(p_1)$ is computed from the subprefix p_1 defined by the first component. This guarantees that all prefixes starting with the same component are stored in the same block, which enables fast lookups. Once the block is selected, the hash values $h_1(p), h_2(p), \ldots, h_k(p)$ are computed using the complete prefix p, resulting in k indexes within the range $\{0, 1, \ldots, m-1\}$. Finally, the bits at the positions $h_1(p), h_2(p), \ldots, h_k(p)$ in the selected block are set to 1.

To find the length of the longest prefix in the PBF that matches a content name $x = /c_1/c_2/\ldots/c_d$, the first step is to identify the index of the block where x or its subprefixes may be stored. Such block is selected using the hash of the first component of the content name, $g(x_1)$. Once this block is loaded, a match is first tried using the full name x, *i.e.,* maximum length. The bits of the positions $h_1(x), h_2(x), \ldots, h_k(x)$ are then checked and, if all bits are set to 1, a match is found. Otherwise, or if a false positive is detected (cf. Section 3.4), the prefix x_{d-1} is checked using the same procedure and, if there is no match, x_{d-2} is then checked, and so on until a match is found or until all subprefixes of x have been tested. At each membership query, the bits of the positions $h_1(x_i), h_2(x_i), \ldots, h_k(x_i)$ are checked, for $1 \leq i \leq d$, accounting for a maximum of $k \times d$ bit checks per name lookup in the worst case. Bits checks require only a single memory access as the bits to be checked reside in the same block.

The false positive rate of the PBF is computed as follows. If n_i is the number of prefixes inserted into the i-th block, then the false positive rate of this block is $f_i = (1 - e^{-kn_i/m})^k$. We consider two possible cases of false positives. First, assume the worst-case scenario where the name to be looked up and all of its subprefixes are *not* in the FIB. In this case, assuming a content name with d components, the number F_i of false positives in the i-th block follows the binomial distribution $F_i \sim B(d, f_i)$. The average number of false positives in this block is then $d \times f_i$. Since the function $g(\cdot)$ is uniform, each block is chosen with probability $1/b$, where b is the number of blocks, and thus the average number of false positives in the PBF for a content name with d components and no matches is $d \times f$, where $f = (1/b)\sum_{i=1}^{b} f_i$ is the average false positive rate.

Consider now the case where either the content name or at least one of its subprefixes *are* in the table, and let l be the lenght of its longest prefix match. In this case, a false positive can only occur for a subprefix whose length is larger than l, i.e., the l-component subprefix is a real positive and the search stops. The number F_i of false positives in the i-th block then follows the binomial distribution $F_i \sim B(d-l, f_i)$, and the average number of false positives in this block is $(d-l) \times f_i$. In general, for a d-component name whose longest prefix match has length l, the average number of false positives in the PBF is $(d-l) \times f$.

3.2 Block Expansion

For fast lookups, the PBF is designed such that prefixes sharing their first component are stored in the same block. It follows that if many prefixes in the FIB share the same first component, then the corresponding block may yield a high false positive rate. To address this, we propose a technique called *block expansion* that redirects some content prefixes to other blocks, allowing the false positive rate to be reduced in exchange for loading a few additional blocks from memory.

Block expansion is used when the number n_i of prefixes in the i-th block exceeds the threshold $t_i = -(m/k)\log(1 - \sqrt[k]{f_i})$ selected to guarantee a maximum false positive rate f_i. For now, assume that prefixes are inserted in order from shorter to longer lengths[1]. Let n_{ij} be the number of j-component prefixes stored in the i-th block. If at a given length l the number $\sum_{j \leq l} n_{ij}$ exceeds the threshold t_i, then a block expansion occurs. In this case, each prefix p with length l or higher is redirected to another block chosen from the hash value $g(p_l)$ of its first l components. To keep track of the expansions, each block keeps a bitmap with w bits. The l-th bit of the bitmap is set to 1 to notify that an expansion at length l occurred in the block. If the new block indicated by $g(p_l)$ already has an expansion at a length e, with $e > l$, then any prefix p with length e or higher is redirected again to another block indicated by $g(p_e)$, and so on.

Figure 2 shows the insertion of a prefix $p = /c_1/c_2/\ldots/c_u$ in a PBF using block expansion. First, block $i = g(p_1)$ is identified as the target for p. Assuming that the threshold t_i is reached at prefix length l, block i is expanded and the l-th bit of its bitmap is set. Since $l \leq u$, a second block $j = g(p_l)$ is then be computed from the first l components of p and, assuming block j is not expanded, positions $h_1(p), h_2(p), \ldots, h_k(p)$ of this block are set to 1.

The lookup process works as follows. Let x be the prefix to be looked up, and $i = g(x_1)$ be the block where x or its LPM should be. First, the expansion bitmap of block i is checked. If the first bit set in the bitmap is at position l and x has l or more components, then block $j = g(x_l)$ is also loaded from memory. Assuming that no bits are set in the bitmap of j, prefixes x_l and higher are checked in block j. In case there are no matches, then prefixes x_{l-1} and lower are checked in block i.

The false positive rate of the PBF with block expansion is similar to the case without expansion, except for two key differences. First, the filter size is now $m - w$ bits, since the first w bits of the block are used for the expansion bitmap. The range of the hash

[1]The dynamic case, where the prefixes in the FIB change over time, is addressed by the control plane, and explained in Section 4.3.

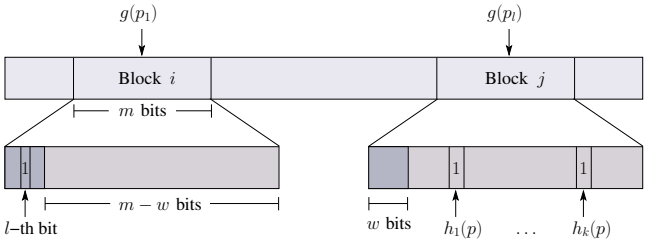


Figure 2: Insertion of a prefix $p = /p_1/p_2/\ldots/p_d$ into a PBF using block expansion. If block $i = g(p_1)$ reached its insertion threshold or if the l-th bit is set in its bitmap and $l \leq d$, then p is inserted into block $j = g(p_l)$.

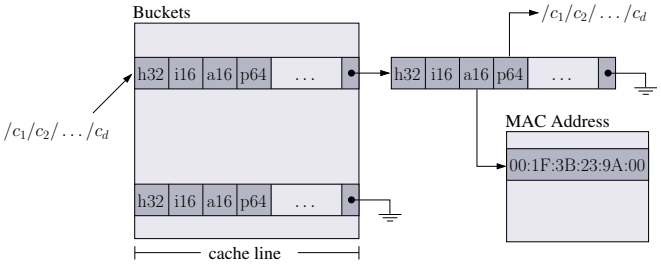


Figure 3: The structure of the hash table used to store the FIB. Each bucket has a fixed size of one cache line, with overflows managed by chaining. Each entry consists of a tuple $\langle h, i, a, p\rangle$ that stores the next hop information.

functions h_i is thus $\{0, 1, \ldots, m - w - 1\}$. Second, the number n_i of prefixes inserted in each block i is now computed from the original insertions, minus the prefixes redirected to other blocks, plus the prefixes coming from the expansion of other blocks.

3.3 Hashing

Hashing is a fundamental operation in our name-based LPM algorithm. For the lookup of a content name $p = /c_1/c_2/\ldots/c_d$ with d components and k hash functions in the PBF, a total of $k \times d$ hash values must be generated for LPM in the worst case, *i.e.,* the running time is $O(k \times d)$. Longer content names thus have a higher overall impact on the system throughput than shorter names. To reduce this overhead, we propose a linear $O(k + d)$ run-time hashing scheme that only generates $k + d - 1$ seed hash values, while the other $(k - 1)(d - 1)$ values are computed from XOR operations.

The hash values are computed as follows. Let H_{ij} be the i-th hash value computed for the prefix $p_j = /c_1/c_2/\ldots/c_j$ containing the first j components of p. Then, the $k \times d$ values are computed on demand as

$$H_{ij} = \begin{cases} h_i(p_j) & \text{if } i = 1 \text{ or } j = 1 \\ H_{i1} \oplus H_{1j} & \text{otherwise} \end{cases}$$

where $h_i(p_j)$ is the value computed from the i-th hash function over the j-component prefix p_j, and $\oplus$ is the XOR operator. The use of XOR operations significantly speeds up the computation time without impacting hashing properties [14].

3.4 Hash Table Design

After the PBF identifies the longest prefix length, the second stage of our name-based LPM algorithm consists of a hash table lookup to either fetch the next hop information or to rule out false positives.

Figure 3 shows the structure of the hash table used in our system, which consists of several buckets where the prefixes in the FIB are hashed to. Our first design goal is to minimize memory access latency. For this purpose, each bucket is restricted to the fixed size of one cache line such that, for well-dimensioned tables, only a single memory access is required to find an entry. In case of collisions, entries are stored next to each other in a contiguous fashion up to the limit imposed by the cache line size. Bucket overflow is managed by chaining with linked lists, but this is expected to be rare if the number of buckets is large enough.

Our second design goal is to reduce the string matching overhead required to find an entry. As a result, each entry stores the hash value h of its content prefix in order to speed up the matching process. String matching on the content prefix only occurs if there is a match first on this 32-bit hash value. Due to large output range of the hash function, an error is expected only with a small probability of 2^{-32}, assuming uniformity.

Finally, our last goal is to maximize the capacity of each bucket. For this purpose, the content prefix is not stored at each entry due to its large and variable size. Instead, only a 64-bit pointer p to the prefix is stored. To save space, next-hop MAC addresses are also kept in a separate table and a 16-bit index a is stored in each entry. A 16-bit index i is also required per entry to specify the output line card of a given content prefix. Each entry in the hash table then consists of a 16-byte tuple $\langle h, i, a, p\rangle$, where h is the hash of the content prefix, i is the output line card index, a is index of the next-hop MAC address, and p is the pointer to the content prefix.

4. CAESAR

This section explains the design and implementation of Caesar, our high-speed content router prototype. Section 4.1 overviews the hardware setup of Caesar, while Section 4.2 and 4.3 present its data and control plane, respectively.

4.1 Hardware

Caesar's hardware is chosen with three key goals in mind:

Enterprise router: Caesar is a router for an enterprise network, *i.e.,* few 10 Gbps ports. This impacts the choice of router chassis as well as the selection of the type and number of line cards used.

Easy deployment: Caesar is easily deployable in current networks, e.g., via a simple firmware upgrade of existing networking devices. This constraints the hardware choice to programmable components already widely adopted by commercial network equipment. We thus resort to network processors optimized for packet processing.

Backward compatibility: Caesar is designed to be backward compatible with existing networking protocols. In particular, its switch fabric is based on regular Ethernet switching, and thus name-based forwarding is implemented on top of existing networking protocols (e.g., Ethernet and IP) in a transparent fashion, without requiring a clean slate approach.

Figure 4 shows the hardware architecture of Caesar. It consists of a micro telecommunications computing architecture (μTCA) chassis with slots for advanced mezzanine cards (AMCs). Four slots of the chassis are occupied by line cards, each equipped with a network processor unit (NPU), a 4-GB off-chip DRAM, a SFP+ 10GbE interface, and a 10 Gbps interface to the backplane. Each NPU has a 10-core 1.1 GHz 64-bits MIPS processor with 32-KB L1 cache per core, and 2-MB L2 shared cache. Some of the remaining slots of the chassis are occupied by an Ethernet switch with 10GbE ports, one connected to each slot via the backplane, and the route

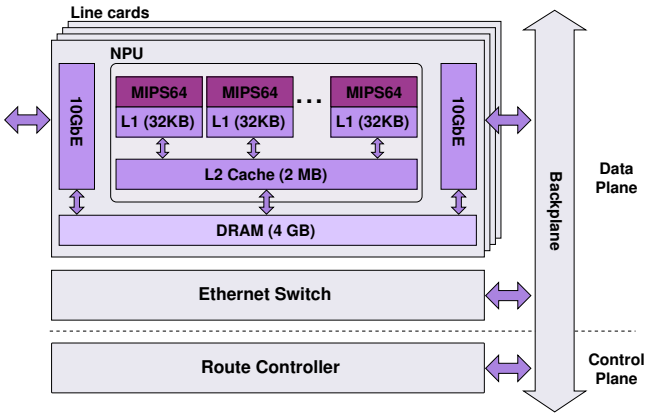


Figure 4: The hardware architecture of Caesar.

controller, composed of an Intel Core Duo 1.5 Ghz processor, 4-GB off-chip DRAM, and a 300-GB hard disk.

4.2 Data Plane

Caesar's data plane is responsible for forwarding packets received by the line cards. Next, we describe the path of packet within Caesar, from the moment it is received until it leaves the system.

Packet input: As a packet is received from the SFP+ 10GbE external interface, it is stored in the off-chip DRAM of the line card. A hardware load balancer then assigns the packet to one of the available cores for processing.

Header parsing: A standard header format for ICN is currently under debate in the ICN research group at the IRTF [15]. In absence of such standard, we use our own header which consists of four fields. First, the 16-bit *length* field specifies the size of the following *content name* field. To expedite parsing, we also include an 8-bit *components* field, which specifies the number of components in the content name, and several *offset* fields, each containing an 8-bit offset for each component in the content name. For backward compatibility, the name-based header is placed after the IP header, which allows network devices to operate with their standard forwarding policy, e.g., L2 or L3 forwarding.

Once dispatched to a core, each packet is checked for the presence of a name-based header by inspecting the protocol field of the IP header. If a name-based header is present, pointers to each field are extracted and stored in the L1 cache. Otherwise, regular packet processing is performed, *i.e.,* LPM on the destination IP address.

Name-based LPM: If such a header is found, our name-based LPM algorithm is used (cf. Section 3). The size of each PBF block is set to one cache line, which is 128 bytes in our architecture (cf. Section 6). To ensure fast hashing calculations, Caesar takes advantage of the optimized instruction sets of the NPU; the $k + d - 1$ seed hash values are computed using the CRC32 optimized instructions, whereas the remaining $(k - 1)(d - 1)$ hash values are computed from XOR operations. In case of a match in the PBF, the content prefix is looked up in the hash table stored in the off-chip DRAM to determine its next hop information, or to rule out false positives. Each table entry has a fixed size of 16 bytes, which, for a bucket of 128 bytes, results in a maximum of 7 entries per bucket in addition to the 64-bit pointer required by the linked list (cf. Section 3.4). We dimension the hash table to contain 10 million buckets, requiring a total of 1.28 GB to store the buckets. An additional 640 MB are required to store the content prefixes, for a total of 1.92 GB of storage.

Switching: The LPM algorithm returns the index of the output line card and the MAC address of the next hop for a packet. The source MAC address of the packet is then set to the address of the backplane interface, and its destination MAC address is set to the address of next hop. Finally, the packet is placed into a per-core output queue in the backplane interface and waits for transmission. Each NPU core has its own queue in the backplane interface to enable lockless queue insertions and avoid contention bottlenecks. Once transmitted over the backplane, the packet is received by the Ethernet switch, and regular L2 switching is performed. The packet is then sent to the output line card over the backplane once again.

Packet output: Once received by the backplane interface of the output line card, the packet is assigned to a NPU core and the source MAC address is overwritten with the address of the SPF+ 10GbE interface. The packet is then sent to the interface for transmission.

4.3 Control Plane

Caesar's control plane is responsible for periodically computing and distributing the FIB to line cards. These operations are performed by the route controller, a central authority that is assumed to participate in a name-based routing protocol [16] to construct its routing information base (RIB). The RIB is structured as a hash table that contains the next hop information for each reachable content prefix.

The FIB is derived from the RIB and is composed of the PBF and the prefix hash table (cf. Section 3). To allow prefix insertion and removal, the route controller maintains a mirror counting PBF (C-PBF). For each bit in the PBF, the C-PBF keeps a counter that is incremented at insertions and decremented at removals. Only when a counter reaches zero the corresponding bit in the PBF is set to 0. The C-PBF enables prefix removal while avoiding to keep counters in the original PBF, which saves precious L2 cache space.

The C-PBF is updated on two different timescales. On a long timescale (*i.e.,* minutes), the C-PBF is recomputed from the RIB with the goal to improve prefix distribution across blocks. On a short timescale (*i.e.,* every insertion/removal) the C-PBF is greedily updated. When inserting a new prefix, additional expansions are performed on blocks that exceed the false-positive threshold. When removing a prefix, block merges are postponed until the next long-timescale update.

The content prefixes stored in the i-th block of the PBF are hierarchically organized into a prefix tree to (1) easily identify the length at which the threshold t_i is exceeded, and (2) efficiently move prefixes during block expansions with a single pointer update operation. The prefix tree of each block is implemented as a left-child right-sibling binary tree for space efficiency.

5. CAESAR EXTENSIONS

In this section, we introduce two Caesar extensions in order to support (1) large FIBs (*i.e.,* tens of gigabytes), and (2) high-speed forwarding (*i.e.,* tens of Mpps). Large FIBs are supported by having each line card store only part of the entire FIB and collaborate with each other in a distributed fashion. High-speed forwarding is supported by offloading large packet batches to a graphics processing unit (GPU). Although efficient, these solutions may introduce additional latency during packet processing and thus are presented here as extensions that can be activated at the operator's discretion.

Large FIBs: In its original design, Caesar stores a full copy of the FIB at each line card, as commonly done by commercial routers. Although this allows each line card to independently process packets at the nominal rate, it also results in FIB replication and waste of storage resources. For IP prefixes, this is usually not a concern, as

a typical FIB contains less than one million entries. In ICN, however, the FIB can easily grow past hundreds of millions of content prefixes [5, 6] and memory space becomes a real concern.

To address this issue, we propose a Caesar extension in Section 5.1 that allows line cards to share their FIB entries. FIBs at different line cards are populated with a unique set of prefixes such that, overall, Caesar is able to store N times more content prefixes, where N is the number of line cards. Since the individual FIB size at each card does not change, line cards are still able to operate at their nominal rate. The key challenge is then how to use a shared FIB to perform LPM on each received packet.

High-speed forwarding: The classic strategy to increase forwarding speeds in routers is a hardware update. However, there is an intrinsic scalability limitation to this approach, in addition to high costs of both hardware and reconfiguration. For instance, upgrading Caesar's line cards from 10 to 40 Gbps requires changing the hardware architecture, with a 10x impact on cost. While this is an option for the deployment of edge/core routers with a large set of networking features, such cost is prohibitive for an enterprise router.

In Section 5.2, we propose an alternative strategy that does not incur such a high cost. Wang *et al.* [9] have recently shown that high-speed LPM on content names is possible by exploiting the parallelism of popular off-the-shelf GPUs. As a second Caesar extension, we propose to use GPUs to accelerate packet processing. Currently, each GPU has an average cost of 10% of the aforementioned architecture upgrade. The challenge is then how to efficiently leverage a GPU to guarantee fast name-based LPM.

For this extension, we assume that a GPU is associated to each line card and that it stores the same FIB entries as the line card. In our platform, a GPU is installed in an external device and connected to a line-card via the switch for power budget reasons. In other platforms (e.g., Advanced Telecommunications Computing Architecture ATCA with enhanced NPU), a GPU can be directly connected to a line card using a regular PCIe bus .

5.1 Distributed Forwarding

To share a large FIB among line cards, we implement a forwarding scheme where LPM is performed in a distributed fashion. The idea is for each packet to be processed at the line card where its longest prefix match resides, *i.e.,* not necessarily the line card that received the packet. A fast mechanism must then be in place for each received packet to be directed to the correct line card for LPM. For this extension, the following modifications to Caesar's control and data planes are required.

Control plane: The route controller now has to compute a different FIB per line card. Each content prefix p in the RIB is assigned to a line card L_i, such that $i = g(p_1) \bmod N$, where $g(p_1)$ is the hash of the subprefix p_1 defined by the first component of p. The rationale here is the same used in the PBF for block selection (cf. Section 3.1); by distributing prefixes to line cards based on their first component, it is possible for an incoming packet to be quickly forwarded to the line card where its longest prefix match resides.

In addition to distributing the FIB, the route controller also maintains a *Line card Table* (LT) containing the MAC address of the backplane interface of each line card. The LT is distributed to each line card along with their FIB, and serves two key purposes.

First, the LT is used by each line card to delegate LPM to another card (see data plane). Second, the LT allows the router controller to quickly recover from failures. With distributed forwarding, the failure of a line card may jeopardize the reachability to the prefixes it manages. We solve this issue by allowing redirection of traffic from a failing line card to a backup line card. Once Caesar detects a failure at a line card L_i, the route controller sends the FIB of L_i to one of the additional pre-installed line cards and updates the LT to reflect the change. The updated table is then distributed to all line cards to complete the failure recovery.

Algorithm 1: Kernel description

Input: Bloom filters B, hash tables H, content names C
Output: Lengths L of the LPM for each name $c \in C$

```
1  prefixLength ← blockIdx div blocksPerLength
2  blockIdxLength ← blockIdx mod blocksPerLength
3  namesPerBlock ← ⌈|C|/blocksPerLength⌉
4  namesOffset ← blockIdxLength × namesPerBlock
5  namesLast ← MIN(namesOffset+namesPerBlock, |C|)
6  tid ← namesOffset+threadIdx
7  while tid < namesLast do
8      c ←READNAME(C, tid)
9      p ←MAKEPREFIX(c, prefixLength)
10     m ← BFLOOKUP(B[prefixLength], p)
11     if m = TRUE then
12         interface ← HTLOOKUP(H[prefixLength], p)
13         if interface ≠ NIL then
14             ATOMICMAX(L[tid], PrefixLength)
15     tid ← tid+ threads
```

Data plane: Upon receiving a packet with content name x, an available NPU core computes the target line card L_i to process the packet, with $i = g(x_1) \bmod N$. If L_i corresponds to the local line card, then the regular flow of operations occurs, *i.e.,* header extraction, name-based LPM, switching, and forwarding (cf. Section 4). Otherwise, the destination MAC address of the packet is overwritten with the address of the backplane interface of L_i fetched from the LT, and the packet is transmitted over the backplane. LPM then occurs at L_i and the packet is sent once again over the backplane to the output line card (if different than L_i) for external transmission.

Distributed forwarding imposes two constraints as tradeoffs for supporting a larger FIB. First, it introduces a short delay caused by packets crossing the backplane twice. Second, extra switching capacity is required. In the worst case, *i.e.,* when a packet is never processed by the receiving line card, the switch must operate twice as fast at a rate $2NR$, where R is the rate of a line card, instead of NR. Nonetheless, as showed in [17], it is possible to combine multiple low-capacity switch fabrics to provide a high-capacity fabric with no performance loss at the cost of small coordination buffers. This is a common approach in commercial routers, e.g., the Alcatel 7950 XRS leverages 16 switching elements to sustain an overall throughput of 32 Tbps [18].

5.2 GPU Offloading

We also propose a Caesar extension to accelerate packet forwarding using a GPU. First, a brief background on the architecture and operation of the NVIDIA GTX 580 [19] used in our implementation is provided. Then, a discussion on our name-based LPM solution using this GPU is presented.

GPU background: The NVIDIA GTX 580 GPU is composed of 16 streaming multiprocessors (SMs), each with 32 stream processors (SPs) running at 1,544 MHz. This GPU has two memory types: a large, but slow, *device memory* and a small, but fast, *shared memory*. The device memory is an off-chip 1.5 GB GDDR5

DRAM, which is accelerated by a L2 cache used by all SMs. The shared memory is an individual on-chip 48 KB SRAM per SM. Each SM also has several registers and a L1 cache to accelerate device memory accesses.

All threads in the GPU execute the same function, called *kernel*. The level of parallelism of a kernel is specified by two parameters, namely, the number of *blocks* and the number of *threads per block*. A block is a set of concurrently executing threads that collaborate using shared memory and barrier synchronization primitives. At run-time, each block is assigned to a SM and divided into *warps*, or sets of 32 threads, that are independently scheduled for execution. Each thread in a warp executes the same instruction in lockstep.

Name-based LPM: We introduce few modifications made to the LPM algorithm to achieve efficient GPU implementation. Due to the serial nature of the NPU, the original algorithm uses a PBF to test for several prefix lengths in the same filter (Section 3). However, to take advantage of the high level of parallelism in GPUs, a LPM approach that uses a Bloom filter and hash table per prefix length is more efficient. Since large FIBs are expected, both the Bloom filters and hash tables are stored in device memory.

For high GPU utilization, multiple warps must be assigned to each SM such that, when a warp stalls on a memory read, other warps are available waiting to be scheduled. The GTX 580 can have up to 8 blocks concurrently allocated and executing per SM, for a total of 128 blocks. Content prefixes are assumed to have 128 components or less, and thus we have one block per prefix length in the worst case. Since such a large number of components is rare, we allow a higher degree of parallelism with multiple blocks working on the same prefix length. In this case, each block operates on a different subset of content names received from a line card.

Algorithm 1 shows our GPU kernel. As input, it receives arrays B, H, and C that contain the Bloom filters, hash tables, and content names to be looked up, respectively. The kernel identifies the length of the longest prefix in the FIB that match each content name $c \in C$ and stores it in the array L, which is then returned to the line card. All these arrays are located in the device memory. We take advantage in the algorithm of a few CUDA variables available to each thread at run-time: blockIdx and threadIdx, which are the block and thread indexes, and blocks and threads, which are the number of blocks and the number of threads per block, respectively.

At line 1, each block uses its blockIdx index to compute the prefix length that it is responsible for. The parameter *BlocksPerLength* is passed to the kernel in order to control how many blocks are used per prefix length. Line 2 computes the relative index (*i.e.,* from 0 to *BlocksPerLength*-1) among the blocks responsible for this prefix length. Line 3–5 show the partitioning of the content names among these blocks. Line 3 uses the batch size $|C|$ to compute the number of content names that the each block must look up. Line 4 computes the offset of the current block in C, and line 5 computes the index of the first name outside the block range. The index of the first content name to be read by the thread is computed in line 6.

Lines 7–15 are the core of the LPM. In each iteration, a thread loads a different content name c (line 8), transforms it into a prefix p (line 9), and performs a Bloom filter lookup (line 10). If a match is found, a hash table lookup is performed (line 12), and line 13 makes sure that the match is not a false positive. Finally, if an entry was found, the prefix length is written to $L[tid]$ using the AtomicMax call (line 14). The AtomicMax(a, v) call is provided by the GPU to write a value v to a given address a only if v is higher than the contents of a. The operation is atomic across all SMs, and thus line 14 ensures LPM is realized. The thread index is then increased in line 15 and matching is initiated on the next content name.

6. EVALUATION

This section experimentally evaluates Caesar along with its extensions and the name-based LPM algorithm. First, Section 6.1 presents the experimental setting and methodology. Section 6.2 then presents results from a series of microbenchmarks to properly dimension the PBF, key data structure in our name-based LPM algorithm. Finally, Section 6.3 and 6.4 evaluate both Caesar and its extensions, namely distributed forwarding and GPU offloading.

6.1 Experimental Setting

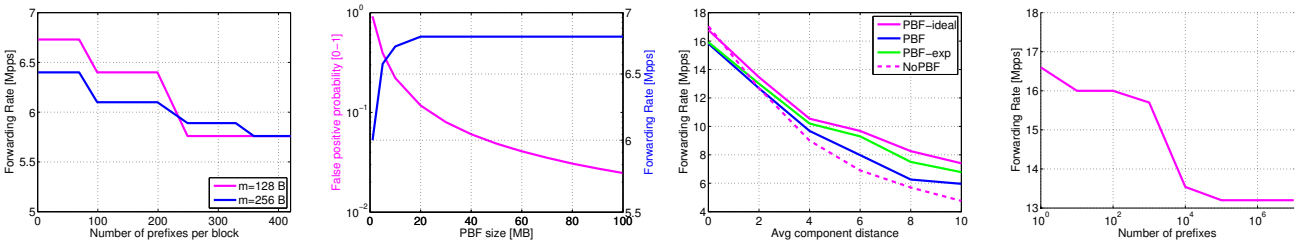
Using optical fibers, we connect Caesar to a commercial traffic generator equipped with 10 Gbps optical interfaces. For easy of presentation, we assume that the four line cards in Caesar work in half-duplex mode, two for input traffic and two for output traffic. Nevertheless, results can be extended to full-duplex configurations, as well as to a larger number of line cards. To support content names, the traffic generator produces regular IP packets with our name-based header as payload (cf. Section 4). Each experiment then consists of three parts: (1) traffic with desired characteristics is originated at the traffic generator and transmitted to Caesar; (2) packets are received by Caesar's line cards and content names are extracted; and (3) forwarding decisions are made and packets are sent back to the generator. For each experiment, we mainly measure the *forwarding rate* and *packet latency*. The forwarding rate is measured as the highest input rate that Caesar can handle, during 60 seconds with no losses. Packet latency is described by the minimum, maximum, and average latency of the packets forwarded within the selected 60 seconds time-frame.

We call *workload* the combination of a set of content prefixes stored in Caesar's FIB, and content names requested via the traffic generator. We derive a *reference* workload from the trace described in [9]; this trace contains 10 million URLs collected by crawling the Web. The assumption here is that the hostname extracted from an URL is representative of a content prefix in ICN. Content names are then generated by adding random suffixes to content prefixes randomly selected from the trace; this is the same procedure used in [9], and produces content names that are 42-Bytes long. Overall, most content prefixes in the reference workload are short, with only 2 components on average, whereas content names have between 3 and 12 components, with 4 components on average. The *average distance* Δ between content names and their matching prefixes is only equal to 2 components. To avoid the effect of congestion and traffic management, we assume next hops associated with content prefixes are uniformly distributed over the two output line cards.

Throughout the evaluation, we also use *synthetic* workloads to assess the impact of system parameters and traffic characteristics on Caesar's performance. Synthetic workloads are generated from the reference workload by varying the following parameters: (1) the average distance Δ, which affects the number of potential PBF/hash table lookups, as well as the complexity of the hashing operation; (2) the number of content prefixes in the FIB, which affects the FIB size and access speed; and (3) the number of content prefixes sharing the first component, which affects the distribution of prefixes among PBF blocks, and thus the false positive rate.

6.2 PBF Dimensioning

We start by motivating the choice of the number of hash functions k used in the PBF. The goal is to minimize the cost of computing seed hash values, as this operation has a high computation time (cf. Table 2). After extensive investigation, we set $k = 2$ since the generation of additional seed hash values significantly hurts Caesar's forwarding rate, with only a marginal false positive reduction.



(a) Single input line card. (b) Single input line card. (c) Two input line cards. (d) Two input line cards.

Figure 5: PBF dimensioning and Caesar's evaluation. (a) Forwarding rate as a function of the number of prefixes per block. (b) False positive and forwarding rate as a function of the total PBF size. (c) Forwarding rate as a function of average component distance Δ. (d) Forwarding rate as a function of the number of prefixes.

We now focus on PBF dimensioning. Figure 5(a) shows Caesar's forwarding rate in millions of packets per second (Mpps) as a function of the number of content prefixes per PBF block. The block size is set to one and two cache lines, corresponding to 128 and 256 bytes, respectively, For simplicity, we assume a single line card is active, and use synthetic workloads. The figure shows a key result: *the fastest forwarding rate is measured when a block fits in a single cache line and there are less than 100 content prefixes per block*. Therefore, for the rest of the evaluation we set the block size m equal to 128 bytes and the expansion threshold t_i for a block i to 75 prefixes, *i.e.,* the largest value which does not reduce the forwarding rate, cf. Figure 5(a).

Figure 5(a) shows another interesting result. When a block contains less than 200 prefixes, increasing the block size slightly reduces Caesar's forwarding rate. A larger block size is instead beneficial to the forwarding rate when the block contains more than 200 prefixes. It comes with no surprise that, overall, a larger block size provides a lower false positive rate for the same amount of content prefixes per block. Accordingly, 200 prefixes per block is the threshold for which the additional memory accesses required to load a larger block are amortized by the lower false positive rate.

Figure 5(b) shows Caesar's forwarding rate as a function of the total PBF size $s = m \times b$, where b is the number of PBF blocks, for the reference workload, *i.e.,* 10 million content prefixes. As above, just one line card is active. When $s < 20$ MB, the forwarding rate quickly grows from 6 to 6.7 Mpps. For $s > 20$ MB, the forwarding rate is constant at 6.7 Mpps. As above, this effect is due to the fact that the false positive rate quickly flattens out as s increases. Accordingly, we set the PBF size s to 30 MB, which is the minimum PBF size that maximizes the forwarding rate.

Based on these parameters ($k = 2$, $m = 128$ bytes, $t_i = 75$), we compute the number of expansions required per prefix in the reference workload. We find that a single expansion is enough to handle 95% of the content prefixes; however, 1% of the prefixes incur four expansions, which is the maximum number of expansions required to handle content prefixes from the reference workload.

6.3 Caesar

This section evaluates Caesar. For completeness, we consider several variants of the first stage of our name-based LPM algorithm (cf. Section 3): (1) *PBF*, where the PBF is used without expansion, (2) *PBF-exp*, where PBF expansion is enabled, (3) *NoPBF*, where no PBF is used. In the *NoPBF* case, all possible content prefixes originated from a requested content name are looked up directly in the hash table, from the longest to the shortest prefix. As an upper bound for performance, we introduce the *PBF-ideal*. This consists of using a PBF with expansion while assuming an ideal synthetic workload where all content prefixes differ in their first component. In this case, content prefixes are uniformly distributed among PBF blocks.

In the remainder of this section, we first evaluate Caesar's performance assuming the reference workload. Then, we present a sensitivity analysis that leverages several synthetic workloads to quantify the impact of workload characteristics on Caesar.

Reference workload: We start by measuring Caesar's forwarding rate under each of the four variants: *PBF*, *PBF-exp*, *NoPBF*, and *PBF-ideal*. The reference workload is used for all variants, with the exception of *PBF-ideal*, that uses the ideal workload. Table 1 summarizes the results from these experiments, differentiating between the cases of one and two active line cards for the forwarding rate. We first focus on the forwarding rate achieved under the *PBF-exp* variant. Assuming a single line card, the table shows that Caesar supports a maximum of 6.6 Mpps when a matching content prefix is found in the FIB (*Match*), and up to 7.5 Mpps when no matches are found (*No Match*), *i.e.,* the corresponding packet is forwarded to a default route. At 10 Gbps, 6.6 Mpps translates to a minimum packet size of 188 Bytes. The table also shows that doubling the active line cards doubles the overall forwarding rate. Accordingly, *Caesar sustains up to 10 Gbps input traffic per line card assuming a minimum packet size of 188 Bytes, and a FIB with 10 million content prefixes.* In the remainder of this paper, we focus on results and experiments where two line cards are active.

Table 1 also shows that *PBF-exp* pays only a 2% reduction of the forwarding rate compared to the ideal case (*PBF-ideal*). This reduction of the forwarding rate is due to the additional memory accesses and complexity required by *PBF-exp* to deal with the non-uniform distribution of content prefixes among blocks. Compared to the *PBF-exp* variant, the absence of the expansion mechanism (*PBF*) costs an additional 2% reduction of the forwarding rate; this is due to the high false positive rate in overloaded PBF blocks. Surprisingly, Table 1 shows that *PBF-exp* only gains about 5% on a solution without a PBF (*NoPBF*), *i.e.,* a forwarding rate of 13.1 Mpps

	PBF-ideal	*PBF-exp*	*PBF*	*NoPBF*
Fwd Rate *Match* (Mpps)	6.7/13.3	6.6/13.1	6.3/12.5	6.3/12.5
Fwd Rate *No Match* (Mpps)	7.5/14.9	7.5/14.9	6.3/12.5	5.2/10.2
Min. latency (μs)	5.4	5.6	5.8	5.8
Avg. latency (μs)	6.4	6.5	6.9	7.0
Max. latency (μs)	8.1	8.1	9.4	9.9

Table 1: Caesar's forwarding rate (Mpps) and latency (μs) with the reference workload. For the forwarding rate, we differentiate between experiments with one and two line cards.

	Total	I/O processing	Hashing	HT lookup	*PBF* lookup
PBF-ideal	1412	371	363	384	294
PBF-exp	1553	371	440	385	357
PBF	1763	371	440	656	294
NoPBF	1781	371	462	948	-
Atomic	-	*371*	*107*	*251*	*129*

Table 2: CPU cycles per operation.

versus 12.5 Mpps. Such small gain is due to the simplicity of the reference workload where Δ, the average distance between a name and its matching prefix, is low (*i.e.,* 2 components on average). In this case, the *NoPBF* variant requires, on average, only two extra hash table lookups to perform LPM compared to both *PBF* and *PBF-exp*. Larger gains from the *PBF* data structure are showed later under the presence of adversarial workloads. Nevertheless, Table 1 shows that *PBF-exp* gains about 30% over *NoPBF* when none of the incoming content names matches a FIB entry. This result suggests that *the PBF-exp is robust to DoS attacks, where an attacker generates non-existing content names to slow down a content router.*

We now focus on packet latency. Table 1 indicates that *PBF-exp* provides a slightly lower latency than both *PBF* and *NoPBF*, on average. In fact due to the simplicity of the reference workload, the switch is responsible for most of the packet latency, and the impact of the name-based LPM variant on the average latency is minimal. The maximum latency, however, shows significant difference. *PBF-exp* reduces the maximum latency by more than 15% compared to both *PBF* and *NoPBF*. The maximum latency is due to packets whose content names have many components (e.g., 12), and a high average distance Δ from prefixes in the FIB. In this case, the algorithmic benefit of *PBF-exp* plays a role as LPM starts contributing to the overall latency. The table also shows that the expansion mechanism only causes a 2-4% latency increase with respect to the ideal case.

We dissect Caesar's performance bottlenecks by tracking the total number of CPU cycles per major operation, assuming the reference workload. Table 2 reports the number of CPU cycles spent, on average, in the execution of the following operations: I/O processing[2], hashing, hash table lookup, and *PBF* lookup. We differentiate between *PBF*, *PBF-exp*, *NoPBF*, and *PBF-ideal*; we also investigate the cost of each operation in isolation (*Atomic* in the table), *i.e.,* the CPU cycles for a single execution of an operation.

Overall, the results for the total CPU cycles in Table 2 confirm the trend showed in Table 1, with *PBF-ideal* being the least and *NoPBF* being the most CPU hungry. In isolation (the *Atomic* row), I/O processing and hash table lookup require the most CPU cycles. However, while I/O processing is performed once per packet, hash table lookup might be performed multiple times according to the LPM variant adopted. Accordingly, the hash table lookup operation accounts for a minimum of 25% (*PBF-exp*) and a maximum of 50% of the CPU cycles (*NoPBF*). This result showcases the algorithmic advantage of the *PBF-exp* in reducing the number of hash table lookups. Conversely, *PBF-exp* requires some additional CPU cycles for the PBF lookup, since occasionally more than one block might be loaded, e.g., *PBF-exp* requires on average 357 CPU cycles whereas both *PBF* and *PBF-ideal* require only 294 cycles. Finally despite hashing per se is not CPU-intensive, on average, it accounts for 25% of the total number of cycles since several seed hash values are computed (cf. Section 3.3).

[2]I/O processing consists of header parsing, MAC address lookup, and header rewriting for packet switching.

Sensitivity analysis: We now analyze the impact of different workload characteristics on Caesar's performance. We start by varying the average distance Δ between the content prefixes in the FIB and the requested content names. The parameter Δ is key to properly characterize a given workload, since it defines the complexity of the LPM operation.

Figure 5(c) shows Caesar's forwarding rate as Δ grows from 0 (equivalent to exact matching) to 10 (highly adversarial workload); as usual, we distinguish between *PBF-ideal*, *PBF*, *PBF-exp* and *NoPBF*. Overall, the rate decreases as Δ increases, which is expected since the number of seed hash values increases linearly with Δ. As Δ increases, the performance gap between *PBF-exp* and *NoPBF* increases too, *i.e.,* when $\Delta = 10$, *PBF-exp* guarantees a forwarding rate twice as fast as the *NoPBF* variant. Compared to *PBF*, *PBF-exp* adds a penalty when Δ is small, which is absorbed as Δ increases. This set of results suggests that *the PBF-exp is robust to adversarial workloads and variable traffic patterns.*

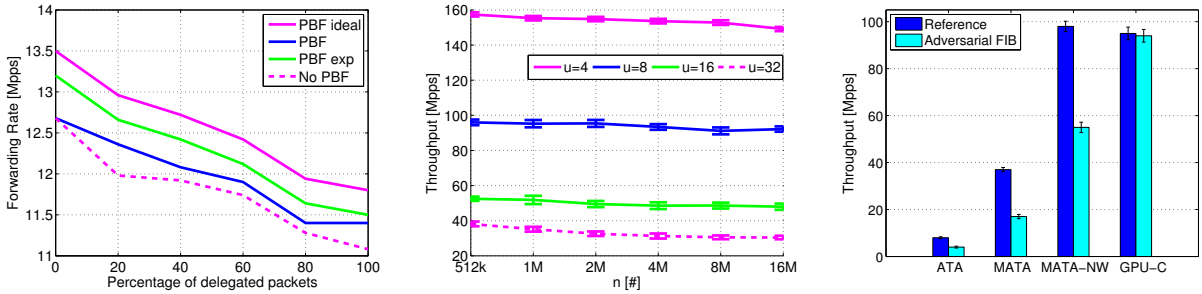
We now investigate the impact of the number of content prefixes n in the FIB. Figure 5(d) plots the evolution of the forwarding rate as n grows from 1 content prefix up to 10 million, as in the reference workload. Overall, the forwarding rate follows a step function, with a large drop in the forwarding rate for $n > 1000$, *i.e.,* from 8.3 to 6.6 Mpps. This phenomenon depends on the hierarchical memory organization of the NPU. When $n = 1$, the only content prefix quickly propagates from DRAM to the L1 cache of every core. As the number of prefix grows, the network processor efficiently stores the prefixes in the L2 cache; after 1,000 prefixes, the L2 caches is exhausted and most prefixes are fetched from the off-chip DRAM, which causes the rate drop. After the 1,000 prefixes threshold, the forwarding rate is almost constant: this indicates that *the amount of prefixes that Caesar support is limited by the amount of off-chip DRAM*. Therefore, with additional DRAM, Caesar could support more content prefixes with little impact on the forwarding rate. Such additional memory is largely available in both edge and core routers. Implementing Caesar on such platforms would allow storing one to two orders of magnitude more prefixes, while still guaranteeing name-based forwarding at wire speed. This is part of our future work.

6.4 Distributed Forwarding

This section evaluates the distributed forwarding extension used by Caesar to allow very large FIBs without requiring additional DRAM (cf. Section 5.1). We populate each input line card with a disjoint set of 10 million content prefixes, 20 millions in total, originated by modifying few characters from the 10 million prefixes in the reference workload. Since Caesar has a switch with a capacity of 10 Gbps per line card, and distributed forwarding requires twice the overall switching speed in the worst case (cf. Section 5.1), we limit the traffic at 5 Gbps per line card and halve the minimum packet size from 188 to 94 bytes.

Figure 6(a) shows Caesar's forwarding rate as a function of ρ, the fraction of packets that require going to another line card for name-based LPM. Overall, the forwarding rate slowly decreases as ρ increases. In the worst-case scenario, $\rho = 100\%$ and these operations account for a drop of only 15% in the rate *i.e.,* from 13.2 to 11.5 Mpps for *PBF-exp*. This reduction of the forwarding rate is due to additional operations required by distributed forwarding, namely packet dispatching, and MAC address rewriting.

We also estimate the impact of distributed forwarding on packet latency in the worst case, *i.e.,* $\rho = 100\%$. We find that distributed forwarding causes an increase of the average and minimum latency in Caesar by 50%. As previously discussed, minimum and average latency mostly derives from the switching latency which doubles



(a) Distributed forwarding. (b) GPU offloading. (c) GPU offloading comparison.

Figure 6: Evaluation of Caesar's extensions. (a) Forwarding rate as a function of ρ in distributed forwarding. (b) Throughput as a function of FIB size n and maximum prefix length u in GPU offloading. (c) GPU offloading comparison with [9].

with distributed forwarding. The maximum latency grows instead by about 30%, and this happens when LPM latency overcomes the switching latency, *i.e.,* in presence of large values of Δ. In any case, the additional latency remains in the order of microseconds and it is thus tolerable even for delay-sensitive applications.

To summarize, *distributed forwarding extends Caesar to support twice as many content prefixes with a reduction of only 15% in the forwarding rate and an additional delay of a few microseconds.*

6.5 GPU Offloading

This section quantifies the speedup that GPU offloading provides to Caesar's forwarding rate. We assume a line card offloads a batch of 8K content names to a GTX 580 GPU [19]. Such batch ensures high GPU occupancy, and a maximum buffering delay of about 1 ms, assuming 188-byte packets and 10 Gbps. Larger packet sizes or slower speeds, which both cause higher delay in forming a batch, can be easily handled by Caesar without the GPU (Section 6.3).

We first measure how many packets per second the GPU can match as a function of the number of content prefixes n in the FIB. Figure 6(b) reports the *throughput* only from the kernel execution time, *i.e.,* we omit the transferring time between line card and GPU, and vice-versa, to be comparable with [9] and not limited by the PCIe bandwidth problem discussed therein. We generate several synthetic FIBs where the number of content prefixes grow exponentially from 0.5 to 16 million, the maximum number of prefixes that fits in the GPU device memory. We also vary the maximum length u of the prefixes in the FIB between 4 and 32 components. For each value of u, content prefixes in the FIB are equally distributed among the possible lengths, e.g., when $u = 4$, a quarter of the prefixes have a single component. Finally, we assume that all content names have 32 components, *i.e.,* $d = 32$.

Figure 6(b) shows two main results. First, the throughput is mostly independent from the number of prefixes n; overall, growing the FIB size from 0.5 to 10 million prefixes causes less than a 10% throughput decrease. Second, the throughput largely depends on u. For example, when $n = 16$ M, increasing u from 4 to 32 components reduces the throughput by 5x, from 150 to 30 Mpps.

We now compare our implementation with the work in [9], which also explores the usage of GPU for name-based forwarding. Their GPU code is open-sourced, which allows us to perform a fair comparison with our implementation. The key idea of the work in [9] is to organize the FIB as a trie as done today for IP. They thus introduce a character trie which allows name-based LPM. Then, they introduce three optimizations, namely the aligned transition array (ATA), the multi-ATA (MATA) and MATA with interweaved name storage (MATA-NW), which leverage a combination of hashing and the hierarchical nature of the content names to realize efficient compression and lookup.

Figure 6(c) compares the performance of our kernel (GPU-C) with the kernels proposed in [9], namely ATA, MATA and MATA-NW by running their code on our GPU. For such comparison we use the reference workload, where $u \sim 3$, as well as a more adversarial workload where $u = 8$. We refer to this adversarial workload as "adversarial FIB."

Compared to the results presented in [9], we measure less than half the throughput for ATA, MATA and MATA-NW. This is expected, since our GPU has half the cores than the GTX 590 GPU used in [9]. The figure also shows that the throughput measured for Caesar, about 95 Mpps, matches the results from the synthetic traces when $u = 8$ and $n = 10$ M, cf. Figure 6(b). MATA-NW is slightly faster than Caesar, 100 versus 95 Mpps, assuming the reference workload. This happens because MATA-NW exploits the fact that most of the content prefixes in the FIB are very short, e.g., 2 or 3 components, to reduce LPM to (mostly) an exact matching operation. Instead, our algorithm does not rely on such assumption; this design choice makes it resilient to more diverse FIBs at the expense of a performance loss with a simplistic FIB. Such feature is visible in the presence of the adversarial FIB, where Caesar is twice as fast as MATA-NW.

To summarize, *GPU offload augments Caesar's forwarding rate by an order of magnitude, with a small penalty in packet latency, and our GPU-based LPM algorithm is resilient to adversarial traffic workloads.*

7. ADDITIONAL FEATURES

Name-based forwarding is the key task of a content router. Additional features are caching, multicasting, and dynamic multipath forwarding. To support these features, a Pending Interest Table (PIT) and a Content Store (CS) are required. The PIT keeps track of pending content requests, or "Interest" in the NDN terminology, already forwarded by the content router. The CS stores a copy of forwarded data packets to satisfy eventual future requests.

The design, implementation, and evaluation of PIT and CS is out of the scope of this paper, and left as future work. However, we have recently started extending Caesar with both PIT and CS based on a set of design guidelines derived in our previous work [20, 5]. In the following, we briefly summarize such integration.

In [20], we identify two challenges in PIT design: *placement* and *data structure*. Placement refers to where in the content router the PIT should reside. Data structure refers to how the PIT entries should be stored and organized to enable efficient operations. The paper concludes that the best approach is a third-party place-

ment leveraging the semantic of content names to select a line card where PIT matching is performed. This idea fits well the distributed forwarding scheme used by Caesar, which we plan to piggyback for the PIT implementation. As data structure, we use an open-addressed hash table (cf. Section 3.4).

The CS consists of a packet store, where data packets are physically stored, and an index table, that keeps track of data packet memory locations in the packet store. Similar to the PIT, the index table is implemented as an open-addressed hash table. In addition to pointers to data packets, the index table stores data statistics, e.g., access frequency and timestamps, to enable replacement policies like FIFO or LRU. We implement the packet store by an extension to Caesar's packet buffer in order to allow Caesar to store data packets after forwarding as well as serve them when needed. An evinction mechanism was also added to support the removal of data packets according to the replacement policy. The CS is physically allocated on the off-chip DRAM memory. Additional levels of storage on lower throughput/higher capacity technologies (e.g., SSD) can complement the packet store design; however, this optimization is not supported by our current hardware setup.

8. CONCLUSION

The Internet usage is currently centered around content distribution, instead of the original host-to-host communication. Future Internet architectures are thus expected to depart from a host-centric design to a content-centric one. Such evolution requires routers to operate on content names instead of IP addresses. A high burden is expected on the routers due to the explosion of the address space, both in number of content prefixes, which are hard to aggregate compared to IP, and their length, expected to be on the order of tens of bytes as opposed to 32 or 128 bits for IPv4 and IPv6, respectively. Our paper investigates the design and implementation of Caesar, a content router capable of forwarding packets based on names at wire speed. Caesar advances the state of the art in many ways. First, it introduces the novel prefix Bloom filter (PBF) data structure to allow efficient longest prefix matching operation on content names. Second, it is fully compatible with current protocols and network equipment. Third, it supports packet processing offload to external units, such as graphics processing units (GPUs), and distributed forwarding, a mechanism which allows line cards to share their FIBs with each other. Our experiments show that Caesar sustains up to 10 Gbps input traffic per line card assuming a minimum packet size of 188 bytes, and a FIB with 10 million content prefixes. We also show that the two proposed extensions allow Caesar to support both a larger FIB and higher forwarding speed, with a small penalty in packet latency.

ACKNOWLEDGMENTS

This work has been partially carried out at the Laboratory of Information, Networking, and Computer Science (LINCS), and results have been partially produced in the framework of the common research laboratory between INRIA and Bell Labs, Alcatel-Lucent.

9. REFERENCES

[1] "Akamai," http://www.akamai.com/.

[2] "Amazon elastic compute cloud (amazon ec2)," http://aws.amazon.com/ec2/.

[3] G. Carofiglio, G. Morabito, L. Muscariello, I. Solis, and M. Varvello, "From Content Delivery Today to Information Centric Networking," *Computer Networks*, 2013.

[4] V. Jacobson, D. K. Smetters, J. D. Thronton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Network Named Content," in *Proc. ACM CoNEXT*, Rome, Italy, Dec. 2009.

[5] D. Perino and M. Varvello, "A Reality Check for Content Centric Networking," in *Proc. ACM ICN*, Toronto, Canada, Aug. 2011.

[6] W. So, A. Narayanan, and D. Oran, "Named Data Networking on a Router: Fast and DoS-Resistant Forwarding with Hash Tables," in *Proc. IEEE/ACM ANCS*, San Jose, California, USA, Oct. 2013.

[7] G. Pankaj, L. Steven, and M. Nick, "Routing Lookups in Hardware at Memory Access Speeds," in *Proc. IEEE INFOCOM*, San Francisco, CA, Mar. 1998.

[8] W. So, A. Narayanan, D. Oran, and M. Stapp, "Named Data Networking on a Router: Forwarding at 20Gbps and Beyond," in *Proc. ACM SIGCOMM (demo)*, Honk Kong, China, Aug. 2013.

[9] 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 *Proc. NSDI*, Lombard, IL, Apr. 2013.

[10] H. Yuan, T. Song, and P. Crowley, "Scalable NDN forwarding: Concepts, Issues and Principles," in *Proc. ICCCN*, Bundeswehr Munchen, Jul. 2012.

[11] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen, "Scalable Name Lookup in NDN Using Effective Name Component Encoding," in *Proc. ICDCS*, Macau, China, Jun. 2012.

[12] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, "NameFilter: Achieving Fast Name Lookup with Low Memory Cost via Applying Two-Stage Bloom Filters," in *Proc. IEEE INFOCOM*, Turin, Italy, Aug. 2013.

[13] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching Using Bloom Filters," in *Proc. ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.

[14] H. Song, F. Hao, M. S. Kodialam, and T. V. Lakshman, "IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," in *Proc. IEEE INFOCOM*, Rio de Janeiro, Brazil, Jul. 2009.

[15] "Information centric networking research group (icnrg)," http://irtf.org/icnrg.

[16] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "NLSR: Named-data Link State Routing Protocol," in *Proc. ACM ICN*, Hong Kong, China, Aug. 2013.

[17] S. Iyer and N. W. McKeown, "Analysis of the Parallel Packet Switch Architecture," *IEEE/ACM Transactions on Networking*, vol. 11, no. 2, pp. 314–324, Apr. 2003.

[18] "Alcatel 7950," http://www.alcatel-lucent.com/products/7950-extensible-routing-system.

[19] "nvidia. gtx 580," http://geforce.com/hardware/desktop-gpus/geforce-gtx-580/.

[20] M. Varvello, D. Perino, and L. Linguaglossa, "On the Design and Implementation of a Wire-Speed Pending Interest Table," in *Proc. IEEE NOMEN*, Turin, Italy, Aug. 2013.