Computer Communications 91–92 (2016) 109–119

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Hardware accelerator to speed up packet processing in NDN router

Weiwen Yu, Derek Pao*

Department of Electronic Engineering, City University of Hong Kong, Hong Kong

ARTICLE INFO

Article history:
Received 4 December 2015
Revised 17 June 2016
Accepted 18 June 2016
Available online 23 June 2016

Keywords:
Packet processing
Named data networking
Hardware lookup table

ABSTRACT

A hardware implementation of the *pending interest table* (PIT) for *named data networking* (NDN) is presented. One of the major challenges in this research is the per-packet update requirement in NDN packet processing. In general, the data structure of the lookup table is optimized in order to minimize the implementation cost and maximize the lookup performance. However, more computation steps are required to update the highly optimized data structure. Thus, the design of the hardware lookup table needs to tradeoff between the implementation cost, lookup performance and update cost. We employ an on-chip Bloom Filter and an off-chip linear-chained hash table in our design. The lookup operation for an interest/data packet and the associated update operation are integrated into one task. This can effectively reduce the overall processing time and the I/O communications with the software control unit. Our design also incorporates a name ID table (*nidT*) to store all distinct name IDs (*nid*) in the PIT. If the content name in an interest packet can be found in the *nidT*, then the router needs not look up the *forwarding information base* (FIB) to determine how to forward the interest packet. This can reduce the workload of the FIB significantly. For proof-of-concept, the proposed hardware architecture is implemented on a FPGA and the overall packet processing rate is about 56 to 60 million packets per second.

1. Introduction

The Internet is one of the greatest inventions of all time. It has enormous impacts on various aspects of human civilization, e.g. science and technology, business operations, social behavior, and government. The Internet is a host-based communication network. A host connected to the Internet is identified by an *Internet Protocol* (IP) address. In order to set up a communication channel, the end users must know the IP address of the other parties. Driven by advancements in both software and hardware technologies, the communication needs have gradually shifted from point-to-point message exchanges to content distributions, i.e. the dissemination of digital media such as music, picture, video, e-book, etc. The global demand for data is nearing 30 exabytes (30 billion gigabytes) per month in 2011. Also, it was estimated that about 500 exabytes of new on-line contents had been created in 2008 alone [1].

Under the current communication model of the Internet, a user has to find out where the contents are located in order to make a request to retrieve the required data. A user has to know "*where*" in addition to "*what*" he/she wants to retrieve. One can easily see a serious semantic gap between the existing host-based communication model and the content-centric communication needs.

Named Data Networking (NDN) [2,3] is a new network architecture proposed by the computer network community to better support the emerging communication needs. Packets under the NDN framework are identified by *names*, instead of the IP address/port number in the conventional *Internet Protocol*. Unlike the fixed-length IP address, a packet name can have arbitrary length. Packet forwarding in a NDN router involves more complex *name-lookup* operations.

Communications in NDN are initiated by receivers. A user wishing to retrieve a given content sends out an *interest* packet (a request) to a NDN router. The interest packet carries the name that identifies the desired data unit, e.g. a segment of a video file. When the NDN router receives an interest packet from an interface (called a *face* in the NDN terminology), the router will perform the following actions:

- The router will check its *cache store* (CS) to see if the requested data packet is already available. If a matching data packet can be found, then the data packet will be sent out along the face where the interest packet is received.
- If the requested data packet is not available in the CS, the router will search the *Pending Interest Table* (PIT) to see if the processing of the request for the same data packet is already in-progress. If a matching entry is found in the PIT, the given

* Corresponding author. Fax: +85234420562.
E-mail addresses: weiwenyu2-c@my.cityu.edu.hk (W. Yu), d.pao@cityu.edu.hk (D. Pao).

http://dx.doi.org/10.1016/j.comcom.2016.06.004
0140-3664/

face will be added to the list of faces associated with the PIT entry.
- If no matching entry is found in the PIT, a new entry for the packet name will be added to the PIT. The router will then look up the *Forwarding Information Base* (FIB) to determine the next-hop for the given *Interest* packet. The lookup operation in the FIB is a component-based *longest prefix match* (LPM).

A response to the interest packet (i.e. a *data* packet) will be generated when (i) the interest packet reaches the source of the requested data; or (ii) the requested data packet can be found in the CS of a NDN router on the path towards the data source. On receiving a data packet, the NDN router will perform the following actions:

- The router looks up its PIT. If a matching interest is found, the data packet will be sent along the list of faces associated with the pending interest, and the corresponding PIT entry will be removed. The data packet may be cached based on the cache management policy. As a result, the CS will also be updated.
- If the given name cannot be found in the PIT, the data packet is unsolicited and it will be discarded.

The functional requirements of packet processing in the NDN architecture have been defined, but the design and implementation of the NDN router is still a research problem pursued by the academia and industry. One of the major research issues is related to the implementation of the lookup tables to support NDN packet processing [4–18]. Most of the previous publications on the implementation of lookup tables in NDN router are software-based, and up to now there are only a couple of publications on hardware implementation. This development trend resembles the historical development of the IP address lookup problem. In the 1990s, researches on IP address lookup methods were mostly software-based [19–21]. Throughput of software-based methods is limited and cannot meet the throughput requirements of core router. Researches on IP address lookup methods gradually shifted to hardware-based approaches in the 2000s [22–26].

Packet processing in NDN involves 3 major lookup tables, namely the FIB, PIT and CS. The CS is an optional component. The network can function properly without the CS. However, its present may enhance the overall system performance. To the best of our knowledge, we have only seen 1 published paper that presents a hardware implementation of the FIB [18]. The method of [18] requires the router to have 2 switch fabrics. Another approach to speed up the FIB lookup is to exploit the massively parallel computation power of GPU [13–15]. Routers without the required special hardware (e.g. one more switch fabric or GPU) may only implement the FIB using conventional software technology. Previously published software implementations of the PIT and FIB can only achieve packet processing rate of a few million packets per second (MPPS). In this paper, we shall present a hardware accelerator that can improve the packet processing rate of the PIT up to 60 MPPS, and reduce the workload of FIB lookup significantly, e.g. by 10 times or more. Hence, a software-based FIB may offer sufficient throughput for the NDN router. Our method can be extended to include the CS table as well. A major issue that is being addressed in this study is the stateful processing requirement in NDN. The hardware accelerator provides a simple interface and aims to minimize the workload of the software. It functions like a TCAM (ternary content addressable memory) co-processor [27] in conventional TCP/IP router. The network processor submits lookup/update requests to the hardware accelerator, which carries out the request and returns the lookup results. The network processor will then carry out the required actions accordingly.

The organization of the remaining parts of this paper is as follows. Basic packet processing requirements in NDN is analyzed in Section 2. A brief review of related work is presented in Section 3. The architecture of the proposed hardware accelerator is presented in Section 4. Performance evaluations and simulations results are presented in Section 4.2. Section 5 is the conclusion and future work.

2. Packet processing challenges in NDN

A fundamental difference between the packet processing in NDN and conventional TCP/IP routers is that NDN requires stateful processing, whereas TCP/IP only requires stateless processing. In the conventional TCP/IP router, the processing (e.g. admission, queueing and forwarding) of an incoming packet is independent of the packet arrival history. The nominal update frequency to the IP routing table is about a few hundred times per second. Some recent hardware implementations of IP address lookup engine [26] and multi-field packet classifier [28] support packet processing rate at 300 plus MPPS. The update to lookup ratio in IP routing table is very low, e.g. a few updates per million lookup. The update to lookup ratio for packet classifier is even lower because classification rules are often defined manually by the network administrator. Hence, the designer can optimize the data structures of the hardware TCP/IP lookup tables to maximize the lookup performance. The system can afford to spend more time on each update operation. Typically, necessary modifications to the data structures for an update request are determined by the management software, and then corresponding memory-write commands are issued to the hardware to make the changes.

The processing of an incoming NDN packet depends on packets that have previously been received by the router. This has a great impact on the design of the hardware lookup tables. The stateful processing requirement of NDN leads to per-packet update to the PIT and the CS tables. For example, an insertion to the PIT is required for each interest packet, and a deletion from the PIT is required for each data packet. Suppose the most recently received data packets are saved in the cache store. Hence, a cache replacement is required for each data packet. This means that there will be 2 updates (1 insertion and 1 deletion) to the CS for each data packet. To cope with the per-packet update requirement, the hardware should be able to execute update and lookup operations to the PIT/CS tables with the same efficiency. As a result, the lookup table design techniques for TCP/IP packet processing may not be applied to NDN. In particular, update operation should not involve any precomputation by the software to determine how to modify the data structures.

Data and interest packets are named. A name is made up of multiple components represented in the type-length-value (TLV) format [29]. A content (e.g. a video) may be divided into multiple segments. Two segmenting conventions, namely the sequence-based segmentation and the byte-offset segmentation, have been proposed. For example, the name /com/youtube/funny-video-8/2/1234 contains 5 components, and it refers to a specific segment of a video called funny-video-8 published by youtube.com. The last two components specify the version number 2 and the segment number 1234 of the given video. Matching of names is component-based instead of character-based. For example, the name /abc/de does not match the name /ab/cde.

According to the NDN proposal a data packet satisfies an interest if the content name in the interest packet is a (component-based) prefix of the content name in the data packet. The objective of the NDN proposal is to allow more flexibility for traffic aggregation and discovery [30]. It is expected that vast majority of interest packets will carry full names. Exact match is used to look up an interest packet name in the PIT. To lookup a data packet name in the PIT, the router is required to find potential matching interest for every component-based prefix of the data name.

All-prefix match is much more time consuming than exact match. Yuan and Crowley [4,5] argue that network traffic aggregation is usually performed at edge router where the router may have large number of lower speed interfaces. On the other hand, core router has fewer interfaces but each interface has much higher line speed. Packet processing speed instead of traffic aggregation is the major concern in core router. They propose to use exact match for data and interest packets lookup in PIT/CS at core router. The Cisco research team [12] also opines that all-prefix match is not feasible at scale and may actually create more problems. In this study, we also adopt the exact match approach in the implementation of the PIT.

Assume the payload of data packet is 1500 bytes (the maximum transmission unit allowed in Ethernet), and the size of interest packet is between 128 and 256 bytes. The ratio of data packet to interest packet is about 1 to 1. Hence, the average packet size is about 900 to 1000 bytes. For a 10 Gbps interface the maximum packet rate is about 1.38 MPPS. The average round-trip delay of today's Internet is about 80 ms. Hence, the number of entries in the PIT for a 10 Gbps interface is about 55 K. For a medium scale router such as the Cisco ASR 9000 with 36×10 Gbps interfaces, the expected overall packet arrival rate is about 45 MPPS, and the expected number of entries in the PIT is about 2 M. The size of the PIT may be overestimated based on the aforementioned simple analysis. Carofiglio et al. [31] develop an analytical model of PIT dynamics. Based on their model the size of the PIT is relatively small in the average case, e.g. 10 K to 200 K. In the worst case, the size of the PIT is up to 20 K entries for edge router, and up to 2 M entries for core router. Discussion on the size of the FIB will be given in Section 3.1.

The PIT/CS tables are defined to have global context with respect to the NDN router, i.e. the two tables are supposed to store information regarding the packet arrival history for all interfaces of the router. If a centralized lookup table is employed, its throughput should be equal to the sum of the packet arrival rates of all interfaces. It is very difficult, if not impossible, for software-based implementation to meet the required throughput of a centralized PIT. An alternative design is to employ a distributed lookup table, where a table partition is allocated in each interface. Tradeoff between the centralized versus distributed implementations of the PIT will be discussed in the next section.

3. Related work

In this section we shall give an overview of related work on the design and implementation of the PIT. Other studies on the implementations of CS, FIB, and NDN protocols will be given in Section 3.1. A major issue in the implementation of PIT is the placement of the PIT in the NDN router. Two different approaches have been proposed, i.e. a centralized PIT serving all interfaces, or a distributed PIT located in each interface. If a centralize PIT is used, the required processing rate should be sufficient to support the sum of the data rates of all interfaces. The major difficulty for hardware implementation of PIT is the per-packet update requirement. To the best of our knowledge, we have not seen any published hardware implementation of the PIT. Previous published methods are software-based. However, a pure software implementation is unlikely to be able to meet the throughput requirement of a centralized PIT for medium to large scale routers. Some software implementations based on conventional *d*-left and linear-chained hash tables [5,12] can only achieve packet processing rate of a few MPPS.

An alternative approach is to employ a distributed PIT. If a distributed PIT is used, there can be 3 possible placements of the PIT, namely in the ingress [6,7], egress [8], or third-part [9] interfaces. The drawback of placing the PIT in the ingress interface is that when a data packet is received, the header of the data packet needs to be broadcasted to all other interfaces to perform the PIT lookup. Thus, the work load of each PIT partition is equal to the sum of the data packet arrival rates on all other interfaces. Hence, the workload of each PIT partition is about 50% of a centralized PIT.

If the PIT is placed in the egress interface, then the FIB has to be looked up before the PIT. In general the FIB lookup time is longer than the PIT lookup time if the FIB is implemented in software. As a result, the FIB will likely become the system bottleneck. Placing the PIT in third-party interface will require another switch fabric to switch the packet headers and lookup results among the interfaces, which represents a very substantial hardware overhead. Moreover, if the FIB is also implemented using the distributed approach [18], then there may be contention for the switch fabric in switching the requests/results for PIT and FIB lookups.

Bloom filter (BF) [32] is a popular approach used in the implementation of NDN lookup tables. BF is a memory efficient data structure for checking if an input key is a member of a data set. However, the BF can only provide a binary yes/no reply, and it is not able to return the identity of the matching key. Moreover, the BF may generate false positive. To support delete operation, the bit-vector is replaced by an array of counters. This variant of BF is known as the counting Bloom filter (CBF). An implementation of a distributed PIT using BF called DiPIT is presented in [6,7]. The PIT partitions are located at the ingress interfaces. When a data packet is received, the packet header is broadcasted to all other interfaces for looking up the PIT. If there is a hit in the BF of interface *i*, the data packet is assumed to be solicited by an interest packet previously received at interface *i*. The data packet will then be forwarded to interface *i*. The DiPIT method is implemented in software [6]. The overall packet processing rate is limited, in particular each PIT partition is required to perform the lookup for data packets received from all other interfaces. In principle this approach can be implemented in hardware. However, there can be two potential drawbacks. First, to keep the false positive rate at a reasonably low level, a larger number of hash functions (e.g. 5–7) are necessary. If *k* hash functions are used, an update operation requires 2 k memory accesses (*k* memory reads to obtain the current count values, and *k* memory writes to update the count values). Second, the size of the counter in the CBF is restricted in hardware implementation, e.g. 3-bit counters are used in [6]. It is highly likely that the 3-bit counter may overflow. When counter overflow occurs, the CBF fails and subsequent update operations cannot be performed correctly. On the other hand, if counters with more bits are used, the CBF may not be stored on-chip.

Another design of PIT using BF called MaPIT is presented in [10]. BF does not identify the matching key. The MaPIT method resolves this problem by introducing an additional index called the *mapping array* (MA). The BF bit-vector is divided into *n* segments, and the MA is an *n*-bit vector. When a hash function map the input key to segment *i* of the BF bit-vector, the *i*th bit of MA is set to 1. The value of MA is used as the address to access the packet store to retrieve the associated control information of the matching key. The size of the BF bit-vector is set to 2^{24}, and *n* is equal to 30. Twelve hash functions are used. The BF bit-vector and the MA are supposed to be stored in on-chip memory, and the CBF (required to support insertion and deletion to the PIT) and the packet store are stored in off-chip memory. The authors of [10] evaluate their method on a conventional Intel i3 CPU. Their evaluations mainly concern about the on-chip memory usage, false positive rate and the setup time of the data structures. Packet processing rate is not reported in their paper. We have four major concerns regarding the MaPIT method. First, the multi-level on-chip cache memory of the CPU is controlled by the memory management unit based on the built-in set-associative mapping and least recently used (LRU)

cache replacement policy. The programmer does not have direct control of the cache memory allocation. Hence, it is not possible to ensure that the full BF bit-vector is always stored in on-chip memory. Second, the packet store has 2^{30} entries (since the MA has 30 bits), while the number of PIT entries is expected to be in the order of million. Not all the packet store locations are addressable since only $k=12$ bits out of $n=30$ bits in the MA can be equal to 1. Hence, the utilization of the packet store is pretty low, e.g. less than 1%. Third, the size of a BF segment is 2^{19}. It is highly likely that multiple distinct keys can have the same MA value. The authors of [10] have not discussed how to resolve collisions in the packet store. Fourth, the software needs to make 24 accesses to the main memory in order to update the CBF. This will limit the overall packet processing rate.

Yuan and Crowley present a software implementation of the PIT using d-left hash table [5]. The PIT is composed of d hash tables (d is equal to 2–4). Each hash table has B buckets, and each bucket can hold $E=8$ entries. When a key is inserted to the PIT, all the d hash tables are probed, and the key is inserted into the hash table bucket with the least load. The system also has an overflow table to store the items that cannot fit into the hash tables. To reduce the memory requirements and also speed up the searching time, a key is only represented by a 16-bit fingerprint in the hash table. The overflow table may help to resolve bucket overflow, but it will slow down the searching time. Another disadvantage of this approach is that the probability of having duplicated fingerprints (corresponding to distinct keys) mapped to the same hash bucket is not negligible. Interest packets with the same name will not be aggregated in their design.

3.1. Related work on the design of FIB, CS and protocol issues

The research team from Alcatel-Lucent [33] comments that there are about 1 trillion webpages as for January 2001, and these webpages are mapped to about 280 million globally unique and routable hostnames. Hence, they opine that the size of the FIB is potentially in the order of 100 million. They also present a distributed implementation of the FIB in hardware [18]. The FIB is divided into multiple disjoint partitions, and one partition is allocated on to each line card. The implementation of the lookup table is based on the DLB-BF method of [34]. The FIB lookup rate is up to 160 MPPS. The DLB-BF method requires an off-chip CBF maintained by the management software. Hence, this implementation technique does not support per-packet update, and it is not applicable to the implementation of PIT.

Some recent studies on name prefixes aggregation suggest that the size of the FIB can be reduced significantly. Afanasyev et al. propose a prefixes aggregation scheme based on the *map-n-encap* approach [35]. When a client wants to request information based on a provider independent name, it will first look up a mapping system (e.g. an extension of the domain name system) to find the corresponding *provider-aggregatable* address (i.e. the ISP prefix). The ISP prefix is stored as an additional field in the packet header, called the *forwarding alias*. The FIB in the NDN router will have two parts, the name prefixes for intra-provider routing and the forwarding alias for inter-provider routing. Major ISPs can have multiple tens of millions of subscribers. Hence, the number of intra-provider name prefixes in the FIB may be in the order of million. Adrichem and Kuipers [36] propose a similar name prefixes aggregation approach based on autonomous system rather than ISP.

The research team from Tsinghua University proposes a software implementation of the FIB [13–15]. They exploit the parallel processing power of GPU to speed up the lookup operation, and can achieve a lookup rate of about 60 MPPS for a name table with 10 M name prefixes. The packet processing rate is quite high. However, the GPU approach may not be suitable for the implementation of the PIT/CS table because of the per packet update requirement. Whenever the table is updated, changes to the data structures are to be determined by conventional CPU. The revised data structures are then uploaded to the GPU memory. The lookup operations are suspended during the update process.

Another research team from Beijing University of Posts and Telecommunications proposes to use a hybrid trie-based and Bloom-filter approach to perform FIB lookup [16,17]. The packet lookup rate is only up to 1 MPPS when implemented in software.

There have been some significant results on the design of the cache store in NDN router. High volume of network traffic passes through a core router per second. In order for the cache store to be effective, its size must be sufficiently big, e.g. in the order of 100 GB. The cost of implementing a cache store of that size in RAM is not affordable. Also the corresponding size of the CS table required by the NDN router for packet processing is not manageable. Rossi et al. [37,38] propose a hierarchical 2-level cache architecture. The L1 cache with 2–10 GB is implemented using SRAM/DRAM, and the L2 cache with larger capacity is implemented using solid state disk (SSD). L2 has sufficient capacity to hold half a million full video files. The system may pre-fetch data packets from L2 to L1 on-demand based on the recently received interest packets.

Conventional cache management policies include the least recently used, least frequently used, and the first-in-first-out policies. Thomas et al. [39] propose another cache management policy for NDN, called the *object-oriented caching*. If a video is popular, caching the video in NDN routers may improve the network performance. A video is divided into thousands of segments. Caching some random segments of the video in a router is not useful. It is highly unlikely that multiple clients are requesting the same segment within a time window of 0.1 s. Their caching policy requires that if a content is to be cached, the cache store will always hold the first k segments of the given content, where k is a system control parameter.

The aforementioned studies consider the PIT, CS or the FIB independently. Some recent studies attempt to implement a prototype NDN router. The Tsinghua research group presents a software implementation of an integrated FIB, CS and PIT using CBF called BFAST [11]. Suppose the content name has L components. Their method will iteratively search the lookup table with i components, where i starts from L and is decremented iteratively until a match is found. Their method running on an Intel Xeon E5645 CPU can attain 36.41 million lookup per second. For the insert and delete operations, they can only achieve 1.56 and 3.01 million updates per second, respectively. Because of the per packet update requirement of the PIT/CS, the overall packet processing rate is limited by the update rate.

Recently, the research team from Cisco presents a prototype design of a software NDN router [12]. Instead of using a simple linear search strategy as in [11], the Cisco team uses a 2-stage searching heuristic to reduce the lookup time of the FIB. The search starts from name prefixes with M components (where M is a design parameter), and either continues to shorter prefixes or restarts from a longer prefix if needed. Their design is based on the Cisco ASR 9000 software router with four 10 Gbps interfaces. Their implementation with 4 sets of (dispatcher + 2 packet processes) can achieve a packet processing rate of 4.5 MPPS.

Some other researches address the issues on forwarding strategies, flow control and communication protocols. The research group from UCLA and University of Arizona investigate the adaptive forwarding strategies [40]. The router maintains the status of each link, and uses this information to make decision on how to forward or retransmit interest packets. They also propose to introduce the negative ACK if a router does not know how to forward an interest packet. A research group from Alcatel-Lucent investigates the flow control of NDN traffics [41]. They present a window-

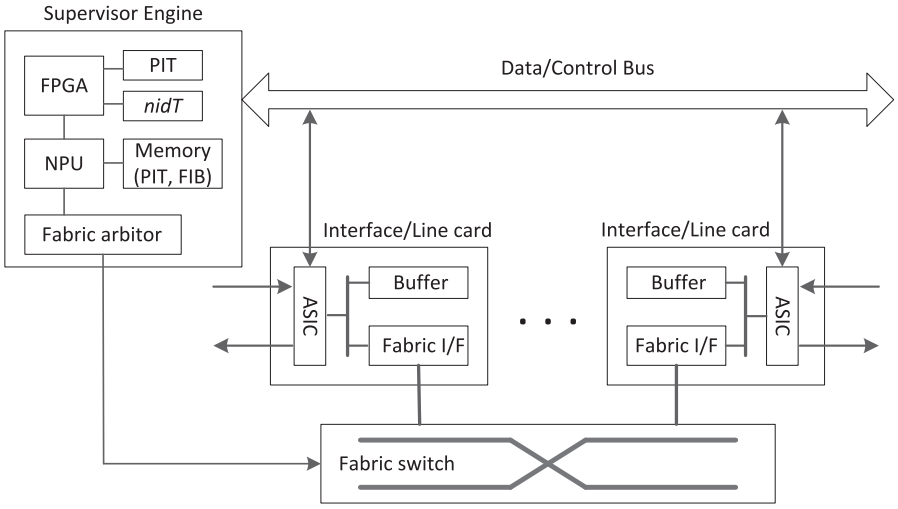

Fig. 1. Block diagram of the NDN router.

based flow control mechanism similar to the TCP flow control. They envision that a steady stream of interest and data packets of the same content will be passing through a NDN router. Aub et al. [42] study interest packet retransmission in lossy communication links and its impact on network performance.

4. Proposed method

We adopt a hardware and software co-design approach. The role of the hardware accelerator is to relieve the workload of the software on PIT lookup and management. The software control unit is mainly responsible for coordinating the operations of the switch fabric, FIB, cache store (if present), and the high level protocols. In most of the times, the lookup operation of an interest packet is followed by an insert operation, and the lookup operation of a data packet is followed by a delete operation. In our design, the hardware will integrate the lookup and the associated update operation into one command. By doing so, the processing time as well as the I/O communications between the software and hardware can be reduced.

The block diagram of the NDN router with the proposed hardware accelerator is depicted in Fig. 1. Packet headers are collected by individual interfaces and sent to the network processor (NPU) via a dedicated bus. The software control unit running on the NPU will then dispatch the lookup/update requests to the hardware accelerator implemented on a FPGA.

In this study, we divide the packet name into two logical parts, the content name (including the version number if applicable), and the segment number (*sn*). A 32-bit *sn* is assumed. Thanks to the TLV representation, the segment number can be easily extracted from the packet name. If the packet name does not contain a segment number, then our system assumes a default *sn* value of $2^{32}-1$ (32 bits of 1). The default *sn* value is to avoid potential confusion with other interest packet that is requesting the first segment of a file with $sn=0$. Lookup operation on the PIT is based on the ordered pair (content name, segment number). The content name can have variable length. It is much more expensive to perform matching of variable-length entities in hardware. To facilitate more efficient implementation of the PIT, the variable-length name is converted to a fixed-length 64-bit name ID (*nid*) using some standard hash functions, e.g. *CityHash* [43] or *SipHash* [44]. By doing so, the lookup/update operation on the PIT is much simplified. If k unique names are uniformly mapped to a 64-bit number space, the collision probability is approximately equal to $1-e^{\frac{-k(k-1)}{2N}}$, where $N=2^{64}$. Suppose the expected number of unique names stored in the PIT of a NDN router is about 0.5 M. The probability of having 2 or more distinct names mapped to the same *nid* is about 7×10^{-9}. Together with the 32-bit segment number, the chance of having 2 distinct interest packets sharing an identical (*nid, sn*) pair in the PIT is negligible. Some previous studies use up to 32-bit hash values, e.g. [5, 12]. When the size of the data set is in the order of million, collision is not avoidable if 32-bit hash values are used.

When the router receives an interest packet and has a PIT-miss, the router needs to find out how to forward the interest packet by looking up the FIB. In this study the FIB is assumed to be implemented in software. We shall take advantage of the locality of reference property in the traffic stream to reduce the number of FIB lookup. A window-based flow control mechanism for NDN has been presented in [41]. The user sends out a sequence of interest packets for a given content and waits for the arrival of the data packets. Receiving one or more data packets allows the user to send out new interest packets. The flow control mechanism regulates the stream of interest packets that pass through a NDN router. In the design of the hardware accelerator, we incorporate a *nid* table (*nidT*) to store all the distinct *nid* present in the PIT. If the router receives an interest packet where the *nid* is the same as some other interest packet currently stored in the PIT, then the router needs not look up the FIB to determine how to do the forwarding. Let L be the average number of interest packets of a flow that can be captured by the *nidT*, the number of FIB lookup can be reduced by a factor of L.

The hardware accelerator implements a mirrored and simplified copy of the PIT table and the *nidT*. The hardware performs the required lookup/update operations to the PIT and generates replies to the NPU. The NPU will then carry out the associated actions accordingly, e.g. forward or discard the packet, update the control information, look up the FIB if required, and etc. Possible extension to include the CS table will be discussed in Section 5. We shall first explain the interactions between the software control unit and the hardware accelerator. The implementation of the hardware lookup table will be discussed in Section 4.1.

The software and hardware maintain its own copy of the PIT. The hardware table contains only the essential data fields to support the search and update operations, while the software table contains other control information that are required for the handling of the data/interest packets. Data fields of the software and hardware tables are shown in Fig. 2. The hardware accelerator au-

Table		Data fields					
PIT	S/W	---	*nid*	*sn*	*nidT_addr*	face list	other control info
	H/W	*next*	*nid*	*sn*	---	---	---
nidT	S/W	---	*nid*	*count*	*nextHop*	*status* (valid, pending)	
	H/W	*next*	*nid*	---	---	---	---
CmdT	S/W	*cmdCode*	*nid*	*sn*	PIT_hit & PIT_addr	*nidT*_hit & *nidT*_addr	packet address

Fig. 2. Data fields of the software and hardware lookup tables.

Command	Parameter	Action & reply
LI (lookup interest)	*tag*, *nid*, *sn*	Action: Lookup and update PIT; lookup and update *nidT* if required PIT_hit: reply PIT_address PIT_miss: insert (*nid*, *sn*) into PIT, reply PIT_address lookup *nidT* with *nid* *nidT*_hit: reply *nidT*_address *nidT*_miss: insert (*nid*) into *nidT*, reply *nidT*_address Reply: *tag*, *cmdCode*, PIT_hit, PIT_address, *nidT*_hit, *nidT*_address
LD (lookup data)	*tag*, *nid*, *sn*	Action: Lookup and update PIT PIT_hit: reply PIT_address, delete the entry from PIT PIT_miss: reply miss Reply: *tag*, *cmdCode*, PIT_hit, PIT_address
CLR (clear entry)	*tag*, *nid*, *sn*	Action: Delete (*nid*, *sn*) from PIT and delete (*nid*) from *nidT* Reply (ACK): *tag*, *cmdCode*
IN (insert to *nidT*)	*tag*, *nid*	Action: Insert (*nid*) to *nidT* Reply: *tag*, *cmdCode*, *nidT*_address
RM (remove from *nidT*)	*tag*, *nid*	Action: Remove (*nid*) from *nidT* Reply (ACK): *tag*, *cmdCode*

Fig. 3. Lookup and update commands.

tomates the management of the PIT and *nidT*. It may take multiple clock cycles to carry out a command, and the software can issue a new command to the hardware before the results for the previously issued commands have been received. The system identifies each command/reply by a tag. The software maintains a pool of free tags, and the commands that are in progress are stored in the command table (*CmdT*). In this study, the size of the command table is set to 512, hence, a command can be identified by a 9-bit tag.

The list of commands that the software control unit may issue to the hardware accelerator is shown in Fig. 3. The packet processing procedure is as follows. When the software control unit receives a data packet, it issues a lookup data (LD) command to the hardware. The software control unit may then proceeds to do other tasks. The hardware looks up the given (*nid, sn*) pair in the hardware table. In case of a PIT-hit, the address at which the given (*nid, sn*) is found is returned to the software control unit. Moreover, the hardware will also delete the given entry from the hardware PIT automatically. When the software control unit receives the reply from the hardware, it carries out the appropriate actions. For PIT-hit the software can find out the detailed control information (e.g. the face list) from the software PIT at the given address, and forward the data packet to the corresponding output interfaces. The *count* value in the corresponding *nidT* entry is decremented by 1. The entry in the software PIT is released, and can be used to store other interest packet. In case of PIT-miss, the data packet is discarded. The software control unit is relieved from the burden of doing PIT lookup and management.

Similarly, when the software control unit receives an interest packet, it issues a lookup interest (LI) command to the hardware. The hardware will look up the given (*nid, sn*) pair in the hardware table. In case of PIT-hit, the address at which the given (*nid, sn*) is found is returned to the software control unit. In case of PIT-miss, the hardware will perform two related tasks automatically. First, it will insert the given (*nid, sn*) to the hardware PIT. Second, it will issue an internal command to look up (and insert) the given *nid* in *nidT*. The results returned to the software control unit include the PIT address and the *nidT* address, if applicable. When the software control unit receives the reply from the hardware, it will carry out the required action accordingly. In case of a PIT-hit, the interface at which the interest packet is received is added to the face list. In case of a PIT-miss, the given (*nid, sn*) pair will be stored in the software PIT at the address returned by the hardware. If *nidT*_hit is true and the *nidT* entry is valid, then the interest packet will be forwarded according to the *nextHop* stored in the software *nidT*. The *count* value of the corresponding *nidT* entry is incremented by 1. If the *status* of the *nidT* entry is pending (another software thread is in the process of looking up the FIB for the given *nid*), the software thread for the processing of the given interest packet is blocked and waits for the status to change to either valid or invalid. If *nidT*_hit is false, the software needs to look up the FIB and the status of the corresponding *nidT* entry is set to pending. When the FIB lookup result is available, the *status* and *nextHop* fields in the *nidT* entry are updated and other software threads that are waiting on the FIB lookup result are notified. If we have a FIB-hit, the interest packet is forwarded. If we have a FIB-miss, the router does not know how to forward the interest packet. The interest packet is dropped and the router may return a NACK to the upstream router. It will also need to remove the PIT entry and the *nidT* entry in the hardware table by issuing a CLR command to the hardware accelerator.

The commands RM (remove) and IN (insert) are used to manage the *nidT* table, if required. In case of a LD command with PIT-hit, the given entry (*nid, sn*) in the hardware PIT will be deleted automatically. The NPU can find out the associated *nid* entry in the *nidT* from the software PIT. When the entry (*nid, sn*) is deleted from PIT,

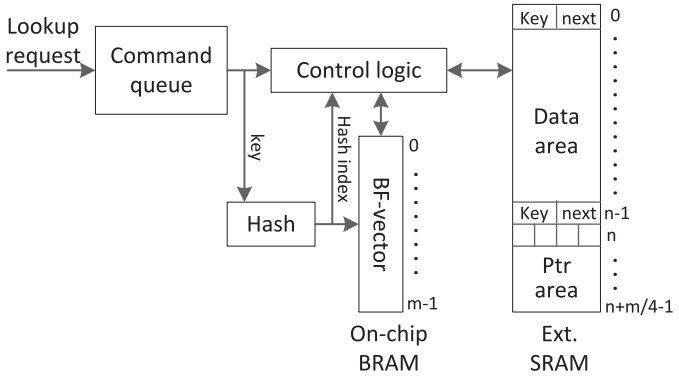

Fig. 4. Block diagram of the hardware lookup table.

the count value of the corresponding *nid* entry in *nidT* is reduced by 1. When the count value is reduced to zero, it means that there is no pending interest packet with the given *nid* in the PIT. The corresponding *nid* entry in the *nidT* is said to have expired, and may be removed. The expired *nid* is added to the pending removal queue. When the system needs to balance the size of the *nidT*, the software retrieves the *nid* to be removed from the pending removal queue, and submits a RM command to the hardware accelerator.

The IN command may be used in some rare occasion. The software is implemented using multi-threading, and the processing of each packet is done by a dedicated thread. It is possible that a LI command and a RM command with the same *nid* are submitted to the hardware by different software threads at more or less the same time. The software will have marked the *nidT* entry as invalid when a RM command is submitted to the hardware. Suppose the hardware completes the LI command before the RM command. Hence, the software will find out that the *nidT* address returned by the hardware is invalid. It will need to look up the FIB for the interest packet concerned, and then performs an explicit insertion to the hardware *nidT* table using the IN command.

4.1. Implementation of the hardware tables

The organization of the PIT and *nidT* are similar. There can be up to 8.5 MB of on-chip memory in state-of-the-art FPGA. Since the size of PIT is in the order of million, it is not possible to store the full lookup table in the on-chip block RAM (BRAM) of the FPGA. In our design, we make use of the limited on-chip BRAM to implement a BF for the lookup table, and store the full lookup table in external SRAM. The purpose of implementing the BF is to allow the system to determine in the shortest possible time that if the input key is not a member of the data set, and the system can take appropriate action accordingly. In our design, the system is not required to maintain a CBF in order to perform updates to the BF.

Block diagram of the hardware table is depicted in Fig. 4. The external SRAM array is divided into the data area and the pointer area. One memory word can hold a (*key, next*) pair in the data area, or 4 pointers in the pointer area. Up to n keys can be stored in the external table. A key (*nid, sn*) is mapped to a home bucket b, where $b = H(nid, sn)$, and H is the hash function. Implementation of the hash function is based on the family of H_3 hash functions of [45]. Suppose the key has i bits $= \{x_{i-1}, \ldots, x_1, x_0\}$. Let Q denote a $i \times j$ Boolean matrix, and q_k denote the kth row of Q. The hash function is defined as $H(key, Q) = x_0 \cdot q_0 \oplus x_1 \cdot q_1 \oplus \ldots \oplus x_{i-1} \cdot q_{i-1}$. The operator ' $\cdot$ ' represents the Boolean AND operator, and $\oplus$ represents the XOR (exclusive-OR) operator. The matrix Q is generated randomly off-line.

For the PIT table, the key contains 96 bits. It is rather expensive to perform the exclusive-OR operation of 96 bit-vectors. Hence, we shall first compress the 96 input bits (*nid, sn*) to 32 bits. The compression is done as follows. A random permutation of the 96 bits is pre-computed off-line, and the permutation is implemented as a hardwired circuit in the FPGA. Each block of 3 bits of the permuted key is fed to a 3-input XOR gate to produce 1 output bit. For the *nidT* table, the size of the key is 64 bits. We shall use the same approach to compress the 64-bit key to 32 bits.

Output of the hash function is within the range of 0 to $m-1$ ($m = 2^j$). Keys mapped to the same bucket are organized in a linked list. The pointer area is used to store the reference addresses of the linked lists (i.e. the physical address of the first key in each linked list). Vacant slots in the data area are chained up to form a free list, and the address of the first vacant slot in the free list is stored in an internal register, i.e. the *freeList* register.

In a LD command, the hardware will first check the BF-vector. If BF[b] = 0, the key is not present and the hardware will return a PIT-miss. If BF[b] = 1, the hardware will traverse the list of keys that are mapped to bucket b. The base address of the pointer area is stored in an internal register r. Each access to the external SRAM will read/write a 128-bit word. The reference address of the linked list for bucket b is the (b mod 4)-th pointer at location $r + b/4$. The hardware traverses the linked list concerned iteratively to search for the input key. If the input key is found at address x, the hardware returns a PIT-hit and will also delete the data item from the linked list. If the linked list becomes empty after the deletion, BF[b] is reset to 0. If the reference address of the linked list is modified (i.e. deleting the first node in the linked list), the hardware will also update the reference address of the linked list in the pointer area.

Fig. 5 shows the steps for the LD and LI commands for cases that require the minimum number of cycles. The minimum number of clock cycles for a LD command is 2 and 5 for PIT-miss and PIT-hit, respectively. The minimum cycle time for a PIT-miss corresponds to the case where the BF-vector bit is zero. The hardware uses the 1st cycle to compute the hash function. The on-chip BF-vector and the pointer area of the external SRAM are accessed in parallel in the 2nd cycle. Since the BF-vector bit is zero, no further processing is required. The minimum cycle time for a PIT-hit corresponds to the case where the key is stored in the 1st node of the linked list. After checking the BF-vector in cycle 2, the hardware read the stored key in the 3rd cycle and finds a match. In the 4th cycle the hardware updates the reference address of the given linked list and the on-chip BF-vector, if required. In the 5th cycle the hardware moves the storage slot to the free list.

In a LI command, if the key is found at address x, the hardware returns a PIT-hit and the value of x. If the key is not found, then the key will be inserted at the end of the linked list automatically. The first empty slot will be allocated to store the inserted key. After the insertion, the *freeList* register should be updated to reference the next free slot. Hence, the hardware is required to bring in the first slot of the free list in cycle 1 in order to find out the address of the next free slot. Suppose the key is inserted at address y. The hardware will record a PIT-miss and the address y in the reply record for the given tag. The reply record will not be returned immediately. The hardware will issue a lookup request (search-insert command) to the *nidT* to look for the given *nid*. The search-insert operation on the *nidT* is similar to that of the PIT. The results (*nidT*-hit or *hidT*-miss, and the associated address) will be stored in the reply record for the given command tag. The reply is then ready for return to the software control unit.

The insert and delete operations on the PIT and *nidT* are similar to that of the LD/LI commands, and we do not repeat the description here. From Fig. 5 we can see that in cycle T1 the hardware computes the hash function and accesses the data area of the ex-

Command	Outcome	Clock cycle				
		T1	T2	T3	T4	T5
LD	PIT-hit (1st node of linked list)	compute $H(nid, sn)$	check BF (BF =1), read pointer area	read data area & compare (match)	update pointer area, update BF if required	update data area (move slot to free list), update *freeList* reg
	PIT-miss (BF=0)	compute $H(nid, sn)$	check BF (BF = 0), read pointer area	---	---	---
LI	PIT-hit (1st node of linked list)	compute $H(nid, sn)$, read data area (get address of next free node)	check BF (BF = 1), read pointer area	read data area & compare (match)	---	---
	PIT-miss (BF=0)	compute $H(nid, sn)$, read data area (get address of next free node)	check BF (BF = 0), read pointer area	update pointer area, update BF, issue lookup request to *nidT*	update data area (insert item into PIT), update *freeList* reg	---

Fig. 5. Steps for the LD and LI commands requiring minimum number of cycles.

ternal memory array to retrieve the address of the second node in the free list. We shall introduce two refinements to the hardware implementation to reduce the cycle time of LD/LI command. The commands that operate on a table are lined up in the respective FIFO queue. The front item of the queue is available at the output interface of the hardware FIFO. We introduce one more register, the *currentCmd* register, in the hardware. By moving the current command to the *currentCmd* register we can have access to the next command in the queue. This allows the hardware to precompute the hash function of the next command while the current command is in progress.

For the LD command with PIT-hit, one access to the external memory (cycle T5) is required when the PIT entry is deleted and the slot is added to the free list. For the LI command with PIT-miss, when the input key is added to the table the *freeList* register needs to be updated. To do this, the hardware needs to perform a memory read in cycle T1 to obtain the address of the next free slot. In general, the LD and LI commands are interleaved. We can reduce the accesses to the external memory by introducing an internal buffer (FIFO) to store addresses of the PIT slots that have just been released. Suppose the size of the FIFO is 32 entries. When a PIT entry is deleted in a LD command, the address of the deleted entry is inserted into the FIFO if the queue is not full. If the queue is full, then the hardware will access the external memory and updates the *freeList* register as shown in cycle T5 of Fig. 5. In a LI command, if the FIFO is not empty then the hardware can get the address of a free slot from the FIFO. There is no need to access the external memory in cycle T1, and no need to update *freeList* register in cycle T4 when the input key is inserted to the PIT. By incorporating these 2 refinements, the time required for a LI command can be reduced by 1 clock cycle, and the time required for LD command can be reduced by up to 2 clock cycles.

For proof-of-concept, the proposed hardware architecture is implemented on a Xilinx Virtex-7 FPGA (model xc7v2000tflg1925-2). The size of the Bloom filter of the PIT is set to 2 M entries. The load factor of the Bloom filter is expected to be about 50%. The size of the data area of the external table is set to 1.25 M. The size of the Bloom filter of the *nidT* is set to 512 K, and the external table size is set to 320 K. The hardware contains two sets of PIT/nidT tables with a total capacity of up to 2.5 M PIT entries and 640 K distinct *nid*. When the hardware receives a command, it computes the parity of the rightmost 16 bits of the input *nid*. The parity bit is used to select one of the 2 sets of tables. The reply from the hardware will also include the parity bit. The external table can be implemented using the 133 MHz Cypress flow-through SRAM. The resource usage of the FPGA is summarized in Fig. 6. The FPGA can operate at 100 MHz after place and route. A higher processing rate is possible of the hardware is implemented with ASIC.

4.2. Performance evaluation and comparison

The packet processing rate of the hardware is studied via computer simulation. Without loss of generality the simulation program only simulates the performance of 1 set of PIT/nidT tables. The main purpose of the simulation is to find out the maximum packet processing rate of the hardware. The input command queue of the hardware is maintained to have at least 2 entries throughout the simulation. Unlike simulation studies that aim at evaluating the queueing discipline, statistical property of the packet arrival rate is not a major concern in our study. Generation of the interest packets is as follows. We assume there are 250 K distinct flows, and the lengths of the flows are generated using a Guassian random number generator. The mean μ varies from 4 to 20, and the standard deviation σ is equal to 0.25 μ. Sequence based segmenting is assumed. The starting value of the segment number of each flow is randomly generated within the range from 0 to 2^{30}. The 250 K flows are stored in the interest queue I_Q. To avoid generating interest packets of a flow having periodic arrival pattern, the queue is programmed to have 4 exits, namely the front of the queue and the three quartiles Q_1 to Q_3. The probability for selecting the flow at the front, and the 3 quartiles Q_1, Q_2, and Q_3 are equal to 60%, 20%, 10%, and 10%, respectively. An interest packet for the selected flow is generated, and the remaining flow length is decremented by 1. If the remaining flow length is greater than zero after the decrement, the flow is moved to the rear of the queue. When the selected flow is expired (i.e. its flow length becomes zero), it is removed from the queue. A new flow is generated and is inserted at the rear of the I_Q.

Generation of data packets is event driven, i.e. based on previously generated interest packets. Flows with pending data packets are maintained in another queue, the data queue D_Q. The management of the D_Q is similar to that of the I_Q. The simulator generates

Hardware Resource	Used	Available	Utilization
Slice	5676	305400	1.86%
Slice LUTs	13127	1221600	1.07%
LUT as Logic	9047	1221600	0.74%
LUT as Memory	4080	344800	1.81%
Slice Register	7511	2443200	0.31%
LUT Flip Flop Pairs	15791	1221600	1.29%
Block RAM (36 Kbit)	172	1292	13.31%
IOB	1182	1200	98.50%

Fig. 6. Summary of hardware resource utilization.

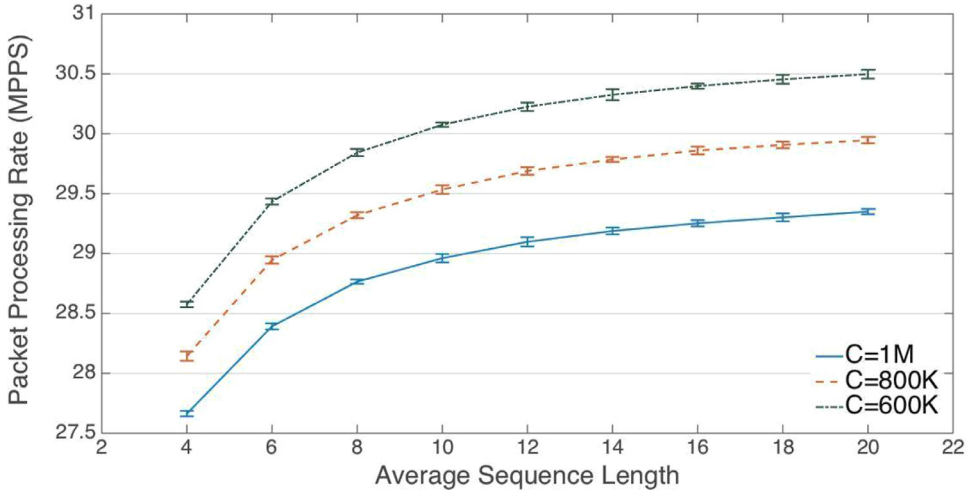

Fig. 7. Packet processing rate for 1 set of PIT/*nidT* tables.

C interest packets from the flows in I_Q in the initialization phase. After the initialization phase, the simulator will choose to select a flow from either the I_Q or the D_Q with equal probability. Hence, the parameter *C* represents the expected number of entries in the PIT. The simulator will then process a stream of *C* mixed interest and data packets before taking measurements. The throughput is measured for a period where 10 M packets are processed. Ten simulation runs are conducted for each set of simulation parameters.

In our simulation, all interest packets lookup will have PIT-miss, and all data packets lookup will have PIT-hit. This represents a somewhat worst case scenario for the hardware. When the last data packet of a flow is received, the corresponding *nid* in the *nidT* is also expired. The simulator will also simulate the software management functionality. The given *nid* is inserted into the pending removal queue. The size of the *nidT* is kept at 256 K. To balance the size of the *nidT*, the front element of the pending removal queue is retrieved and a RM command is issued to the hardware.

Fig. 7 shows the packet processing rate of the hardware for different average flow lengths and PIT populations. One can see that the packet processing rate is improved when the PIT population is lower. With lower PIT population (i.e. the Bloom filter has a smaller load factor), the average length of the linked lists in the external table is shorter and the throughput of the hardware can be improved. When the average flow length is increased, the number of RM commands is reduced. Hence, the workload of the hardware is also reduced. Fig. 8 shows the required FIB lookup rate for different average flow lengths. The FIB lookup rate can be reduced to less than 2 MPPS. The throughput of 1 set of PIT/*nidT* tables is about 28–30 MPPS. Two sets of PIT/*nidT* tables can be implemented on a FPGA. Hence, the overall packet processing rate of the hardware accelerator can be up to 60 MPPS.

To the best of our knowledge, we have not seen any publication on hardware implementation of the PIT. A comparison of the packet processing rate with some previously published software implementations of PIT is given in Fig. 9. DiPIT [6] and MaPIT [10] are not included in the comparison because packet processing rate of these 2 methods is not available in the original papers.

Only 1 hash function is used in the implementation of the hardware lookup table. The false positive rate of the Bloom filter can be reduced by using more hash functions. We have compared two different designs that use 1 hash function and 2 hash functions, respectively. It is found that if 2 hash functions are used, we have more flexibility in choosing to store a key in 1 of 2 possible linked lists, e.g. the linked list with shorter length. The lookup performance can be slightly improved. However, the update performance is degraded. More detailed evaluations via simulation indicate that the implementation using 1 hash function actually offers better overall performance.

5. Concluding remarks and future work

A hardware implementation of the PIT is presented. A major issue that needs to be resolved is the per-packet update requirement in the lookup table. The hardware will take care of the lookup and update operations autonomously. In our design, the lookup and the associated update operations are integrated. The software is relieved from the burden of doing the table lookup and update. For proof-of-concept, the proposed method is implemented on a FPGA, and the packet processing rate is about 56 to 60 MPPS. In our design, we incorporate a *nid* table (*nidT*) to store all the distinct *nid* present in the PIT. By doing so, if the router receives an interest packet where the *nid* is the same as some other interest packet currently stored in the PIT, then the router needs not look up the

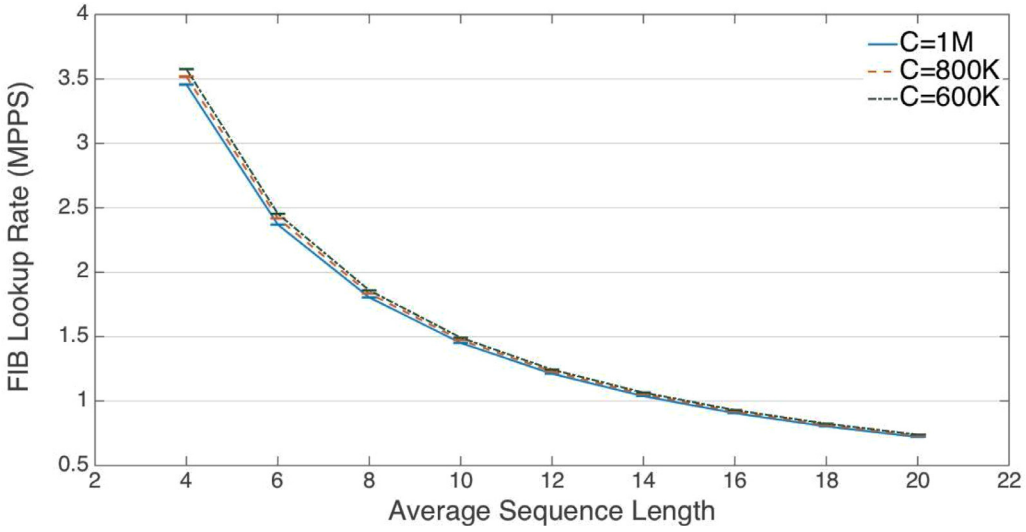

Fig. 8. FIB lookup rate for 1 set of PIT/*nidT* tables.

Method	Approach	Table size	Implementation platform	Packet processing rate
Yuan [5]	Fingerprint-based hash tables, with 16-bit fingerprints.	PIT: 64K to 2M	Intel Xeon E5540	Lookup/update latency: 1.2 μs (≈ 0.83 MPPS)
Dai [8] (NCE)	Name component encoding trie	PIT: 1.5M	Intel Xeon E5520	Lookup: 1.4 MPPS Insert: 0.9 MPPS Delete: 0.9 MPPS
Varvello [9]	Open-addressed d-left hash table.	PIT: 62K to 1M	Cavium Octeon Plus CN5650	3.4 MPPS overall
Dai [11] (BFAST)	Bloom filter with auxiliary CBF and unified index	Combined PIT, CS & FIB: 3M to 10M	Intel Xeon E5645	Lookup: 36.4 MPPS Insert: 1.56 MPPS Delete: 2.86 MPPS
So [12]	Linear chaining hash table and 2-stage FIB search strategy	PIT: 1M to 2M, FIB: 1M to 64M	Integrated Service Module, Cisco ASR 9000	4.5 MPPS overall
Proposed method	On-chip Bloom filter plus off-chip linear chained hash table	PIT: 2M	FPGA with ext. SRAM	56 to 60 MPPS overall

Fig. 9. Comparison of packet processing rate with other methods.

FIB to determine how to do the forwarding. Hence, a software implementation of the FIB may offer sufficient throughput. We shall, however, investigate possible hardware implementation of the FIB in our future work.

Fixed-length *nid*, instead of the variable-length content name, is commonly used by the research community in the implementation of the PIT and CS tables. Computation of the *nid* from the content name is performed repeatedly when the packet is forwarded from one router to the next router. Hence, it is desirable to standardize the hash function for the computation of the *nid*, and also include the *nid* in the packet header such that the computation workload of the NDN router can be largely reduced.

In our discussion we have only considered the PIT. It is possible to integrate the CS table with the proposed PIT with minor refinements. One more control bit is added to the software PIT/CS table to indicate if the entry corresponds to a pending interest or a cached data. When a data packet is cached, the table entry needs not be removed and the control flag in the PIT/CS entry is updated. Accordingly, one control bit is added to the hardware PIT/CS table. Suppose the most recent data packets are cached, then the LD command with PIT-hit will simply update the control bit instead of removing the entry from the PIT/CS. If a LD command finds a matching PIT/CS entry which corresponds to a cached data packet, then the hardware will generate a PIT-miss instead of a hit. As a result, the software will simply discard the data packet. An advantage of integrating PIT with CS is that the number of update operation for each data packet can be reduced from 3 to 1. If PIT and CS are implemented as two independent tables, the processing of a data packet may require 3 update operations (1 deletion from PIT, 1 deletion and 1 insertion to CS due to cache replacement). By integrating PIT with CS, the deletion from PIT and insertion to CS is effectively eliminated. The system only needs to update the information stored in the given table entry without modifying the data structure. One minor modification to the hardware implementation is required. One more command is required to carry out explicit deletion of the corresponding PIT/CS entry when a data packet is removed from the cache store. While the organization of the hardware table remains more or less unchanged, the size of the table needs to be adjusted according to the capacity of the cache store. We shall further investigate how to integrate our method with the 2-level hierarchical cache store architecture proposed in [37].

Adaptive forwarding strategies [40] can be incorporated with a simple refinement to the proposed method. In addition to the *nextHop* value, the address of the matching FIB entry can be stored in the software *nidT* table. Knowing the matching FIB entry, the router can carry out the adaptive forwarding strategies as presented in [40].

In this study, we use exact match instead of all-prefix match to look up data packet names in the PIT. How to support non-exact match and the associated per-packet update at high speed requires further research. The prefix match requirement also has great impacts on the management of the CS table. According to the NDN proposal, an interest name is matched against prefixes of the data names in the cache store. To facilitate fast CS table lookup using hashing, a data packet name with L components needs to be expanded to L discrete names with 1 to L components, respectively. The L discrete names for the given data packet are stored in the hash table. When a data packet is added to or removed from the cache, we need to perform L insert or delete operations to the CS table. The size of the CS table is increased, and the CS table will be overwhelmed by the large number of update operations. The vast majority of interest packets carry full names, and only a very small percentage of interest packets may carry partial names. Hence, it is worthwhile to investigate if name discovery can be resolved by the application layer such that NDN router needs not perform prefix match of names.

Acknowledgments

This work was supported by a grant from the Research Grant Council of the HKSAR, China (Project No. CityU 120513). The authors are grateful to the reviewers for their support and valuable comments.

References

[1] J.F. Gantz et al., The expanding digital universe: a forecast of worldwide information growth through 2010, IDC White Paper, 2007. http://www.emc.com/collateral/analyst-reports/expanding-digital-idc-white-paper.pdf.
[2] L. Zhang, et al., Named data networking (NDN) project, Technical Report NDN-0001, 2010. http://named-data.net/publications/techreports/.
[3] V. Jacobson, D.K. Smetters, J.D. Thornton, M. Plass, N. Brigs, R. Braynard, Networking named content, Commun. ACM 55 (1) (2012) 117–124.
[4] H. Yuan, T. Song, P. Crowley, Scalable NDN forwarding: concepts, issues and principles, IEEE ICCCN, 2012.
[5] H. Yuan, P. Crowley, Scalable pending interest table design: from principles to practice, in: IEEE INFOCOM, 2014, pp. 2049–2057.
[6] W. You, B. Mathieu, P. Truong, J.-F. Peltier, G. Simon, DiPIT: a distributed Bloom-filter based PIT table for CCN nodes, IEEE ICCCN, 2012.
[7] W. You, B. Mathieu, P. Truong, J.-F. Peltier, G. Simon, Realistic storage of pending requests in content-centric network routers, IEEE International Conference on Communications in China, 2012.
[8] H. Dai, B. Liu, Y. Chen, Y. Wang, On pending interest table in named data networking, in: IEEE/ACM ANCS, 2012, pp. 211–222.
[9] M. Varvello, D. Perino, L. Linguaglossa, On the design and implementation of a wire-speed pending interest table, in: IEEE INFOCOM WKSHPS, 2013, pp. 369–374.
[10] Z. Li, K. Liu, Y. Zhao, Y. Ma, MaPIT: an enhanced pending interest table for NDN with mapping Bloom filter, IEEE Commun. Lett. 18 (11) (2014) 1915–1918.
[11] H. Dai, J. Lu, Y. Wang, B. Liu, BFAST: unified and scalable index for NDN forwarding architecture, in: IEEE INFOCOM, 2015, pp. 2290–2298.
[12] W. So, A. Narayanan, D. Oran, Named data networking on a router: fast and DoS-resistant forwarding with hash tables, in: IEEE/ACM ANCS, 2013, pp. 215–225.
[13] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, Scalable name lookup in NDN using effective name component encoding, in: IEEE ICDCS, 2012, pp. 688–697.
[14] Y. Wang, H. Dai, T. Zhang, W. Meng, B. Liu, GPU-accelerated name lookup with component encoding, Comput. Netw. 57 (2013) 3165–3177.
[15] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, Wire speed name lookup: a GPU-based approach, USENIX Symposium on Networked Systems Design and Implementation, NSDI, 2013.
[16] W. Quan, C. Xu, J. Guan, H. Zhang, L.A. Grieco, Scalable name lookup with adaptive prefix Bloom filter for named data networking, IEEE Commun. Lett. 18 (2014) 102–105.
[17] W. Quan, C. Xu, A.V. Vasilakos, J. Guan, H. Zhang, L.A. Grieco, TB2F: tree-bitmap and Bloom-filter for a scalable and efficient name lookup in content-centric networking, IFIP Networking Conf., 2014.
[18] M. Varvello, D. Perino, J. Esteban, Caesar: a content router for high speed forwarding, in: ACM ICN, 2012, pp. 73–78.
[19] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, in: ACM SIGCOMM 1997, Cannes, France, 1997, pp. 3–14.
[20] S. Nilsson, G. Karlsson, IP-address lookup using LC-tries, IEEE J. Sel. Areas Commun. 17 (June, 6) (1999) 1083–1092.
[21] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Trans. Networking 7 (1999) 324–334.
[22] P. Gupta, S. Lin, N. McKeown, Routing lookups in hardware at memory access speeds, in: IEEE INFOCOM, 1998, pp. 1240–1247.
[23] D. Pao, C. Liu, A. Wu, L. Yeung, K.S. Chan, Efficient hardware architecture for fast IP address lookup, IEE Proc. Comput. Digit. Tech. 150 (Jan., 1) (2003) 43–52.
[24] V.C. Ravikumar, R.N. Mahapatra, TCAM architecture for IP lookup using prefix properties, IEEE Micro 24 (March-April, 2) (2004) 60–69.
[25] I. Sourdis, G. Stefanakis, R. de Smet, G.N. Gaydadjiev, Range tries for scalable address lookup, in: ACM/IEEE ANCS, 2009, pp. 143–152.
[26] D. Pao, Z. Lu, Y.H. Poon, IP address lookup using bit-shuffled trie, Comput. Commun. 47 (2014) 51–64.
[27] IDT Network search engine, http://www.idt.com.
[28] D. Pao, Z. Lu, A multi-pipeline architecture for high-speed packet classification, Comput. Commun. 54 (2014) 84–96.
[29] NDN Project Team, "NDN packet format specification", http://named-data.net/doc/ndn-tlv/.
[30] NDN Project Team, "NDN protocol design principles", http://named-data.net/project/ndn-design-principles/.
[31] G. Carofiglio, M. Gallo, L. Muscariello, D. Perino, Pending interest table sizing in named data networking, in: ACM ICN, 2015, pp. 49–58.
[32] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (1970) 422–426.
[33] D. Perino, M. Varvello, A reality check for content centric networking, in: ACM ICN, 2011, pp. 44–49.
[34] H. Song, F. Hao, M. Kodialam, T.V. Lakshman, IPv6 lookups using distributed and load balanced bloom filters for 100 Gbps core router line cards, in: IEEE INFOCOM, 2009, pp. 2518–2526.
[35] A. Afanasyev, C. Yi, L. Wang, B. Zhang, L. Zhang, "Scaling NDN Routing: Old Tale, New Design", Technical Report NDN-0004, 2013.
[36] N.L.M. van Adrichem, F.A. Kuipers, Globally accessible names in named data networking, IEEE INFOCOM WKSHPS, 2013.
[37] G. Rossini, D. Rossi, M. Garetto, E. Leonardi, Multi-terabyte and multi-Gbps information centric routers, IEEE INFOCOM, 2014.
[38] R.B. Mansilha, L. Saino, M.P. Barcellos, M. Gallo, E. Leonardi, D. Perino, D. Rossi, Hierarchical content stores in high-speed ICN routers: emulation and prototype implementation, in: ACM ICN, 2015, pp. 59–68.
[39] Y. Thomas, G. Xylomenos, C. Tsilopoulos, G.C. Polyzos, Object-oriented packet caching for ICN, in: ACM ICN, 2015, pp. 89–97.
[40] C. Yi, A. Afanasyev, L. Wang, B. Zhang, L. Zhang, Adaptive forwarding in named data networking, ACM SIGCOMM Comput. Commun. Rev. 42 (3) (2012) 62–67.
[41] G. Carofiglio, M. Gallo, L. Muscariello, ICP: design and evaluation of an interest control protocol for content-centric networking, in: IEEE INFOCOM, 2012, pp. 304–309.
[42] A.J. Abu, B. Bensaou, J.M. Wang, Interest packets retransmission in lossy CCN networks and its impact on network performance, in: ACM ICN, 2015, pp. 167–176.
[43] F. Pike, J. Alakuijala, CityHash: fast hash functions for strings, 2011. http://code.google.com/p/cityhash.
[44] J.-P. Aumasson, D.J. Bernstein, Siphash: a fast short-input PRF, Cryptology ePring Acrhive, Report 2012/351, 2012. http://eprint.iacr.org.
[45] M.V. Ramakrishna, E. Fu, E. Bahcekapili, Efficient hardware hashing functions for high performance computers, IEEE Trans. Comput. 46 (1991) 1378–1381.