NDN-NIC: Name-based Filtering on Network Interface Card

Junxiao Shi
The University of Arizona
shijunxiao@cs.arizona.edu

Teng Liang
The University of Arizona
philoliang@cs.arizona.edu

Hao Wu
Tsinghua University
wu–h11@mails.tsinghua.edu.cn

Bin Liu
Tsinghua University
liub@tsinghua.edu.cn

Beichuan Zhang
The University of Arizona
bzhang@cs.arizona.edu

ABSTRACT

In Named Data Networking (NDN) content consumers request data by names instead of sending requests to specific destination addresses. This fits shared media particularly well for benefits such as native multicast, mobility support, and fault tolerance. However, since current network interface cards only filter packets based on destination addresses, all NDN packets have to be delivered to software for name-based filtering, resulting in significant CPU overhead. We propose NDN-NIC, a network interface card that filters packets based on their content names, to minimize the CPU overhead while running NDN over shared media. This paper tackles NDN-NIC's main research challenge: using the limited amount of on-chip memory (in tens of kilobytes) to support packet filtering based on a large number of rulesets (in hundreds of thousands). We use Bloom filters to store various name tables on NDN-NIC, and design a number of mechanisms to further adjust the name prefixes that go into the Bloom filters, minimizing false positives under given memory limit. Using traffic traces collected from a department network, simulations show that NDN-NIC with 16KB of memory can filter out 96.30% of all received packets and reduce the main CPU usage by 95.92%.

CCS Concepts

•**Networks → Network adapters;** *Packet classification;*

Keywords

Named Data Networking; Network Interface Card

1. INTRODUCTION

Named Data Networking (NDN) [8, 21] is a new Internet architecture that changes the network semantic from *packet delivery* to *content retrieval*. It promises benefits in areas such as content distribution, security, mobility support, and application development. In NDN, Interest packets carrying content names are sent to retrieve contents rather than to a particular destination identified by an address. This model fits shared media particularly well, where a single transmission is received by every node within range. For example, when a node requests a web page, any node with the content can reply, without the requester having to figure out or specify a particular receiver. It makes content retrieval more efficient, more scalable, and more fault-tolerant.

In shared media, however, there are also many packets that a receiver wants to filter out because they are not of interest to the receiver. In wireless networks, for instance, signals are transmitted to every node within radio range. Traditionally packet filtering is conducted at the network interface cards (NIC) based on the destination MAC address, which is not applicable to NDN traffic because NDN packets do not carry source or destination address. The current implementation of NDN over Ethernet is to multicast NDN traffic to all NDN nodes, and let NDN software conduct name-based filtering at user space. Compared to filtering on NIC, user-space filtering not only incurs significant overhead to the main CPU, but also increases system power consumption because the system can be awoken by irrelevant packets. Although it is possible to treat the shared media as a group of point-to-point links, put unicast MAC addresses on NDN packets, and utilize NIC's address filtering to reduce the overhead, doing so will lose the benefits of NDN and fall back to the address-based network architecture.

In order to reduce CPU overhead and system power consumption while keeping NDN's benefits, we propose **NDN-NIC**, a network interface card that can filter out irrelevant NDN packets based on their names. The main technical challenge is how to support scalable name-based filtering using only a small amount of on-chip memory. This paper designs data structures and algorithms to realize efficient name-based filtering with small false positives using tens of kilobytes of memory, commonly available on conventional, low-cost NICs.

A typical NDN end host may have hundreds of thousands name rulesets due to the content it serves, the Interests it sends, and the content cache it maintains. It is infeasible to fit all these rulesets into the limited amount of memory on network cards. We propose to store these rulesets in three Bloom filters (BFs) on NDN-NIC and conduct packet filtering based on these BFs. A unique property of BF is the guarantee of no false negative, which means NDN-NIC will not drop any packet that should be accepted. However, as BF has false positives, some unwanted packets may pass the filter and be delivered to the system. The technical chal-

ICN'16, September 26-28, 2016, Kyoto, Japan
© 2016 ACM. ISBN 978-1-4503-4467-8/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2984356.2984358

lenge is to keep NDN-NIC's false positive as low as possible while only using a small amount of memory on the NIC and minimal CPU overhead in maintaining the data structures, thus the overall benefits in CPU usage reduction and power consumption can be achieved.

To have a compact representation of the rulesets in NDN-NIC, we need to carefully choose which names or name prefixes should be stored in the Bloom filters. In this paper we have designed and evaluated a number of techniques that can reduce the number of names stored in BFs but may make name matching less accurate; the overall result is reduced memory footprint with acceptable false positives and no false negative.

We evaluated the design of NDN-NIC by running traffic traces collected from a university department network, and simulating packet filtering and Bloom filter maintenance. Using an NDN-NIC equipped with 16KB memory, our design can filter out 96.30% of all received packets, and reduce CPU usage by 95.92% compared to a regular NIC.

This paper is organized as follows. Section 2 introduces the background of packet processing in NDN. Section 3, 4, and 5 explain the design of NDN-NIC. Section 6 evaluates the performance of NDN-NIC with different settings. Section 7 summarizes related work. We conclude the paper in Section 8.

2. BACKGROUND

In this section, we first introduce conventional address-based filtering on NICs. Then we describe the three tables used in NDN forwarding, including their name matching rules and implementation, where the challenges of name-based filtering stem from. We also briefly introduce Bloom filters, the primary data structure used in NDN-NIC design.

2.1 Address-based Filtering on NICs

A conventional Ethernet NIC has an address-based filtering logic that finds a match between the destination MAC address of an incoming packet and a list of acceptable unicast and multicast addresses. Most NICs implement the list as an array of addresses and conduct exact match: a packet is delivered to the system if its destination address equals to any address in the array. The size of the address list varies among different NICs, e.g., according to driver implementations in Linux kernel 4.4, Intel PRO/Wireless 2200BG keeps one address, while Broadcom BCM4401-B0 can store up to 32 addresses.

Some NICs also use a hash table to accommodate more addresses (for instance, the Linux driver of Intel 82540EM uses a 4096-bit hash table; each accepted address is represented as a bit in the hash table, and a packet passes the filter when the hash value of its destination address hits a bit set to 1. Hash table may incur false positives due to hash collisions.

Moreover, almost every NIC supports "ALLMULTI" mode, which accepts all multicast packets by simply checking a bit in the destination address that identifies a multicast address.

2.2 Name Matching in NDN

NDN communication is receiver-driven: a consumer first sends an *Interest* packet to the network, and then a *Data* packet flows back along the same path in reverse direction. Both Interest and Data carry content names. A Data packet can satisfy an Interest if the Interest name is the same as the Data name or a prefix of the Data name [12].

NDN forwarding is currently implemented in a user-space software program, the NDN Forwarding Daemon (NFD) [2]. NFD makes forwarding decisions based on the name rulesets stored in three *tables*: *Forwarding Information Base* (FIB), *Pending Interest Table* (PIT), and *Content Store* (CS).

For an incoming Interest with name `/A/b`: (1) NFD queries the CS for any Data whose name starts with `/A/b`. This could match `/A/b`, `/A/b/1`, `/A/b/1/2`, `/A/b/3`, etc. If one or more matches are found, the "best" match as determined by a selection procedure is returned to the requester. (2) NFD inserts a PIT entry for the Interest if it doesn't already exist, which involves an *exact match* for `/A/b`, and then forwards the Interest according to the result of FIB lookup. (3) NFD performs a *longest prefix match* lookup on the FIB. This could match FIB entries `/A/b`, `/A`, or `/`. If there's no match, the Interest is dropped.

For an incoming Data with name `/A/b/1`: (1) NFD queries the PIT to find all pending Interests this Data can satisfy. This would match any PIT entry whose name is a prefix of `/A/b/1`, such as `/`, `/A`, `/A/b`, and `/A/b/1`. If no match is found, the Data is dropped. (2) NFD puts the Data into the CS, which involves an *exact match* lookup.

In NFD implementation, FIB, PIT, and CS are stored on the **name tree**, a tree structure that follows the name hierarchy. Each node is identified by a name, and can attach FIB/PIT/CS entries with that name; each edge goes from a shorter name to a longer name with one more name component. Having the name tree allows the three tables to share a common index structure, which not only reduces memory consumption, but also saves computation overhead during table lookups.

2.3 Bloom Filter

Bloom filters [3] (BFs) are widely used in network processing to implement constant-time set membership testing with efficient memory usage. A Bloom filter is defined by allocating an array of m zero-initialized bits, and selecting k independent hash functions that take a key as input and output a number within range $[0, m-1]$. To add a key to the BF, we compute its hashes with each of k hash functions and get k positions in the array, and then set the bits at these positions to 1. To query whether a key is in the BF, we compute its k hashes with the same hash functions, and check the bits at these positions. If any of these bits is 0, the key is definitely not in the set; otherwise, we consider the key in the set. However, these bits could have been set to 1 due to other added keys, resulting in a **false positive**.

After adding n keys into an m-bit BF with k hash functions, its false positive probability [7] is:

$$p \approx \left(1 - \mathrm{e}^{-kn/m}\right)^k \tag{1}$$

This equation tells us that, given a BF with fixed size, when we add more keys (higher n), there would be more false positives. On the other hand, with same number of keys, a larger BF (larger m) has less false positives than a smaller BF.

A standard Bloom filter does not support key removal, because we do not know how many keys have caused a bit to be set to 1. Counting Bloom filter (CBF) [7] overcomes this shortcoming by replacing each bit in the BF's array with a counter. The counter is incremented when adding a key hashed to this array position, and is decremented when removing that key. To query whether a key is in the CBF, we check whether the counters at array positions chosen by the hash functions are all non-zero.

3. DESIGN OVERVIEW

Name-based filtering in NDN brings three challenges over address-based packet filtering. First, unlike most IP hosts that only have a handful of unicast/multicast addresses, an end host on the NDN testbed can have hundreds of FIB and PIT entries, and hundreds of thousands of CS entries; a consumer grade NIC has only tens of kilobytes of memory, and cannot fit a copy of all those names. Second, IP and Ethernet addresses are relatively stable, but name rulesets in the three tables, especially PIT and CS, are changing frequently, which requires NDN-NIC to be updated accordingly. Last but not least, Ethernet packet filtering only needs exact match, but NDN adopts different name matching rules for each table (Section 2.2), which requires NDN-NIC to support the same semantics on name-based filtering.

We believe that Bloom filter (BF) is an ideal data structure for NDN-NIC. Benefit from its efficiency in memory, hundreds of thousands of variable-length names can be stored in a BF that only takes a fixed amount of memory. Although BF brings possibility of false positive, there is no false negative, so that every non-matching packet can be safely dropped. In case a packet passes the filter due to BF false positive, NFD will find the packet not matching any table, and then drop it; this incurs some CPU overhead, but causes no correctness concern. Using Bloom filters allows us to fit a compact representation of hundreds of thousands of name rulesets into the small amount of memory present on the NIC.

BFs do not support key removal (unless we clear the BF and rebuild it from scratch), which is necessarily when a table entry is deleted. Counting Bloom filters (CBFs), in contrast, support key removal, but consume more memory because every single bit is replaced with a multi-bit counter. NDN-NIC utilizes a hybrid usage of BFs and CBFs. We maintain CBFs in software, and mirror their contents into standard BFs in hardware. Therefore, we can support key removal without increasing memory requirements on the NIC.

BF queries are exact match membership tests. NDN name matching not only uses exact match queries, but also has two types of prefix match queries: (a) a name in the table is a prefix the name of an incoming packet; (b) the name of an incoming packet is a prefix of a name in the table. As introduced in Section 2.2, (a) is used to query incoming Interest names on the FIB and to query incoming Data names on the PIT; (b) is used to query incoming Interest names on the CS. Furthermore, incoming Interests are matched against FIB and CS but not PIT, while incoming Data are matched against PIT but not FIB or CS. This warrants a separate BF for each table, and different lookup method on each BF. BF-FIB and BF-PIT store FIB or PIT entry names, and implement (a) by querying the BF with every name prefix of an incoming Interest/Data. BF-CS stores all prefixes of CS entry names, and implements (b) by querying the BF with the incoming Interest's name. This technique allows us to support different name matching rules as per NDN semantics.

The overall architecture of an NDN host equipped with

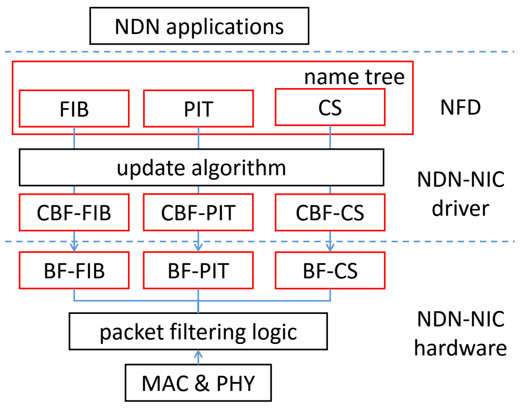


Figure 1: Overall architecture

NDN-NIC is depicted in Figure 1. The NDN-NIC design consists of two components: the **NDN-NIC hardware**, a network interface card that filters incoming NDN packets (Section 4), and the **NDN-NIC driver**, an NFD module that notifies hardware what to filter (Section 5).

The NDN-NIC hardware performs name-based filtering based on the packet filtering logic and three BFs, BF-FIB, BF-PIT, and BF-CS. When a packet arrives, the packet filtering logic queries the BFs with its name, and makes a deliver/drop decision. The NDN-NIC driver maintains three CBFs in response to changes in FIB, PIT, and CS tables, and updates the BFs on hardware accordingly.

Note that it is unnecessary and invalid to add all table names into the BFs. (1) The FIB contains both entries pointing to local producers, and entries pointing to the network (most notably, the "default route" entry `ndn:/`). It's unnecessary for NDN-NIC to accept an Interest matching a FIB entry in the second category, because an Interest received from a network interface cannot be forwarded out of the same network interface (otherwise it would cause a loop). Therefore, a FIB entry name is added to the NIC only if it has a nexthop other than NDN-NIC itself [1]. (2) The PIT contains both entries representing Interests expressed by local consumers and forwarded to the network, and Interests received from the network requesting Data from local producers. Only the former category of PIT entry names need to be added to the NIC [2]. (3) Unlike FIB and PIT, every CS entry names should be added to the NIC, because the cached Data can be used to satisfy an Interest from the network, regardless of whether this Data was previously received from the network or generated by a local producer.

After knowing what names to put on NIC, we need an update algorithm to generate a compact representation of those names to be downloaded to the NIC. The update algorithm must be able to incrementally adapt to changes in table entries and update the CBFs, whose contents are mirrored into the BFs. We propose three update algorithms: *Direct Mapping*, *Basic CS*, and *Active CS* (Section 5). Direct Mapping

[1]In case of a multi-homed host with multiple NICs, a FIB entry name should be added to an NDN-NIC if it has a nexthop pointing to either a local producer or another NIC.
[2]In case of a multi-homed host with multiple NICs, a PIT entry name should be added to an NDN-NIC if the Interest has been forwarded out of this NDN-NIC, and it's not yet satisfied or timed out.

simply adds every name in a table to the corresponding BF. Basic CS and Active CS are designed to reduce the number of names in the BFs so as to improve filtering accuracy.

This overall architecture is a logical design. In practice, the name-based filter module, i.e., the BFs and packet filtering logic can either be deployed on a NIC or between a conventional NIC and the host system. In this paper, we take the former case to explain our design.

4. NAME FILTER ON HARDWARE

The NDN-NIC hardware is a network interface card that helps to offload NDN packet filtering. In addition to all the components of a regular NIC (PHY, MAC, USB/PCI, etc), we download Bloom filters to the NIC and add a packet filtering logic to support name-based packet filtering.

NFD's FIB, PIT, and CS tables are represented as three BFs on the NDN-NIC (Figure 1). For each incoming packet, the packet filtering logic queries BFs differently according to the packet type. As introduced in Section 3, for an incoming Interest (e.g. `/A/b`), the NIC queries **BF-FIB** with all prefixes of the Interest name (i.e. `/`, `/A`, and `/A/b`) and queries **BF-CS** with the complete Interest name (i.e. `/A/b`). For an incoming Data, the NIC queries **BF-PIT** with all prefixes of the Data name. The packet is delivered to NFD if any BF query finds a match, otherwise the NIC drops the packet.

The packet filtering accuracy of NDN-NIC depends on the characteristics of the Bloom filters, in particular: size of each BF, number of hash functions, and choice of hash functions. They are not only related to false positive probability, but also constrained by the hardware complexity and manufacturing cost. We evaluate different BF sizes and the number of hash functions in Section 6.3. Besides, we choose *H3* hash functions [14], since they can be computed efficiently in hardware logic without requiring a specialized digest chip, and are also simple enough for computing in software.

Although we did not build the NIC hardware, we anticipate that, with careful circuit design, the hardware can achieve high speeds with reasonable manufacturing cost. The BFs are compact enough to fit in fast SRAM or block RAM of a FPGA chip. BF queries can streamlined: we can use k hashing circuits to compute k hashes in parallel while reading the packet, and query the relevant BF after reading each name component.

Since NDN-NIC design requires frequent BF updates in order to keep in sync with NFD's tables, updates to the BFs should be implemented in the data plane as hardware logic, rather than in the control plane. NDN-NIC driver sends BF update commands over PCI/USB in the same way as sending outgoing packets, so that the hardware logic can process BF update commands as fast as they arrive.

5. NDN-NIC DRIVER

The NDN-NIC driver is a software module that keeps BFs on hardware up to date with NFD's FIB, PIT, and CS. Its main components are three Counting Bloom filters (CBFs) and an update algorithm.

Whenever an entry is inserted or deleted in FIB, PIT, or CS, the update algorithm adds or removes names in the CBFs as necessary. Updates to CBFs are then downloaded into BFs on hardware: a non-zero CBF counter value sets the corresponding BF bit to 1, and a zero CBF counter value clears the BF bit to 0. In this way, the BFs have exactly same matching results as the CBFs but smaller memory footprints. Updates to the BFs are applied incrementally: during a CBF update, we keep track of which CBF counters are incremented from zero to non-zero (corresponding BF bits should be set to 1), and which CBF counters are decremented to zero (corresponding BF bits should be cleared to 0). Then the NDN-NIC driver sends BF update commands to the NIC hardware. We first perform 0-to-1 changes on BFs, and then perform 1-to-0 changes. This allows the hardware to continue processing incoming packets on partially updated BFs without danger of false negatives.

One concern is that frequent table updates potentially cause frequent hardware updates. Update commands would compete with outgoing traffic because the host cannot send outgoing traffic to the NIC when an update command is being processed. However, with our proposed algorithms, not every table update would generate a CBF update, and not every CBF update would result in a BF update on hardware. The evaluation of BF update overhead is in Section 6.6.

In the following subsections, we propose one simple update algorithm, Direct Mapping, and two optimizations, Basic CS and Active CS.

5.1 Direct Mapping

Direct Mapping (DM) is straightforward: entries in each table are directly mapped to its corresponding BF. FIB entry names go into BF-FIB, PIT entry names go into BF-PIT, CS entry names and their prefixes go into BF-CS [3].

DM has a reasonable filtering accuracy when tables have a small number of entries compared to BF sizes, but its accuracy degrades quickly with larger tables. According to Equation 1, with perfect hash functions, a 65536-bit Bloom filter has a false positive probability of 4.33% after adding 10000 names. This number quickly grows to 35.96% with 30000 names, and exceeds 50% with 50000 names. In practice, the false positive probability could be even higher due to imperfect hash functions.

As we have observed on the NDN testbed, a typical end host has small FIB and PIT, but a large CS. FIB and PIT are expected to have no more than a few hundred entries each. Suppose there are 100 programs running on the host, each serving 5 name prefixes, there would be 500 FIB entries. If 20 programs are actively being used, each having 40 pending requests (similar to today's web browsers), there would be 800 PIT entries. On the other hand, if we allocate 1GB memory to the CS, assuming an average Data packet size of 4KB, the CS could contain 262144 entries. All these names need to be added to BF-CS; furthermore, in order to overcome BF's exact-match-only limitation, all their prefixes are added as well, which would require more than 500000 names in BF-CS. If we want to keep BF-CS false positive probability under 10% with so many names, 2404160 bits would be required, which will not fit in most NICs.

DM's computation overhead mainly comes from hash function computation. To reduce this overhead, we can compute the hashes once when it's added to a BF for the first time and remember the hash values on the name tree node, trading memory for CPU. DM also incurs high BF update overhead on hardware: CBF is updated on every table change, and unless the BF bits happen to be the same, almost all table

[3]More precisely, the names are updated into CBFs first and then downloaded into BFs on hardware. We omit the CBF step to simplify the explanation.

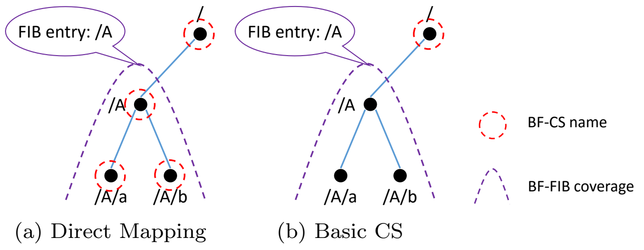
changes require BF updates.

Observing that most names come from the CS, we propose two novel optimizations, Basic CS and Active CS, which add less names to BFs while still ensuring no false negative, so as to improve filtering accuracy and reduce BF update overhead.

5.2 Basic CS

Basic CS reduces false positives in BF-CS by not adding names under existing FIB entries, because Interests with those names are already accepted by matching BF-FIB.

As illustrated in Figure 2, suppose there is a FIB entry `/A`, and two CS entries, `/A/a` and `/A/b`. Direct Mapping (DM) adds `/A` to BF-FIB, and add `/A/a`, `/A/b`, `/A`, `/` to BF-CS. Basic CS only adds `/` to BF-CS, because the other three Interest names will be accepted by NDN-NIC hardware because they can match `/A` in BF-FIB.



(a) Direct Mapping (b) Basic CS

Figure 2: BF-CS usage, DM vs Basic CS

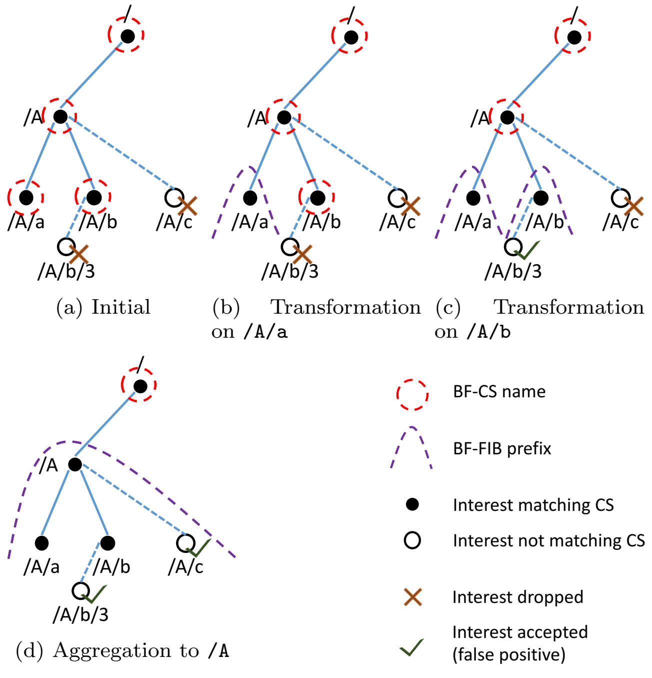
Basic CS is effective when most Data in CS came from local producers, since those Data names fall under FIB prefixes of these local applications which have been added to BF-FIB. In this case, only the prefixes shorter than the FIB entry name (such as `/` in the above example) are added to BF-CS, regardless of how many Data packets are cached. However, the CS also caches Data that were requested by local applications and received from the network; those Data would not match FIB entries in BF-FIB, and Basic CS would be ineffective in this case.

The extra computation overhead of Basic CS, compared to DM, is minimal. When NFD inserts a CS entry, it already needs to walk down the name tree to find the name tree node where the CS entry is stored. To implement Basic CS, the NDN-NIC driver just needs to check each name tree node along the way and see whether it also has a FIB entry, and stop adding names into BF-CS once it finds a FIB entry.

When a producer quits, its FIB entry is deleted but Data generated by this producer is still valid in the cache. This would cause a larger overhead: at that time, the CS still contains Data packets under the former FIB prefix, but that prefix is no longer in BF-FIB. Basic CS would have to re-add CS entry names under the former FIB prefix into BF-CS. However, this occurs infrequently; when this situation does occur, we can re-add names to BF-CS asynchronously, and delay removal of the FIB prefix from BF-FIB until re-adding is completed.

5.3 Active CS

Unlike Basic CS that passively waits to reuse FIB entries, Active CS actively creates the opportunity to reduce BF-CS usage by adding appropriate prefixes into BF-FIB, without requiring corresponding FIB entries.



(a) Initial (b) Transformation on `/A/a` (c) Transformation on `/A/b`

(d) Aggregation to `/A`

Figure 3: Two basic operations in Active CS

Active CS has two basic operations: **Transformation** replaces a single BF-CS name with a BF-FIB prefix, and **Aggregation** aggregates multiple BF-FIB prefixes up to a common shorter prefix in BF-FIB. While Transformation reduces BF-CS usage, it increases BF-FIB usage by the same amount. Aggregation on the other hand reduces BF-FIB usage. By repeatedly applying these two basic operations, BF-CS usage can be significantly reduced at the cost of small BF-FIB usage increment. For example, Figure 3 illustrates a CS with two entries `/A/a` and `/A/b`, initially using four names in BF-CS. Transformations on `/A/a` and `/A/b` remove two names from BF-CS, and add two prefixes to BF-FIB. Aggregation to `/A` replaces two BF-FIB prefixes with one shorter prefix in BF-FIB.

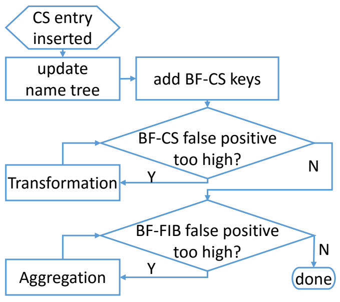
There is no false negative before or after any of these operation, because all CS entry names and their prefixes are matching at least one Bloom filter. However, as shown in Figure 3c and 3d, Interests `/A/b/3` and `/A/c` would match the newly added BF-FIB prefixes `/A/b` and `/A`, and therefore the NIC would deliver them to NFD, but NFD would drop them because they do not match any CS entry. The reason is that a BF-FIB prefix (using prefix matching) is less accurate than BF-CS names (using exact matching). Therefore, while Active CS can reduce BF false positives, it introduces a new type of false positive, **prefix match false positive**, which is a semantic false positive caused by broad coverage of BF-FIB prefixes.

The goal of Active CS is to minimize overall false positives of both types. While we can estimate BF false positive probability with Equation 1, prefix match false positives are dependent on traffic and hard to predict. Active CS works around this problem with three strategies that answer three questions:

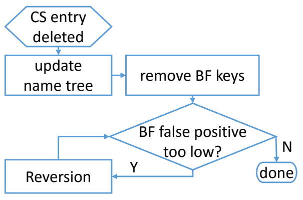
- When to do operations? It imposes an upper bound threshold on BF-CS/BF-FIB false positives, keeps as many names in BF-CS/BF-FIB as allowed, and doesn't do Transformation or Aggregation until BF false positives are too high.

- Where to do operations? Transformation and Aggregation are performed on the deepest eligible name tree node, to minimize the increase of prefix match false positives, under the assumption that a longer prefix in BF-FIB would match less irrelevant Interests. A name tree node is eligible for Transformation if its name is in BF-CS. A name tree node is eligible as a target of Aggregation (i.e. its name becomes the new prefix added to BF-FIB, while its descendants are removed from BF-CS/BF-FIB) if it has two or more descendants whose names are in BF-FIB.

- When to undo operations? When BF-CS/BF-FIB false positive probability is below a lower bound, we undo the two operations in an effort to reduce prefix match false positives.



(a) CS entry insertion



(b) CS entry deletion

Figure 4: Active CS flowchart

The workflow of Active CS is shown in Figure 4. When a CS entry is inserted, the name tree is updated accordingly, and names not already matched by BF-CS and BF-FIB are added to BF-CS as necessary [4]. Then, we check false positive probabilities of both BF-CS and BF-FIB. If either BF's false positive probability is over the upper bound threshold, Transformation or Aggregation is applied repeatedly until both BF false positives fall under the upper bound.

When a CS entry is deleted, the name tree is updated accordingly; if a name tree node is deleted, its name is removed from BF-FIB and BF-CS. After that, we check BF false positive probabilities. If both BF's false positive probabilities are below the lower bound threshold, we can perform a **Reversion** to undo Transformation and Aggregation: it replaces a BF-FIB prefix with a BF-CS name, and adds BF-FIB prefixes for its children in the name tree. Reversion is performed on the shortest BF-FIB prefix, in anticipation to maximize the reduction of prefix match false positives. To prevent oscillation between Transformation/Aggregation and Reversion, the lower bound threshold should be configured to have sufficient distance from the upper bound.

[4]More precisely, we only update CBF-CS in this step and remember which counters are changed; we send BF updates to NIC hardware at the end, so that names added to CBF-CS in this step but removed later by Transformation/Aggregation do not incur BF update overhead.

Active CS is considerably more complex than DM and Basic CS. It needs an efficient algorithm design so that its computation overhead does not outweigh the savings on packet processing overhead. To quickly locate where to perform Transformation, Aggregation, or Reversion, Active CS maintains additional information in the name tree. For Transformation, every name tree node remembers the distance to the deepest eligible descendant, and a pointer to the child containing that descendant. For Aggregation, every node remembers similar distance and pointer fields to the deepest eligible descendant, and also keeps track of how many of its descendants are in BF-FIB to indicate whether a node itself is eligible (Figure 5 shows an example of fields maintained for Aggregation). For Reversion, every node maintains the distance to the shallowest descendant having a BF-FIB prefix, and a pointer to the child containing that descendant.

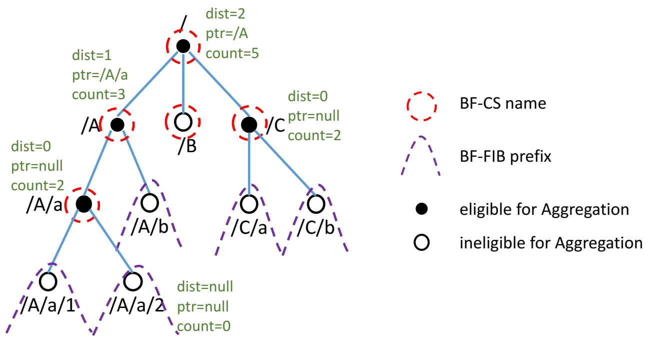


Figure 5: Name tree node fields for Aggregation
Fields for some nodes are omitted due to space constraint.

With those additional maintained information, Active CS can locate the desired node by walking down the name tree from the root following the pointers, which is an procedure of $O(h)$ time complexity, where h is the height of the name tree. To maintain these information, every time a CS entry is inserted or deleted, in the worst case, it's necessary to walk up the name tree and update all ancestors; at each node, the distance fields on all children of the node are checked to find the maximum (for Transformation/Aggregation) or minimum (for Reversion), and then the distance and pointer fields on the node itself can be calculated accordingly. The time complexity of these updates after a CS insertion/deletion is $O(kh)$, where k is the degree of a name tree node.

While h is relatively small because it's bounded by the number of name components in a name, k represents the fan-out in the name hierarchy which can potentially be large. To keep the CPU overhead low, we introduce **degree threshold**: if the degree of a name tree node is over a certain threshold, we add a BF-FIB prefix for this node; its descendants are then covered by the BF-FIB prefix, and will not trigger any updates in the distance fields and pointers. Afterwards, k is bounded by the degree threshold instead.

However, imposing a degree threshold may force a name prefix with high irrelevant Interest traffic to be added to BF-FIB, and cause significant increase of prefix match false positives. Therefore, it's a trade-off between name tree updating cost and prefix match false positives. We design and evaluate three types of degree threshold settings in Section 6.5.

6. EVALUATION

We evaluate NDN-NIC with different Bloom filter settings and update algorithms using an NFS trace collected from a department network, and observe its filtering accuracy, CPU usage, and BF update overhead on hardware. Simulation shows that, with 65536-bit BF-FIB & BF-CS, and 256-bit BF-PIT (16.03KB total size), NDN-NIC is able to filter out 96.30% of all received packets, and reduces CPU usage by 95.92% compared to a regular NIC. The above accuracy is achieved with a small hardware overhead of 6.72% extra clock cycles for BF updates, compared to clock cycles spent on processing outgoing packets.

6.1 Simulation Setup

NDN-NIC simulation is conducted in two stages. We first collect traffic and table trace from an emulated shared media, and then simulate NDN-NIC hardware and driver with a trace. In the first stage, we run Mininet [10] to emulate a shared media with a number of hosts. Each emulated host runs a modified version of NFD, which writes a *traffic and table trace* that logs every packet received from the shared media and whether it's accepted or dropped by NFD, and every change to the FIB, PIT, and CS tables. In the second stage, we process the traffic and table trace with an NDN-NIC simulator written in Python. Received packets are passed to the simulated hardware, which decides whether to deliver or drop it. This decision is then compared with NFD's decision: if the hardware delivers a packet but NFD drops it, a false positive is logged. Table changes are passed to the NDN-NIC driver, which may trigger Bloom filter updates. To quantify the overhead of such updates, we count how many bits are changed from 0 to 1 or from 1 to 0 during each table change.

We use realistic Network File System (NFS) traffic collected in a university department network in Nov-Dec 2014. We mirrored the traffic of NFS servers to a computer running nfsdump [6], which extracts NFS operations from network packets. The trace contains timestamp, NFS procedure code, client and server IP addresses, NFS filehandle, and file names (anonymized but preserving name hierarchy). Then we implement a pair of NDN applications to replay the trace over a NDN-based file access protocol that we have designed. This protocol generates two way traffic: to read a file or directory, an NFS client expresses Interests with the file name, and the NFS server replies with Data packets; to write a file, an NFS client sends a command as an Interest, and then the NFS server expresses Interests to retrieve the file from the client.

In Mininet, we emulate 2 NFS servers and 57 NFS clients, all on a single shared media, and replay 4 hours of NFS trace with these hosts. During the emulation, on each host, the FIB has only 1 entry, the PIT has up to 28 entries. The CS is configured to have a maximum capacity of 65536 entries; it is filled to capacity on one NFS server, has 50683 entries on the other NFS server, and has up to 10067 entries on NFS clients. 30416567 packet arrivals and 1443961 table changes are logged among 59 hosts; among those received packets, 758317 packets (2.49%) are accepted by NFD. Trace from each host is then simulated in the NDN-NIC simulator separately; the plots in this paper show the total across all hosts.

This simulation setup does not allow us to directly measure CPU usage. Since the workload is mostly memory bound, we estimate CPU usage in terms of memory accesses: we count how many name tree nodes are accessed during packet processing and name tree updating, and use that number to represent CPU usage.

6.2 Overall Filtering Accuracy

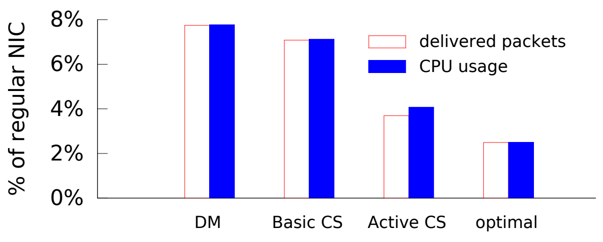


Figure 6: NDN-NIC overall filtering accuracy

Figure 6 shows the percentage of packets delivered by NDN-NIC to NFD with three update algorithms, as well as the estimated CPU usage, compared to regular NIC at 100%. BFs are set to reasonable target sizes for cheap NICs: 65536 bits each for BF-FIB and BF-CS, and 256 bits for BF-PIT; each BF uses two H3 hash functions with random polynomial terms. Active CS parameters are set to achieve the best result for the NFS trace. NDN-NIC filters out 92.26% of all packets with Direct Mapping (DM), 92.92% with Basic CS, and 96.30% with Active CS. NDN-NIC reduces CPU usage by 92.23% with DM, 92.83% with Basic CS, and 95.92% with Active CS. As a comparison, an optimal NIC would filter out 97.51% packets and reduce CPU usage by 97.49% CPU usage. No false negatives are found in these simulations.

6.3 Bloom Filter Characteristics

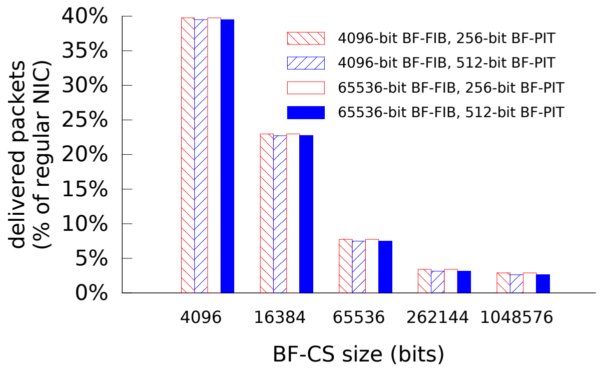


Figure 7: Effect of Bloom filter size, Direct Mapping

Bloom filter size directly affects filtering accuracy of NDN-NIC. Figure 7 shows the percentage of delivered packets with different BF sizes using Direct Mapping algorithms; each BF uses two random H3 hash functions. With Direct Mapping, having larger BFs improves filtering accuracy significantly, because there are less BF false positives with the same number of names added. Since CS is the largest among the three tables, increasing BF-CS size gives the most benefit. A similar trend is observed with Basic CS algorithm.

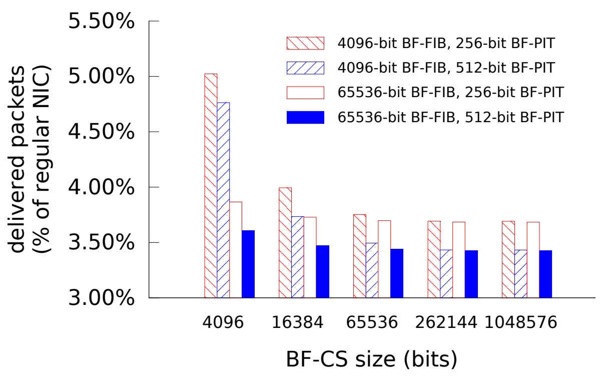


Figure 8: Effect of Bloom filter size, Active CS

Figure 8 shows the same metric with Active CS algorithm. BF false positive threshold is set to 0.1% upper bound, and degree threshold is set to "/64/64/32/16" (see Section 6.5). We can see a significant accuracy improvement when BF-CS size is increased from 4096 to 65536 bits, but the gain of increasing BF-CS size beyond 65536 bits becomes marginal because many name tree nodes have exceeded the degree threshold.

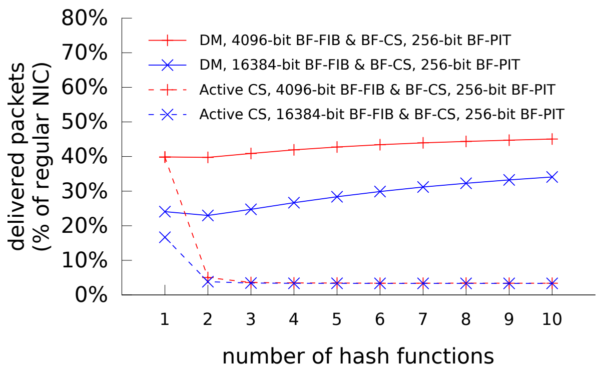


Figure 9: Effect of number of hash functions

The number of hash functions also affects BF false positive probability. Figure 9 show the percentage of delivered packets with two BF size settings, using different number of random H3 hash functions. We find that two hash functions is the best choice in our simulations. Having only one hash function degrades a BF into a hash table, and the filtering accuracy worsens significantly. Having more than two hash functions gives at most a marginal improvement on filtering accuracy, but requires more logic resources in hardware and more CPU time in software to compute these hashes. Although techniques such as double hashing [9] and segmented hashing [15] can keep the number of actual hash functions low but derive more hash values from the actual hash functions, the total entropy of those hash values is no more than the number of output bits from the actual hash functions; since each H3 hash function outputs just enough bits for indexing into the Bloom filter array (e.g. H3 outputs 16 bits for a 65536-bit BF), applying those techniques on H3 hash functions would create correlation among hash values and increase BF false positives.

6.4 Update Algorithms

In this section, we compare the CPU usage of regular NIC (no filtering), NDN-NIC with DM, Basic CS, and Active CS. The CPU usage includes packet processing and name tree updating cost.

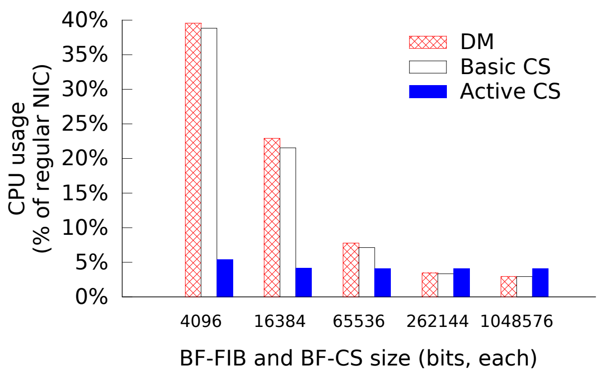


Figure 10: Comparison among update algorithms

Figure 10 shows the estimated CPU usage with different BF sizes. Basic CS performs only slightly better than DM, because in our traffic trace, few CS entries are covered by FIB entries registered by local producers. On the other hand, Active CS has much less CPU overhead than DM and Basic CS with smaller Bloom filters; the CPU usage of Active CS with 4096-bit BF-FIB/BF-CS is only slightly higher than that of Basic CS with 1048576-bit BFs. However, with larger BFs, Active CS has somewhat higher CPU usage than DM and Basic CS, because BF false positives are already low with little room for further reduction, and as a result name tree updating cost exceeds the potential saving on packet processing. Considering the limited memory resources on NIC, Active CS is better than the others.

6.5 Active CS Parameters

Active CS algorithm has two parameters, BF false positive thresholds and degree threshold. BF false positive thresholds control the BF false positive probability that can be tolerated in exchange for decreased prefix match false positives. Degree threshold limits the overhead of updating name tree nodes, but it may increase prefix match false positives. In this evaluation, we use 65536-bit BF-FIB/BF-CS, 256-bit BF-PIT, and two H3 random hash functions in each Bloom filter.

Figure 11 shows the CPU usage under different upper bound thresholds of BF false positives; degree threshold and Reversion operation are disabled in this test. We observe two trends: (1) Name tree updating cost increases with higher thresholds, because more names are kept in BF-CS, resulting in more nodes to update. (2) Packet processing cost is minimized at 0.05% threshold. Lower thresholds cause more CS entry names to be aggregated onto shorter BF-FIB prefixes which increases prefix match false positives; higher thresholds suffer from more BF false positives. The total CPU usage is a sum of the two costs; in our experiment, it's minimized at 0.01% threshold.

The effect of degree threshold is shown in Figure 12. We have tried three types of degree threshold settings: (1) "fixed" settings use the same degree threshold for all names; (2) "name-

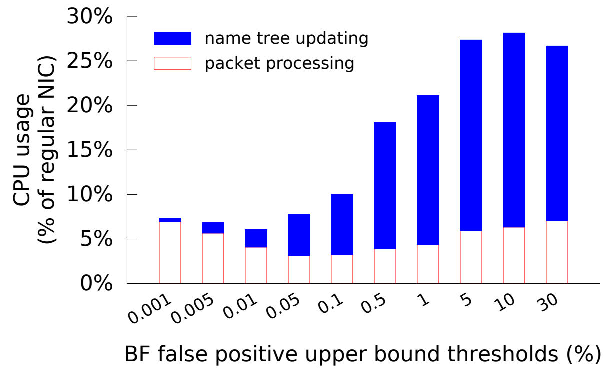


Figure 11: Effect of BF false positive thresholds

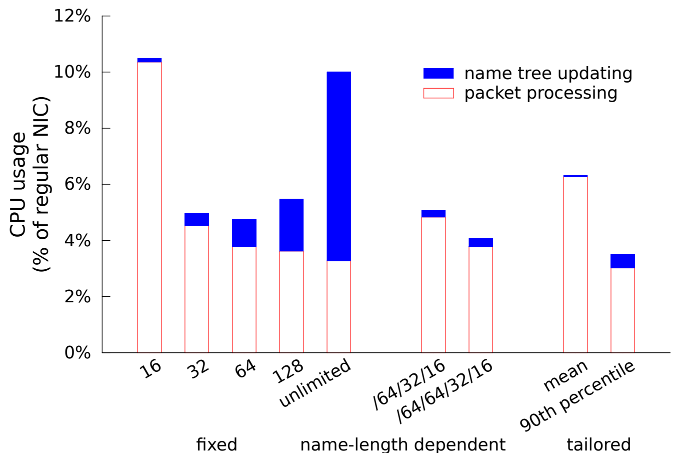


Figure 12: Effect of degree threshold

length dependent" settings provide different degree thresholds at different name lengths; (3) "tailored" settings derive degree thresholds from the traffic. In this test, BF false positive thresholds are set to 0.1% upper bound and an appropriate lower bound that prevents oscillation [5].

Among fixed settings, higher degree threshold incurs lower packet processing cost, because less BF-FIB prefixes are added due to exceeding degree threshold and thus less irrelevant Interests are accepted. On the other hand, it increases name tree updating cost, since there are more nodes needing updates.

With name-length dependent settings, shorter prefixes are given a larger degree threshold, because adding a short prefix to BF-FIB causes more prefix match false positives than longer prefixes. For instance, "/64/32/16" assigns degree threshold "64" to the first name component, "32" to the second name component, and "16" to all other components. As shown in the plot, "/64/32/16" and "/64/64/32/16" have similar packet processing cost compared to "32" and "64" respectively, but their name tree updating costs are lower.

The ideal degree threshold setting depends on traffic. Since NDN-NIC driver is a software component which can be easily reconfigured, in a real deployment, it's possible to adapt the degree threshold according to traffic patterns used by the applications. We count the degree of name tree nodes at different name lengths, and then tune the degree thresholds accordingly. Degree thresholds tailored to the 90th percentile of name tree node degrees at each name length performs the best among all tested settings.

[5]The lower bound is set to be low enough so that at least two Reversions can be performed from a BF at upper bound before its false positive probability reaches the lower bound.

6.6 BF Update Overhead on Hardware

In this section we quantify the impact of Bloom filter updates on hardware packet processing. We anticipate that Bloom filter updates are implemented as hardware logic (rather than through control plane). A regular NIC can process one byte of outgoing traffic per clock cycle [1]; BF updates share these clock cycles and compete with outgoing traffic [6], thus reduce outgoing bandwidth.

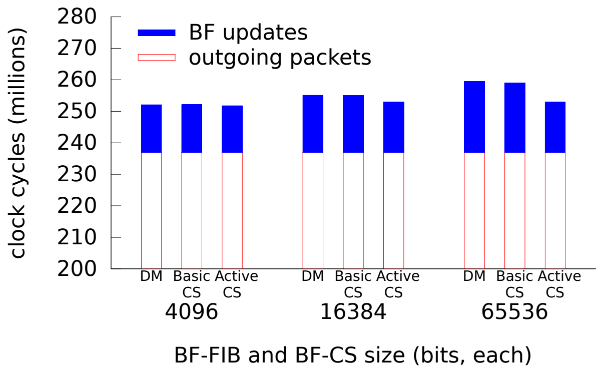


Figure 13: BF update overhead

In our NFS trace, NICs send a total of 318358 Interests and 206066 Data packets; using 100-byte average Interest size and 1000-byte average Data size, we can calculate that total size of outgoing packets is 237.9MB and they consume a total of 237.9 million clock cycles. Then we estimate the extra clock cycles consumed by BF updates. We assume each BF update needs 8 bytes (and thus 8 clock cycles), including 2 bytes to differentiate a BF update command from an outgoing packet and select which BF to update, 2 bytes as an address within the BF, and 4 bytes as the new value of the memory word [7]. We further assume the worst case that every bit updated during a table change is in a different memory word. Figure 13 shows the clock cycles spent on outgoing packets and BF updates with different BF-FIB/BF-CS sizes and update algorithms; BF-PIT has 256 bits.

We can see that BF updates consume between 14.8 million and 22.5 million clock cycles, depending on BF sizes and the update algorithm. This represents 6.21% to 9.45% overhead above the clock cycles spent on outgoing packets. Larger BF sizes tend to require more updates, because there are more bits in BFs so that when a name is added, the bit positions chosen by hash functions are less likely to be "1" already. Active CS incurs less BF update overhead than DM and Basic CS, because many CS entry insertions and deletions

[6]Incoming traffic is processed in a separate logical unit and does not compete with outgoing traffic.

[7]A faster implementation is possible, but we assume 8-clock-cycle BF updates to show an upper bound of BF update overhead.

occur under BF-FIB prefixes and do not require updates to BF-CS.

7. RELATED WORK

Bloom filters [3] (BFs) are building blocks widely used in IP networking, such as peer-to-peer network applications, resource/packet routing and measurements [4]. Fan et al. [7] exploited a hybrid usage of CBF and BF, applying it to web cache sharing, i.e., each Web cache tracks its own cache contents with a CBF, and broadcasts the corresponding standard BF to other caches, dramatically reducing both local updating cost and message traffic. Inspired by their approach, we maintain CBFs in NDN-NIC driver but download standard BFs into NDN-NIC hardware.

Dharmapurikar et al. [5] are the first to use BF to implement longest prefix match in IP forwarding. In NDN, longest prefix match is also needed for variable-length names in the FIB. So et al. [17] proposed to use hash tables combined with BF, and data pre-fetching for implementing the FIB. Wang et al. [19] proposed NameFilter, a two-stage BF-based scheme for fast longest name lookup in FIB. Quan et al. [13] proposed Adaptive Prefix Bloom filter, a scalable name lookup engine making hybrid use of BF and trie. Perino et al. [18] proposed a prefix Bloom filter scheme using GPU for FIB. Beyond FIB, Li et al. [11] proposed an enhanced implementation of PIT using mapping bloom filter, reducing on-chip memory consumption. While most work focuses on NDN name lookup in software, our work uses Bloom filters on hardware to achieve name-based filtering.

8. CONCLUSION

This paper proposes NDN-NIC, a network interface card for operating NDN on a shared media. NDN-NIC filters incoming packets on hardware by querying Bloom filters maintained by software, and drops irrelevant packets before they are delivered to NFD for software processing. This reduces CPU overhead and energy consumption. Although we did not build the NIC hardware, this paper tackles NDN-NIC's main research challenge: using the limited amount of on-chip memory to support packet filtering based on a large number of rulesets. Our simulations have shown that NDN-NIC can filter out most irrelevant packets, and achieve significant savings in CPU usage.

In the future, we plan to extend NDN-NIC design to deal with fragmented packets [16] and Nack packets [20], study various benchmarks, and implement NDN-NIC on a real hardware device to measure its actual performance.

9. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1345142 and 1455794, the Huawei Innovation Research Program, and NSFC 61373143 and 61432009. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

10. REFERENCES

[1] *NetFPGA-1G-CML Reference Manual*, 2016.

[2] A. Afanasyev et al. NFD Developer's Guide. Technical Report NDN-0021, Revision 6, Mar 2016.

[3] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7), July 1970.

[4] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4), 2004.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *SIGCOMM 2003*.

[6] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *LISA 2003*.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *Transactions on Networking*, 8(3), 2000.

[8] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking Named Content. In *ACM CoNEXT*, 2009.

[9] A. Kirsch and M. Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Struct. Algorithms*, 33(2), Sept. 2008.

[10] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *HotNets-IX*.

[11] Z. Li, K. Liu, Y. Zhao, and Y. Ma. MaPIT: An Enhanced Pending Interest Table for NDN With Mapping Bloom Filter. *Communications Letters, IEEE*, 18(11):1915–1918, 2014.

[12] NDN Project Team. NDN Packet Format Specification (version 0.2-alpha-3), 2015.

[13] W. Quan, C. Xu, J. Guan, H. Zhang, and L. A. Grieco. Scalable name lookup with adaptive prefix bloom filter for named data networking. *Communications Letters, IEEE*, 18(1), 2014.

[14] M. Ramakrishna, E. Fu, and E. Bahcekapili. A Performance Study of Hashing Functions for Hardware Applications. In *Int. Conf. on Computing and Information*, 1994.

[15] C. E. Rothenberg, C. A. B. Macapuna, M. F. Magalhães, F. L. Verdi, and A. Wiesmaier. In-packet Bloom filters: Design and networking applications. *Computer Networks*, 55(6):1364 – 1378, 2011.

[16] J. Shi. NDNLPv2. https://redmine.named-data.net/projects/nfd/wiki/NDNLPv2.

[17] W. So, A. Narayanan, D. Oran, and Y. Wang. Toward fast NDN software forwarding lookup engine based on hash tables. In *ANCS 2012*.

[18] M. Varvello, D. Perino, and J. Esteban. Caesar: A content router for high speed forwarding. In *SIGCOMM ICN*, 2012.

[19] 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 *INFOCOM 2013*.

[20] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang. A Case for Stateful Forwarding Plane. *Comput. Commun.*, 36(7), Apr. 2013.

[21] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, kc claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. Named Data Networking. *SIGCOMM Comput. Commun. Rev.*, 44(3), July 2014.