Named Data Networking on a Router: Fast and DoS-resistant Forwarding with Hash Tables

Won So, Ashok Narayanan and David Oran
Cisco Systems, Inc. Boxborough, MA, USA
{woso, ashokn, oran}@cisco.com

ABSTRACT

Named data networking (NDN) is a new networking paradigm using named data instead of named hosts for communication. Implementation of scalable NDN packet forwarding remains a challenge because NDN requires fast variable-length hierarchical name-based lookup, per-packet data plane state update, and large-scale forwarding tables.

In this paper, we review various design options for a hash table-based NDN forwarding engine and propose a design that enables fast forwarding while achieving DoS (Denial-of-Service) resistance. Our forwarding engine features (1) name lookup via hash tables with fast collision-resistant hash computation, (2) an efficient FIB lookup algorithm that provides good average and bounded worst-case FIB lookup time, (3) PIT partitioning that enables linear multi-core speedup, and (4) an optimized data structure and software prefetching to maximize data cache utilization.

We have implemented an NDN data plane with a software forwarding engine on an Intel Xeon-based line card in the Cisco ASR 9000 router. By simulation with names extracted from the IR-Cache traces, we demonstrate that our forwarding engine achieves a promising performance of 8.8 MPPS and our NDN router can forward the NDN traffic at 20Gbps or higher.

Categories and Subject Descriptors

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

General Terms

Experimentation, Design, Performance

Keywords

Named data networking, packet forwarding engine, hash table

1. INTRODUCTION

Named data networking (NDN) is a networking paradigm that tries to address issues with the current Internet by using *named data* instead of named hosts for the communication model. NDN communication is consumer-driven; a client requests a named content via an *Interest* packet, then any node receiving the interest and having the content satisfies it by responding with a *Data* packet [13].

Since NDN challenges some fundamental technical tradeoffs of the current Internet, it redesigns many aspects including naming, security, forwarding, routing, and caching. Among those, implementation of scalable NDN packet forwarding remains a big challenge because NDN forwarding is fundamentally different, and inherently more complex than IP forwarding.

First, NDN forwarding lookup uses variable-length hierarchical tokenized names, rather than fixed-length addresses. The forwarding decision is made based on either an exact-match or componentized longest prefix match (LPM) of the name against a set of tables (discussed below). For example, consider an NDN Interest packet with the name `/com/cisco/ndn` that consists of 3 components delimited by `/`. This Interest can match on the FIB with `/com/cisco/ndn`, or a shorter prefix, `/com/cisco` or `/com`.

Second, NDN uses 3 different forwarding data structures: (1) the *Pending Interest Table (PIT)* is a table that stores unsatisfied Interests – an entry is added when a new Interest packet arrives and removed when it is satisfied by the corresponding Data packet, (2) the *Content Store (CS)* is a buffer/cache memory that stores previously processed Data packets in case they are re-requested, and (3) the *Forwarding Information Base (FIB)* fills the analogous role as with IP – it is used for forwarding Interest packets based on the longest prefix match. As shown in Figure 1, the NDN packet forwarding process requires consulting all three data structures at different points. These tables can be quite large; approximately $O(10^8)$ for the FIB, $O(10^7)$ for the PIT [28, 5] and the CS based on an assessment of the likely traffic matrix seen by a large-scale NDN router.

Lastly, unlike IP some NDN data plane data structures require per-packet state update. If we simplify the NDN forwarding process with regard to operations on the forwarding tables (refer [13] for details), Interest forwarding consists of CS lookup, PIT lookup & insert, and the longest prefix match on the FIB while Data forwarding is composed of CS lookup, PIT lookup & delete, and CS delete & insert (i.e. cache replacement). For processing every Interest or Data packet, the data plane must perform an insertion or deletion on the CS or PIT. This requires us to rethink the algorithms and data structures widely used for IP forwarding engines that only do read operations on a per-packet basis.

We note there are other unique properties of the NDN data plane such as long-term caching, implicit multicast, and a forwarding strategy. For this work, we focus only on the design issues regarding name lookup and forwarding tables.

In this paper, we present a design of a fast NDN data plane based on hash tables. We review various design options for an NDN forwarding engine including hash function choice, hash table design, lookup algorithms for FIB, PIT and CS, multi-core parallelization, and router security. We then describe our design that enables high-speed forwarding while achieving DoS (Denial-of-Service) resistance. It features (1) name lookup via hash tables with fast collision-resistant hash computation, (2) an efficient FIB lookup algorithm that provides good average and bounded worst-case FIB lookup time, (3) PIT partitioning that enables linear multi-
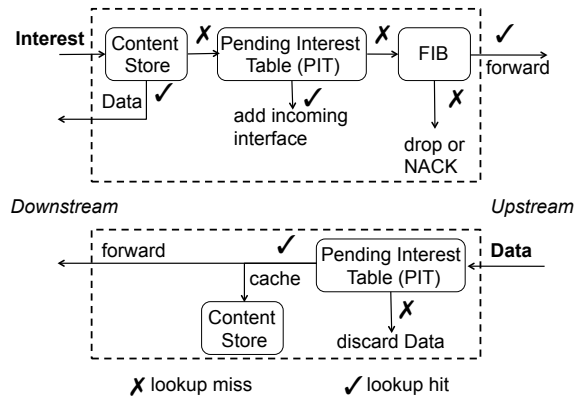
Figure 1: NDN Interest/Data forwarding process (Figure courtesy of B. Zhang [27]).

core speedup, and (4) an optimized data structure and software prefetching to maximize data cache utilization.

Based on the proposed design, we have implemented an NDN data plane entirely in software on an Intel Xeon-based line card in the Cisco ASR 9000 router. By simulation with names extracted from the IRCache traces, we demonstrate that our forwarding engine achieves a promising performance of 8.8 MPPS and our NDN router can forward NDN traffic at 20Gbps or higher. To the best of our knowledge, this is the first work that includes a comprehensive NDN data plane design based on hash tables with fast and DoS-resistant forwarding and implemented on a real commercial router.

The rest of this paper is organized as follows. We describe design details of our forwarding engine in Section 2. Section 3 provides an overview of our router-based system implementation. Our workload and experimental results are described in Section 4. Finally, we discuss related work and conclude in Sections 5 and 6.

2. FORWARDING ENGINE DESIGN

An efficient name lookup engine is a critical component of fast NDN packet forwarding. Since NDN lookup is based on variable-length names, it can be seen as a string match problem. After investigating 2 approaches – DFA (Deterministic finite automata) vs. hash table (HT), we chose HT because it is simpler and we believe better matched the nature of the NDN lookup problem. The difficulty of a DFA-based approach is that each prefix in NDN names is associated with an unbounded string, and hence it requires a special encoding scheme [24]. (See Section 5.1 for details.)

As illustrated in Figure 2, an HT-based lookup algorithm for NDN Interest forwarding works as follows: (1) after decoding the packet and (2) parsing the name components, (3) the full-name hash (H_n) is generated from the Interest name and matched against the CS, PIT then FIB HT, and (4) successive lookups are performed with shorter name prefix hashes ($H_{n-1},...,H_1$) against the FIB, which stores all prefixes in a large HT. Data packet forwarding only needs CS and PIT lookups with the full-name hash.

For the rest of this section, we review various design options for hash table-based lookup. These include selection of the hash function, HT data structures, and ways to improve lookup performance for FIB, CS and PIT. We focus on pure software forwarding partly so that we can establish a useful lower bound that any hardware assisted or pure hardware forwarding approach must surpass to be competitive. Further efforts based on custom hardware may lead to different design decisions.
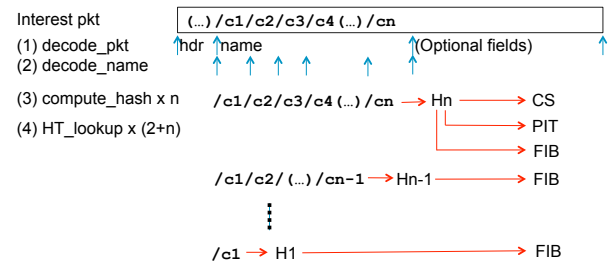
Figure 2: NDN Interest forwarding using hash tables.

2.1 Hash function

The choice of a hash function affects the performance of HT-based lookup in 2 ways: the computational load of a hash function itself, and the quality of generated hashes. We require an efficient hash function that computes hashes fast for character strings, yet has low hash collision probability, and short average HT chain length to minimize the hash bucket search time.

We have evaluated the performance and quality of various widely-used cryptographic and non-crpytographic hash functions for our purposes, building a hash table with 64K random strings of 10∼60 characters each. Table 1 summarizes the results; the upper 3 rows show the performance metrics while the lower 4 rows show the quality metrics. The performance of cryptographic hash functions (5 right columns) is noticeably worse despite some utilize the Intel's AES-NI instruction sets. Most of the quality metrics are similar though CCNx and CRC32 suffered hash collisions – 2 distinct random strings in the sample generated the same hash value. We should avoid these functions because even infrequent hash collisions negatively affect lookup performance by forcing extra string comparison during HT lookup. Among the rest, CityHash64 shows the best performance in terms of both byte hash and HT lookup performance.

Despite their better performance, non-cryptographic hash functions lack collision resistance, resulting in an NDN router becoming vulnerable to hash flooding DoS (Denial-of-Service) attacks. Aumasson et al. [2] have proved that seed-independent collisions can be generated with popular hashes such as MurmurHash, City-Hash and SpookyHash. This is particularly dangerous for an NDN router because a PIT entry is inserted based on the name of an Interest packet generated by a user. If a malicious user generates Interest packets with a set of names that cause seed-independent hash collisions, a router (or routers) will suffer significant lookup performance degradation due to the hash collision that happens at every PIT HT lookup. Note that using a different seed at each router does not alleviate this attack because the collision happens with seed-independence.

The recently proposed *SipHash* [2] offers a good balance as it provides collision resistance and comparable performance to non-cryptographic hashes. Our evaluation shows that SipHash is 23% slower than CityHash64 but 61% faster than MD5 with a string shorter than 60 bytes. Since collision resistance is a necessary property for the NDN router security, we have chosen SipHash for our HT-based lookup design. In order to effectively generate a set of hashes ($H_n,...,H_1$) for Interest forwarding, we have implemented our own SipHash that computes all name prefix hashes with one pass while simultaneously parsing the name components.
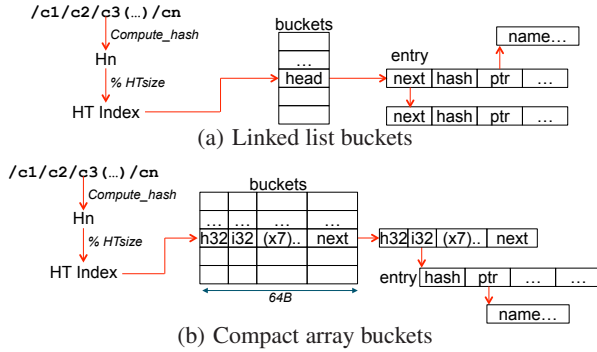
2.2 Hash table design

For an HT design, we chose a basic chained HT (a.k.a separate chaining, closed addressing) because it is simple and requires only

Table 1: Performance and quality of various hash functions.

Hash type	Non-cryptographic					Cryptographic				
Hash function	CCNx	CRC32	CityHash64	Spooky	Jenkins	MD5	LANE	ECHO	Grøstl	SipHash
Average cycles/byte hash	8.88	4.77	4.80	5.03	5.65	15.23	20.13	36.82	51.34	5.94
Incremental cycles/byte hash	4.93	1.08	0.69	0.77	3.29	7.67	5.41	3.18	23.67	3.53
Cycles/HT lookup*	436.73	323.23	296.66	380.12	389.28	906.53	1316.51	2237.78	2407.47	421.96
Avg. length of chain	1.60	1.58	1.57	1.58	1.58	1.58	1.57	1.58	1.58	1.58
Max. length of chain	9	7	7	8	8	8	7	8	9	7
Empty bucket %	38.18	37.17	36.98	37.25	37.16	37.34	37.03	37.25	37.22	36.69
Hash collision	164	1	0	0	0	0	0	0	0	0

* HT lookup tests are done with the chained HT with 64K hash buckets and 64K pre-populated entries (i.e. load factor = 1).

Source: CCNx [3], CRC32 [15], CItiHash64 [9], Spooky, Jenkins [14], MD5 [20], LANE, ECHO [7], Grøstl [10], SipHash [2].

(a) Linked list buckets

(b) Compact array buckets

Figure 3: Hash table data structures.

one hash function. We rejected designs that require multiple hash functions (e.g. Cuckoo hashing) because the extra computational overhead would be huge as we need a set of hashes $(H_n,...,H_1)$ for an NDN name. For faster lookup with a chained HT, our first design goal is to minimize string comparison that negatively affects the performance. For this, the 64-bit hash value generated from a name is stored at every hash table entry. While searching for a matching entry during HT chain walk, string comparison is performed only when it finds an entry that has the exactly same 64-bit hash value as the input lookup key. Use of a strong 64-bit hash function (SipHash) is well aligned with this design because the hash collision probability is extremely low; $(1/10^6)$ with 6.1 million values [26].

Our second goal is to design a data cache (D$) friendly data structure that causes fewer cache misses during HT lookup. On an Intel-based platform, a single D$ miss introduces more than a hundred memory stall cycles and therefore it is crucial to minimize the number of D$ misses. Instead of using linked list buckets, we have designed a D$-friendly HT data structure using a *compact array*, where each hash bucket contains 7 pre-allocated compact entries (32-bit hash and 32-bit index to the full entry) in 1 cache line (64 bytes). As shown in Figure 3, the key difference is that a single D$ miss occurs in every linked list chain walk while it happens only once in every 7 compact array chain walks if there is no hash collision in the 32 bit hashes. Compared with the linked list bucket design, the compact array shows better performance with a smaller memory footprint as long as the HT load factor (#entries/#buckets) stays no greater than 4. (See Section 4.2 for details.)

Software prefetching also helps reduce the impact of D$ misses by hiding the D$ miss latencies. Due to the need for sequential lookups in NDN forwarding (Interest: CS→PIT→FIB×n, Data: CS→PIT), an HT bucket used for a future lookup can be prefetched. We have examined 2 schemes: (1) *cascade* – a prefetch instruction is issued at 1 prior lookup (i.e. prefetch PIT bucket at CS lookup,

prefetch nth FIB bucket at PIT lookup and so on), and (2) *all-at-once* – all prefetch instructions are issued at the beginning just after computing all hashes $(H_n,...,H_1)$ but before any lookups. With our evaluation, all-at-once prefetching exhibits better overall performance because the memory latency is much longer than one HT lookup cycle. Therefore, we have employed all-at-once prefeching for our implementation. Bucket preallocation in the compact array design is necessary for the prefetching efficiency because it enables prefetching of 7 entries, not just a pointer.

2.3 FIB lookup algorithm

In a typical Interest forwarding scenario, the FIB HT lookup can be a bottleneck because lookup is iterated until it finds the longest prefix match (LPM). This problem is called *prefix seeking* [8], and it is common to any HT-based lookup approach. The most discussed solution is using Bloom filters (BF) to store the compressed FIB prefixes [18, 22]. This is quite effective when the hardware (ASIC or FPGA) performs parallel BF-check on the faster memory (SRAM) and the hash table in the slower memory (DRAM or RL-DRAM) is accessed only for the prefixes where BF returns FALSE. However, it is not appropriate for software implementation because a BF-check operation (that may involve multiple hash computations and a single memory read) is iterated for the maximum n times. For software forwarding, we need a different approach.

The basic HT-based lookup algorithm – *longest-first* strategy as lookup starts from the longest prefix – is undesirable for both performance and security reasons. In terms of performance, it will result in a large average lookup time because FIB matches tend to happen at prefixes considerably shorter than the full name. This is expected because publishers advertise short prefixes rather than the full object names and hierarchical name prefixes typically get aggregated inside the network. In terms of security, it is not good either because the worst-case lookup time is determined only by the number of name components in the Interest packet. With the longest-first strategy, non-matching (or default-route matching) Interests consume the most lookup time because it requires n FIB lookups. This causes a serious security issue because it makes a router vulnerable to an DoS attack against FIB processing wherein an adversary injects Interest packets with long non-matching names containing garbage tokens. On the other hand, one could consider the opposite *shortest-first* strategy. The FIB match may be found earlier with this approach, but the LPM algorithm still does not terminate until it examines all prefixes in order to guarantee that there is no longer prefix match.

To address this, we propose a *2-stage LPM algorithm* using *virtual* FIB entries (or prefixes). With this scheme, lookup first starts from a certain short (M component) name prefix and either continues to shorter prefixes or restarts from a longer (but shorter than the longest) FIB prefix if needed. Every FIB entry with M prefixes maintains the maximum depth (MD) of prefixes that start with the same prefix in the FIB. At the first stage of the algorithm, lookup
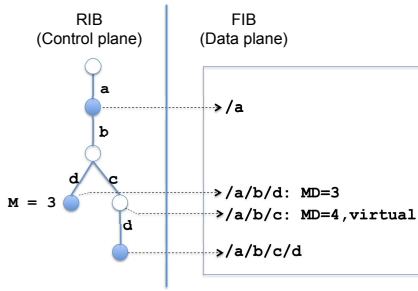
RIB
(Control plane)

FIB
(Data plane)

a

b

d   c

M = 3

d

→ /a

→ /a/b/d: MD=3
→ /a/b/c: MD=4,virtual

→ /a/b/c/d

Figure 4: Example of RIB and FIB.

Table 2: Lookup count saving with 2-stage FIB lookup.

m, n, M	MD	FIB lookup count	Lookup saving*
$m \leq n < M$	N/A	$1 + n - m$	0
$m \leq M \leq n$	$M = MD$	$1 + MD - m$	$n - MD$
	$M \leq MD < n$	$1 + MD - m$	$n - MD$
	$M < n \leq MD$	$1 + n - m$	0
$M < m \leq n$	$(MD + 1) \leq n$	$2 + MD - m$	$n - MD - 1$
	$n < (MD + 1)$	$2 + n - m$	(-1)

n: number of name components in an Interest packet
m: prefix depth where an Interest name matches at the FIB ($m \leq n$)
MD: maximum depth stored in the FIB entry at M
* Lookup count is always $(1 + n - m)$ with the longest-first algorithm.

starts from M name prefix and continues to shorter prefixes until it finds a match. If no match is found at this stage, there is no LPM for this Interest. If a match is found but it indicates that there is a possible LPM at the maximum prefix depth of $MD > M$, lookup starts the second stage from MD and continues until it finds a match.

Let's suppose an example that $M = 3$ and /a, /a/b/d, and /a/b/c/d exist in a routing table. When the name prefix /a/b/c/d is added to the FIB, a virtual FIB entry /a/b/c is also added, which stores $MD = 4$. (If a virtual or real /a/b/c already exists, MD is updated.) The prefix /a (and /a/b/d) inserts only one entry into the FIB because its prefix depth is not larger than M. As illustrated in Figure 4, the routing table maintained by the control plane (RIB, Routing Information Base) is represented as a name component prefix tree while the FIB on the data plane is our hash table. For forwarding an Interest /a/b/c/d/e/f, lookup starts from /a/b/c. Since it finds a match at /a/b/c and $MD = 4$, lookup restarts from /a/b/c/d then finds the LPM immediately. When an Interest /a/c/d/e/f/g comes, it does not find a match at /a/c/d (i.e. there is no longer prefix in the FIB starting with /a/c/d), therefore continues to /a/c, and then finds the LPM at /a. For an Interest /a/b/d/e/f/g, lookup starts at /a/b/d

Algorithm 1 2-stage FIB lookup algorithm.

```
 1: FIB_LPM_STAGE_1:
 2:  n ← number of name components in an Interest packet;
 3:  for pfx ← min(n, M) to 1 do
 4:    FIB_entry ← LookupFIB(name, pfx);
 5:    if FIB_entry then
 6:      if MD(FIB_entry) > M and n > M then
 7:        start_pfx ← min(n, MD(FIB_entry));
 8:        FIB_entry_1s ← FIB_entry;
 9:        goto FIB_LPM_STAGE_2;
10:      else if not IsVirtual(FIB_entry) then
11:        return FIB_entry;
12:      end if
13:    end if
14:    pfx ← pfx − 1;
15:  end for
16:  return FIB_NOT_FOUND;
17: FIB_LPM_STAGE_2:
18:  for pfx ← start_pfx to 1 do
19:    if pfx = M then
20:      FIB_entry ← FIB_entry_1s;
21:      if not IsVirtual(FIB_entry) then
22:        return FIB_entry;
23:      end if
24:    end if
25:    FIB_entry ← LookupFIB(name, pfx);
26:    if FIB_entry and (not IsVirtual(FIB_Entry)) then
27:      return FIB_entry;
28:    end if
29:    pfx ← pfx − 1;
30:  end for
31:  return FIB_NOT_FOUND;
```

and immediately finds the LPM. Algorithm 1 shows the details how the algorithm works.

The 2-stage FIB algorithm saves FIB lookups in most cases. In the example above, FIB lookup counts are 2, 3, and 1 instead of 3, 6, and 3 respectively. With a good choice of M, most Interest packets need only the first stage that performs M or fewer lookups. In case that it needs the second stage, lookup may start at a shorter prefix (MD) than the longest prefix of the input name. Table 2 summarizes FIB lookup count savings in various conditions. There is only one case that lookup is penalized by our method; (-1) saving (i.e. $+1$ in lookup count) when the FIB match happens at the prefix longer than M and the name is shorter than $(MD + 1)$ at the bottom. (e.g. with the same example FIB, an Interest name /a/b/c/d needs 1 lookup with the longest-first algorithm, but it requires 2 lookups at /a/b/c and a/b/c/d with our algorithm.) In all the other cases, our algorithm performs as well or better than the basic algorithm in terms of FIB lookup count.

Besides improving the average FIB lookup time, our algorithm provides DoS-resistance for FIB processing because it makes the worst-case lookup time bounded and independent of input names. With our algorithm, the worst-case lookup count is only a function of FIB (or RIB). If no FIB match is found at M, the FIB lookup count is bounded by M. Otherwise, it is bounded by MD, which is no greater than the maximum MD in the FIB. The bounded worst-case lookup time makes an NDN router less vulnerable to DoS attacks against FIB processing. In the garbage-token name attack scenario described earlier, the FIB lookup count will be always M with our algorithm while it will be n with the longest-first algorithm.

One disadvantage of our method is a FIB expansion caused by additional virtual FIB entries. However, we expect that impact of FIB expansion will not be significant with a good choice of M because it is less likely that the longer prefix is added to the FIB without any matching shorter prefixes. (e.g. when a prefix /com/youtube/cats is inserted to the FIB, /com/youtube is likely to be present as well.) Also note that a FIB expansion can be always statically computed from the RIB for a certain M. Any M that causes the FIB expansion over a certain threshold (e.g. 10%) should be avoided because the over-populated HT entries will hurt the lookup performance eventually by increasing the average HT chain length.

There are 2 more questions to be answered with regard to this algorithm. The first is how to determine the parameter M. As shown in Table 2, we reduce FIB lookup counts in most cases. However, if M is too large ($m \leq n < M$), there is no lookup saving. If M is too small ($M < m \leq n$), lookup can be penalized in some cases. The most lookup saving comes form the sweet spot where M is in the middle ($m \leq M \leq n$). With our initial assessment, we have found that selecting the optimal M without knowing the actual input set is extremely hard because the final result is determined by the component distribution of input names, FIB prefixes

and their matching statistics. We also think that it may be too early to develop such an accurate scheme because it should be based on an analysis of the real NDN traffic traces and FIB data, which are not available yet.

For now, we can think of 2 simple heuristics that work based on the FIB (or RIB) name component (prefix) distribution: (1) *most-likely* – choose M that contains the most number of components, or (2) *cumulative-threshold* – choose M where the cumulative component distribution exceeds a certain threshold such as 75%. With either heuristic, a FIB expansion should be considered as a negative decision factor. Since choosing a threshold value is another question when it comes to the cumulative-threshold heuristic, we have used the most-likely heuristic for our experiments shown in Section 4. By our evaluation with a bell-shaped FIB component distribution, the most-likely heuristic works fine showing only 2% forwarding performance degradation compared to the optimal M. (See details in Section 4.3.) As we better understand actual NDN traffic patterns, we can investigate more accurate methods, which we will leave for future work.

The second question is how it works in a router. In reality, M should change over time as the RIB changes. Since the choice of M and the virtual FIB entries (and MD values) are coupled, the FIB needs careful updating when M is updated. All virtual FIB entries associated with the new M should be inserted first (and MD updated for every existing node at M) before a router updates M. The virtual FIB entries for the old M should stay as long as any packet processing threads are using the old M, and should be deleted later. Note that inserting new virtual FIB entries does not affect any forwarding operations with the old M. In theory, forwarding can use a different M for each packet as long as the FIB has all necessary virtual prefixes for multiple M. (However, this is at the expense of larger FIB expansion.) The initial value of M can be large when the RIB is empty and then reduced as the RIB grows. Though there is no lookup saving (nor penalty) with large M, it still provides DoS-resistance with bounded worst-case FIB lookup time.

2.4   CS and PIT lookup

HT-based lookup fits quite well for PIT and CS lookup because it only uses the full-name as the key for matching[1] and an update (insert or delete) operation is easy with a hash table. On top of the basic HT lookup used for FIB, we have devised 2 more techniques to optimize access for PIT and CS.

First, PIT and CS lookups are combined into one HT lookup because they are adjacent and use the same full-name key. By combining CS and PIT lookup into a single hash table, it reduces 2 separate HT lookups to one. This lookup time saving applies to both Interest and Data processing because both require CS and PIT lookup at the beginning. In addition, our approach reduces the frequency of HT insertion and deletion on PIT and CS. If we observe the lifetime of the CS and PIT entries associated with an Interest/Data packet pair, PIT entry deletion and CS entry insertion happen at the same time when the Data packet arrives. This is shown in Figure 5; *CS insert* means a new Data packet is being cached while *CS delete* means the Data packet is evicted from the CS. (We assume that a simple replacement policy such as FIFO or LRU is used for the CS; this is reasonable because a caching decision for the CS should be done in a wire speed as part of the forwarding process in NDN.) In a typical Data forwarding scenario, 'PIT lookup & deletion + CS LRU replacement' requires 2 delete and 1 insert operations while

---

[1] NDN supports the LPM lookup for the PIT and CS as well. However, we do not believe that this type of matching is feasible at scale, and creates more problems. Hence, our implementation deliberately does not support this. Details are not within the scope of this paper.
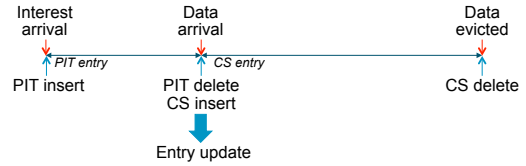

Figure 5: Lifetime of PIT and CS entries.

sharing an CS/PIT entry needs only 1 deletion because a PIT entry simply evolves to a CS entry by an entry update as shown in Figure 5. For this to work, a flag bit is used to distinguish whether an entry belongs to CS or PIT.

The second technique to optimize PIT and CS lookup is to partition the PIT (i.e. combined CS/PIT) across cores (or threads). With our experiments, we have found that sharing PIT entries among parallel packet processing threads significantly impedes multi-core parallelization speedup because (1) a read or write lock on the shared entry can block threads due to access serialization, and (2) a memory write on the shared entry causes invalidation of copies in other per-core caches due to the cache coherence mechanism (e.g. MESI) and therefore lowers D\$ utilization dramatically. PIT partitioning enables each core to exclusively access its own PIT without any locking or data aliasing among cores. For this to work, Interest and Data packets with identical names must be looked up in the same PIT instance. This is achieved by distributing packets to cores based on names. In our implementation, we use full-name hashes to distribute packets to the forwarding threads. This is efficient because the full-name hash needs to be computed in any case for the CS/PIT HT lookup.

2.5   Content Store design

There are 2 levels of caching that can be done in an NDN router. Short-term caching of packets provides for any re-transmission that is needed in case of link congestion, lower-layer transport errors or client mobility. Long-term caching of Data packets is mainly for efficiently serving popular content. NDN allows long and short-term caching to share one mechanism because NDN architecturally integrates storage as a forwarding component. However, from a router implementation point of view, the different resource requirements lead us to treat short and long-term caching separately.

Short-term caching will be implemented as a packet buffer mechanism with the same memory technology used for the forwarding data structures. In today's IP router architecture, the packet buffer provides 2 basic primitives to a forwarding engine: (1) *rx* – receive a packet, which is read by the forwarding engine, and (2) *tx* – transmit a packet after it is processed by the forwarding engine. The lifetime of a packet in the buffer is coupled with each primitive; *rx* is the beginning while *tx* is the end. To support caching using a packet buffer mechanism, the lifetime of a packet must be decoupled from current tx/rx primitives and the packet buffer has to support at least 3 additional primitives: (1) *tx-and-store* – transmit and cache a packet (extend the life of a packet), (2) *delete* – evict a packet (end the life of a packet), and (3) *tx-from-buffer* – transmit a packet directly from the packet buffer. If caching is implemented as a forwarding engine only function without these mechanisms, expensive memory copy operations are needed to move big Data packets in and out.

Long-term caching is typically implemented using non-volatile storage media such as hard drives or SSDs. This can represent another huge challenge because a router still needs to service Data packets at wire-speed while keeping them in a storage device. It may also require a complex replacement policy to cache popular

(a) ASR 9000   (b) Integrated Service Module
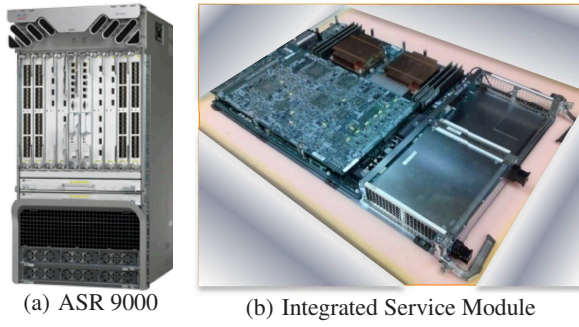
Figure 6: ASR 9000 router and ISM.

Figure 7: NDN packet flow in ASR 9000 with ISM.

contents in a long term. We are currently working on implementing the first level of CS in the packet buffer. The complete design and implementation of a Content Store with hierarchical storage will appear in our future work.

3. SYSTEM IMPLEMENTATION

3.1 Target platform

To implement our NDN data plane design on a router, we have selected the Cisco ASR 9000 router with the Integrated Service Module (ISM) [4] shown in Figure 6. This is a flexible platform because forwarding on an ISM is purely done by the software running on 64-bit Linux. The ISM is currently used for video streaming and NAT applications; it features 2 Intel 2.0GHz hexa-core Westmere EP (Xeon) Processors with Hyper-threading (24 logical cores) and 4 Intel Niantic 10GE interfaces. Additionally, it can include up to 3.2 TB of SSD with 2 modular flash SAM (Service Accelerator Module) that can be used as NDN cache storage.

3.2 Packet format

The wire format of packets and names has a big impact on forwarding. The XML packet and name format currently used in CCNx [3] – PARC's implementation of the NDN architecture – provides great flexibility for early research. However, its CCNb binary encoding incurs significant decoding overhead as it consumes 35% of execution cycles for forwarding [29]. We have found that the CCNb is not forwarding-friendly mainly for 2 reasons: (1) there is no easy way to locate the name tag without decoding all proceeding tags, and (2) it includes unnecessary overhead characters that decrease the lookup efficiency.

In order to address the issues with the current CCNb format, we are currently working on defining a flexible and forwarding-friendly binary NDN packet format based on a fixed header and TLV (Type-Length-Value) fields. Adding the length at every variable-length field will be critical for forwarding performance because it enables the data plane to quickly skip the fields that are not used for forwarding. The details are not the main interest of this paper and the final format proposal will be published to the NDN community separately. For our experiments, we use a simple format that includes a 1-byte header with a packet type and a simple name format, which includes the length of the full name and encoded name components with the component length and characters.

3.3 Packet input and output

Software routers make use of a software forwarding engine built with off-the-shelf general-purpose hardware. Since our platform is based on a server-like system architecture with Intel Xeon CPUs
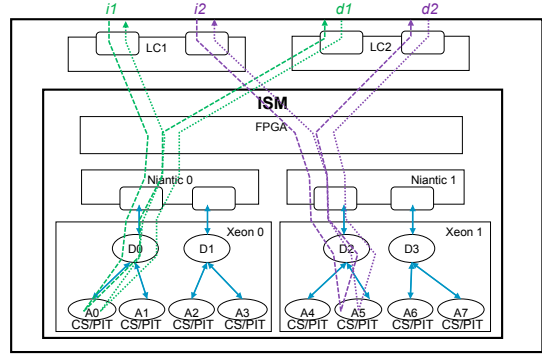
and Linux OS, it can be considered as a software router. There are two well-known software router designs: RouteBricks [11] parallelizes router functionality cross the multiple CPU cores in a single server as well as multiple servers, and PacketShader [6] exploits the massively-parallel processing power of GPU to address the CPU bottleneck in current software routers.

Based on lessons learned from those designs, we have applied 2 techniques for fast packet input and output in our system. First, we bypass the Linux networking kernel stack completely for fast packet delivery. Our packet I/O system is built on user-space device drivers for the Intel 10GE Niantic Network Interface Card (NIC) and DMA (Direct Memory Access) [16, 17]. For both incoming and outgoing packets, the user-space device driver directly transfers packets between an NIC and a forwarding process via DMA.

Second, we utilize a batch processing mode in the NIC driver to reduce per-packet processing overhead [11]. Our I/O system can process a batch of 255 packets with one receive or transmit transaction. Multiple Rx/Tx queues in the Niantic NIC can also be used to boost I/O performance when they are accessed by different CPU cores [6]. However, this is not an option for our system because the packet that arrives on the Niantic NIC in the ISM does not begin with a standard Ethernet header, but the custom Network Protocol Header (NPH) compatible with the ASR 9000 switching fabric. Furthermore, support for multiple Rx queues requires the NIC to hash the address and port fields in the packet to determine which Rx queue the packet goes to, which would not be straightforward for NDN packets.
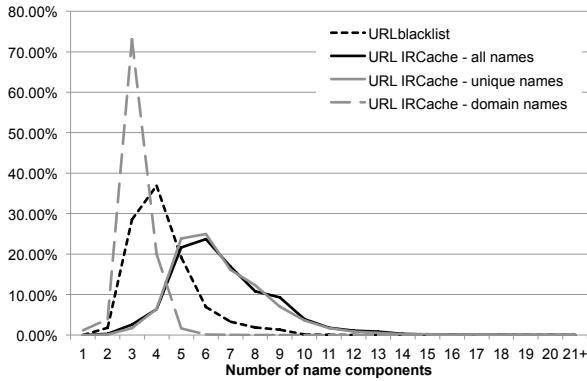
In our implementation, there is a process (*Dispatcher*) designated for packet I/O for each 10GE interface and NDN packet forwarding is performed by separate processes (*App*). The packet buffer resides in a shared memory region that both types of process can access.

3.4 System overview

Since our NDN data plane runs on a service line card in an IP router, the NDN traffic flows as an IP overlay. Figure 7 shows how NDN packets are processed in the ASR 9000 with an ISM. The router is connected externally to the interfaces on regular 10GE transport line cards (LC1-2). A high-speed switching fabric carries packets between the ISMs and transport line cards. Each ISM has 4 10GE internal interfaces to the fabric. When an NDN packet arrives at a fabric interface on the ISM, it is received by a *Dispatcher* process (D0-3) attached to that interface by the user-space driver via DMA. The *Dispatcher* identifies the offset of the name in the packet and computes the hash of the full name, which is used to identify which *App* process (A0-A7) will process the packet (in order to support the partitioned PIT). The full-name hash is passed in

220

Table 3: 16 IRCache URL traces.

Trace	Total # of names	Total unique names	Average name comp	Average chars / name	Average chars / comp
bo2-09	237869	143916	6.47	58.62	9.06
bo2-10	205993	132971	6.48	57.20	8.83
ny-09	446555	297075	6.42	57.21	8.92
ny-10	389987	274104	6.44	56.36	8.75
pa-09	206337	129584	6.42	54.77	8.53
pa-10	280047	161901	6.57	59.17	9.01
rtp-09	3164942	1649229	6.68	60.30	9.03
rtp-10	2976468	1497909	6.72	59.23	8.82
sd-09	1418522	875807	6.54	56.52	8.64
sd-10	1493782	931209	6.58	56.59	8.60
sj-09	361035	156182	7.08	57.52	8.13
sj-10	346382	150764	7.13	57.75	8.10
sv-09	527132	191376	6.97	60.10	8.62
sv-10	447600	196830	6.83	61.53	9.01
uc-09	527276	277148	6.50	55.20	8.49
uc-10	518888	269054	6.63	55.86	8.42
Total/Avg.	13548815	7335059	6.66	58.39	8.78

Table 4: Memory consumption (MBytes) by FIB HT.

FIB entries	1M	4M	16M	64M
Compact array buckets	16.00	64.00	256.00	1024.00
Dynamic buckets	0.82	3.27	13.10	52.40
FIB entries	64.00	256.00	1024.00	4096.00
FIB names	71.72	263.73	1031.73	4103.73
Total consumed	152.56	587.00	2324.83	9276.13
Total for lookup	35.82	143.27	573.10	2292.40

Figure 9: Performance vs. HT load factor.

Figure 8: Name component distribution of traces.

with the packet buffer so that the *App* need not recompute it. The *App* performs all the NDN forwarding operations – CS/PIT lookup, FIB lookup – and then returns the packet to the *Dispatcher* along with output interface information. The *Dispatcher* transmits the returned packet to the outgoing interface through the switching fabric. By varying the number of *App* processes, our architecture can correctly balance the work performed at each step for maximum utilization.
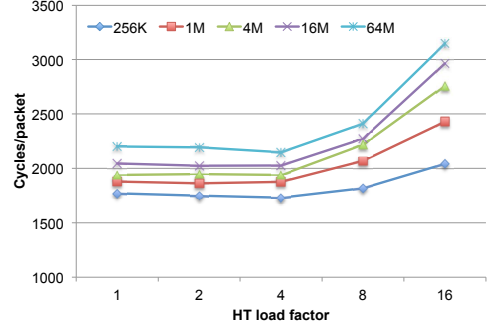
4. EXPERIMENTAL RESULTS

4.1 Workload generation

Since NDN is a future Internet architecture, it is difficult to obtain workloads from real traffic traces. Researchers to date have translated existing Internet traces to generate NDN workloads [5]. For our experiments, we used the same approach. We have extracted 13 million HTTP URLs from 16 IRCache traces [12] and converted them into NDN names using the following method: (1) each sub-domain name is converted into a single NDN name component in a reverse order, (2) the entire query string after '?' character is always treated as one component, and (3) every '/' character in the rest of URL is considered as a name component delimiter. (e.g. `cisco.com/ndn` is converted to `/com/cisco/ndn`.) Table 3 presents the statistics of NDN names generated from the IRCache traces.

Building a realistic routing table (FIB or RIB) is another issue because NDN has not been deployed at Internet scale. Since no

previous work has a good description of how to generate an NDN routing table, we have developed our own method. Our hypothesis is that an NDN routing table has a similar long-tailed component distribution to that of NDN object names, but with a smaller average and more skewed shape. *URLblacklist* [21] provides a collection of domains and short URLs for access blocking. As seen in Figure 8, its distribution matches our hypothesis; a long-tailed distribution with the average 4.26 components (compared with 6.66 for IRCache trace names). Another possible candidate for the FIB distribution is the domain name distribution. However, it was rejected because its distribution is heavily skewed; most (73%) names have 3 name components.

Based on the URLBlacklist component distribution, our FIB names are synthesized from all possible name prefixes of 16 traces so that they match the target distribution. As a result, our Interest forwarding workload results in an average of 3.87 FIB lookups; the mean name component distance between Interest and FIB names is 2.40.

4.2 Scalability of HT-based lookup

In order to examine the scalability of our HT-based lookup, we have evaluated Interest forwarding performance by varying target FIB sizes from 256K (K=2^{10}) to 64M (M=2^{20}) entries. Figure 9 shows the cycles/packet for Interest forwarding by increasing HT load factors (LF=$FIBsize/HTsize$) from 1 to 16. (e.g. when $FIBsize$=1M and $HTsize$=256K, LF=4.) Since our synthetic FIB generated from the traces only provides about 1M entries, random FIB entries are interleaved during insertion of actual FIB entries if the target FIB size is larger than 1M. For this experiment, the baseline forwarding code is used and the PIT $HTsize$ is fixed to 256K. Interest packets are generated from unique names of bo2-09 trace and the Interest forwarding involves a CS/PIT miss and the average 3.70 FIB lookups.

As shown in Figure 9, performance is rarely affected by increasing the load factor up to 4, but it gets noticeably worse when the LF is 8 or greater. This is the effect of our D$-friendly HT data structure because it incurs only one D$ miss if a matching entry is found within 7-entry walk. When the LF is 4, the average HT chain length is 4.07 and the compact array contains about 89% entries. However, with the LF of 8, the average chain length is 8 and

Table 5: Impact of 2-stage FIB lookup algorithm with different M.

Configuration	Baseline	M=3	M=4	M=5	M=6	M=7	M=8
Average # FIB lookup per Interest	3.87	3.79	3.18	3.08	3.18	3.36	3.56
FIB lookup saving		1.99%	17.64%	20.21%	17.78%	12.98%	7.94%
FIB size (# entries)	990031	990031	1053043	1053843	1034548	1018426	990031
FIB expansion		0.00%	6.36%	6.45%	4.50%	2.87%	0.00%
Performance improvement	0.00%	3.99%	7.34%	8.69%	8.33%	6.82%	4.85%
FIB component distribution		28.5%	36.9%	19.1%	6.9%	3.3%	3.4%
FIB component cumulative		30.3%	67.2%	86.4%	93.3%	96.6%	100%

the compact array only includes 31% of entries. It means that most HT chain walks involve multiple D$ cache misses when the LF is 8 or larger. Since the memory size required for the compact array is inversely proportional to the load factor given a target FIB size, the LF of 4 is the sweet spot for our HT data structure in terms of space-time trade-off. The LF of 4 is therefore the best design target size of the hash table; however, our implementation performs quite well even if the initial design was short by 100% (i.e. LF=8).

With the same load factor of 4, the lookup performance is negatively affected by increasing the FIB sizes. This is because the working set size increases with a larger HT while the D$ size is fixed. Assuming that a random bucket is selected from $HTsize$ buckets, the probability that the bucket used for a subsequent lookup exists in the D$ (i.e. D$ hit) linearly decreases as increasing $HTsize$ – approximately $(D\$size/64)/HTsize$. However, the impact of D$ misses will become smaller as increasing $HTsize$ because this probability converges to 0. As shown in Figure 9, the average performance degradation by quadrupling the FIB sizes between 1M, 4M, 16M and 64M is 4.22% while it shows 7.78% degradation from 256K to 1M.

Table 4 shows the FIB HT memory consumption for different target FIB sizes. In our implementation, every piece of memory used dynamically is preallocated and then assigned as needed in a cache line unit of 64 bytes. Total memory consumption (Total consumed) includes the total cache lines used for buckets, entries and names in MBytes. If we only count the space purely used for lookup (Total for lookup) except names and unused fields in a FIB entries (only 17 bytes are used for lookup including 64-bit hash and etc.), the HT only constitutes about 25% of the total consumption. Compared with a Trie-based approach used in Name Component Encoding (NCE) [24], the memory consumption of our HT data structure is competitive; NCE's lookup data structure (ENPT-STA) consumes 55.30 MBytes for about 2 million FIB entries while ours needs 35.82 for 1M entries. (Note that this is not an exact apples-to-apples comparison because of differences in input name sets.)

4.3 Impact of FIB lookup algorithm

To examine effectiveness of our 2-stage FIB lookup algorithm, we have run the forwarding code on the ISM with about 7 million Interest packets generated from all 16 unique-name traces and the same synthetic FIB based on URLBlacklist. Table 5 summarizes the results; the first 2 rows present FIB lookup count savings and the next 2 rows show FIB expansion with different M configurations. At M=5, it achieves the best lookup count saving of 20.21% at the penalty of the largest FIB expansion of 6.45%. However, both M=4 and M=6 show quite similar lookup savings and FIB expansions as M=5. The best performance improvement of 8.69% is at M=5, but M=4 and M=6 also perform quite well; only 1.35% and 0.36% lower than the best.

The last 2 rows show the name component distribution and its cumulative distribution, which can be used for the M-selection heuristics suggested in Section 2.3. The most-likely method chooses M=4 while the cumulative-threshold method chooses 4, 5 or 6 re-
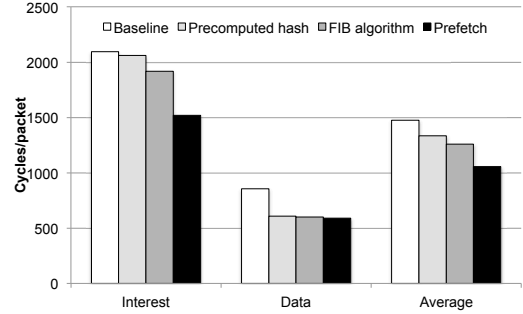
Figure 10: CPU cycles for NDN forwarding.

spectively for thresholds of 50%, 75% and 90%. Since the cumulative-threshold method presents difficulties with choosing a threshold, we instead use the most-likely heuristic for the following performance evaluation (i.e. M=4 with the third best performance). Though details are not shown here, we also have evaluated our algorithm with another synthetic FIB generated from domain names, which have a more skewed name component distribution in Figure 8. With this FIB, M=3 chosen by the most-likely heuristic shows the best FIB lookup saving of 68.29% (28.59% speedup) with a 10.11% FIB expansion.

4.4 Single-core performance

We directly measured the performance of our NDN forwarding engine on the target platform, ISM with NDN forwarding simulator. This runs the identical forwarding code but reads packets from files instead of NICs. Figure 10 shows the per-packet CPU cycles for Interest, Data packet forwarding and the average by single-core simulation. In order to measure Interest and Data forwarding performance separately, every trace with unique names is generated into a series of Interest and Data packets that are sequentially fed into the simulator. Therefore, the workload for Interest forwarding includes a CS/PIT miss, PIT insertion and the average 3.73 FIB lookups while Data forwarding involves a CS miss, a PIT hit & deletion, and CS replacement. It shows that how much performance gain (i.e. cycle count reduction) is achieved as we incrementally apply our proposed design techniques; *Precomputed hash* – the full-name hash (H_n) is pre-computed by a Dispatcher process, *FIB algorithm* – 2-stage FIB lookup algorithm, and *Prefetch* – software prefetching. Note that right one includes techniques shown on the left; *Prefetch* bars show the cycles when all techniques are combined. The target FIB and PIT sizes are set to 16M and 1M entries respectively with the same HT load factor of 4, and M=4 based on most-likely heuristic when 2-stage FIB algorithm is used.

The precomputed hash gives 1.76% gain for Interest while it gives 40.76% gain for Data. The reason is that Interest and Data forwarding use different types of the hash function; the Interest forwarding code uses our own implementation of SipHash that decodes name components and simultaneously computes all prefix
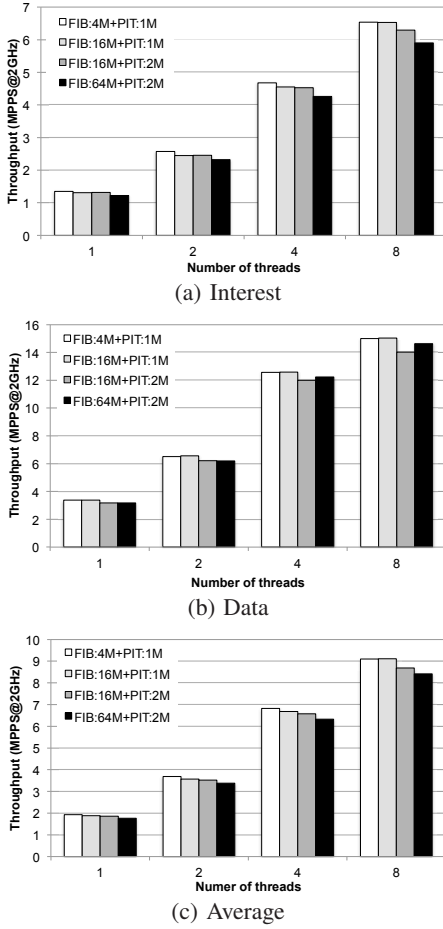
(a) Interest

(b) Data

(c) Average

Figure 11: Multi-core forwarding performance.

Figure 12: Performance with real system configurations.

hashes incrementally while the Data forwarding uses the default SipHash function to compute the full-name hash only. Therefore, precomputed full-name hash dramatically reduces the Data forwarding cycles because it offloads the major workload while it reduces a small portion of hash computation workload for Interest forwarding. However, combining all techniques for Interest forwarding gives 37.87% speedup because our Interest forwarding code is optimized so that it performs name parsing, SipHash computation, and the first stage of FIB lookup algorithm simultaneously via software prefetching. Putting this all together, our forwarding engine on a single core shows an average forwarding throughput of 1.90 MPPS; 1.32 MPPS Interest and 3.39 MPPS Data forwarding respectively.

4.5 Multi-core and system performance

Figure 11 shows the performance with multi-core simulation varying FIB and PIT sizes from 1M to 64M entries. Note that we use MPPS (Million Packets per Second) instead of CPU cycles, and higher bars hence indicate better performance. The same workload from 16 traces is used but we increase the number of parallel forwarding threads from 1 to 2, 4, and 8 running on different CPU cores. Since there are only 6 cores on a single Xeon processor, we utilize 8 cores across 2 CPUs for 8-thread simulation so that 4 parallel threads run on one CPU and the rest on the other. It achieves near-linear speedup as increasing the number of threads up to 4; multi-core speedup is 3.47 for Interest, 3.76 for Data and 3.54 in
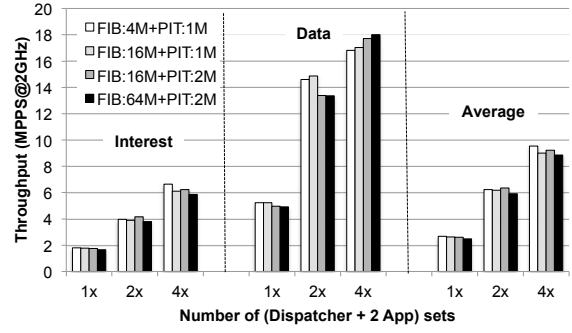
average. With 8 threads, it however shows only 4.86, 4.47 and 4.77 multi-core speedup. This is due to the effect of the NUMA (Non-Uniform Memory Access) architecture. In 8-thread simulation, the main thread for simulation starts on one CPU but forwarding threads are invoked across 2 CPUs. In such case, thread invocation, packet distribution, and FIB lookup that happen on 4 cores in the other CPU require remote memory accesses, which take more time than local memory accesses that happens within the same CPU.

The throughput is minimally affected by different sizes of FIB and PIT; from the smallest to the biggest configuration for Interest forwarding, it shows the maximum of 9.51% performance degradation. By deploying 8 threads, our forwarding engine can process NDN packets at 8.82 MPPS; 6.30 MPPS Interest and 14.67 MPPS Data forwarding respectively.

For the actual forwarding system on the ISM, we use 4 Dispatcher processes designated for packet I/O with 4 10GE interfaces, and 8 App processes that run the NDN forwarding code as shown in Figure 7. In order to find the best mapping of threads to CPU cores, we first have measured packet forwarding throughput of a single Dispatcher; it shows 1.5 MPPS when Tx and Rx threads run on 2 different CPU cores. By precomputing the full-name hashes at the Dispatcher so that our CS/PIT partitioning scheme work, there is a slight performance drop; a single Dispatcher shows 1.3 MPPS.

Since our Interest forwarding throughput on a single core is slightly worse than this (1.22~1.35 MPPS), it still makes sense to combine 1 Dispatcher with 2 App processes in order that the NDN forwarding task not be a bottleneck in the system. For this mapping, 2 App processes have to share a single core by using 2 hyper-threads (HTh) because a Dispatcher preoccupies 2 cores.

In order to examine the impact of using hyper-threads, we have performed the same set of simulations by applying the CPU mapping used by the Dispatcher and App processes in the system. Figure 12 shows the throughput with real system configurations: 1x - 2 HTh on 1 core, 2x - 4 HTh on 2 cores, 4x - 8 HTh on 4 cores across 2 CPUs. It turns out that the use of hyper-threads does not limit the performance; when using 8 hyper-threads in 4 cores (4x) across 2 processors, it shows the almost identical throughput to the case when using 8 full cores.

By fully utilizing 4 sets of (Dispatcher + 2 App), the system currently shows about 4.5 MPPS forwarding throughput. Assuming the average NDN packet size is 550 bytes with small Interest but large Data packets, the throughput of our NDN forwarding is above 4.39 MPPS required for 20Gbps NDN traffic. We believe that further system-level optimization (e.g. improving the driver code for faster packet delivery) may result in better utilization of resources and we will leave it for future work.

5. RELATED WORK

The NDN project proposal [18] contains a good overview of various design issues with NDN including forwarding. Yuan et al. [29] focused on NDN forwarding and identified 3 key issues: exact string matching with fast updates, longest prefix matching for variable-length unbounded names, and large-scale flow-maintenance based on a detailed analysis on the CCNx forwarding software.

Early research on a content centric router focused on feasibility. Perino et al. [19] evaluated the performance of CCN forwarding components (CS, PIT and FIB) based on system modeling of the hardware and the software technologies used in modern routers. They concluded that a CCN router can be deployed in ISP scale while it is not ready for Internet scale deployment. Arianfar et al. [1] proposed a sample line-speed content centric router design and estimated the amount of resources used by a router with regard to the interaction with the pull-based protocols.

5.1 Trie-based name lookup

The Named Component Encoding (NCE) [24] was the first attempt to address the variable-length tokenized name lookup problem with a Trie-based approach. With NCE, every name component is encoded as a code (or symbol) and FIB lookup is performed with a series of encoded symbols. The FIB is represented as an Encoded Name Prefix Trie (ENPT) while the component symbols are stored and looked up in a Component Character Trie (CCT). Though the LPM lookup on an ENPT shows a reasonable performance (1800∼3200 CPU cycles/packet), a bottleneck exists at the encoding process (i.e. CCT lookup) as 3 parallel code lookup modules are used for 1 ENPT lookup module. The same ENPT was applied to PIT lookup as well with an improved data structure (Simplified STA) and a hash-based code allocation function [5]. For the throughput requirement for PIT accesses, it requires 4 parallel encoding modules to keep a single lookup module busy.

In order to avoid the bottleneck in the encoding process, Wang et al. [25] used a character Trie instead of ENPT. To store a sparse tree in an effective manner, a special data structure, aligned transition array (ATA) or multi-stride ATA was used. A software lookup engine was implemented on a PC with Intel CPUs and NVIDIA GPU and it shows the great performance (63M searches per second with 2×512 cores) that can be obtained by utilizing massive parallel cores in the GPU.

We advocate an HT-based design for NDN name lookup because we believe it is better matched the nature of the NDN lookup problem. When using a component prefix tree (Trie), it requires a special encoding scheme, which introduces a serious amount of extra workload and often may end up being a bottleneck of the whole lookup process. In case of a character Trie, it does not have the encoding issue, but the lookup efficiency may decrease linearly as the length of an input name grows because the lookup is processed in a byte unit. This makes it hard to design a lookup engine with predictable performance. An HT-based engine shares the same concern with regard to hash computation, but it affects the only small portion of the whole lookup process; once hashes are generated, lookup mostly uses hash values. Lastly, while simple Trie-based data structures handle updates fairly well, most data structures used to efficiently pack the Trie are not as update-friendly and therefore they are not adequate for PIT and CS lookup.

5.2 Hash table-based name lookup

Caesar [22] is the first content centric router design with a hash table based lookup engine. The FIB is distributed across the line cards based on the first name component hashes (i.e. 3-stage forwarding: ingress LC, FIB-storing LC, and egress LC). At each

LC, FIB is represented as 2 data structures: counting Bloom filters (BF) in on-chip SRAM and a hash table in off-chip DRAM (or RLDRAM). It is based on the *Bloom-filter accelerated longest prefix match* design [18], which is effective with appropriate hardware support. Different from Caesar, our forwarding engine design is targeted more to software forwarding as it requires no extra logic or data structures other than HT. We pursue only algorithmic improvements and utilization of architectural features.

In order to address the *prefix seeking* issue without using Bloom filters, Fukushima et al. [8] proposed a FIB lookup scheme similar to our 2-stage FIB lookup algorithm. It suggests carrying a prefix length field at every Interest packet so that the next hop router can start seeking a prefix from the prefix where a previous hop router has found the LPM. Though it uses a similar idea as ours, it is different in 3 ways: (1) it requires protocol level support (while ours does not), (2) it does not provide resilience to DoS attacks against FIB processing, and (3) it may cause a significantly larger FIB expansion because it requires to add all dummy intermediate prefixes (while ours only requires intermediate *virtual* prefixes at a specific prefix depth, M). However, we consider that carrying such information in a packet will be useful for further optimization and hence these 2 schemes can be complementary in practice.

5.3 PIT design and NDN workload generation

In order to address the high scalability requirement for PIT, a few different distributed PIT designs have been proposed. DiPIT [28] represents PIT as multiple distributed Bloom filters associated with each face. Despite good scalability, DiPIT has issues such as requiring parallel lookups on all BFs for matching a Data packet, and mis-delivery of a packet in case of a false-positive match.

Regarding how to distribute PIT across the line cards (LC), Dai et al. [5] proposed an output LC placement along with Trie-based PIT lookup. Varvello et al. [23] suggested a third-party LC placement by applying the same approach used for the distributed FIB design [22]. In our router, this choice is not available to us because forwarding is centralized on the ISM. We instead address how to scale PIT inside a forwarding engine, and propose the partitioned CS/PIT across the packet processing elements (or threads) to achieve better performance and scalability.

Besides the PIT placement, Dai et al. [5] proposed a basic methodology to generate NDN workloads with approximate and application-driven translation of the current IP-generated traces. Though it is more comprehensive than ours in terms of NDN workload generation, it does not include any method to generate an NDN routing table (or FIB). In this paper, we have proposed a practical method for building the FIB from the name trace. We hope that this can advance efforts to generate realistic NDN workloads such as traffic traces and routing tables for future NDN research.

6. CONCLUSION & FUTURE WORK

In this paper, we proposed a design of an NDN data plane based on hash tables, which not only enables high-speed forwarding but also provides DoS-resistance for an NDN router. Our design includes (1) name lookup via hash tables with fast collision-resistant hash computation, (2) an efficient FIB lookup algorithm that provides good average and bounded worst-case FIB lookup time, (3) PIT partitioning that enables linear multi-core speedup, and (4) an optimized data structure and software prefetching to maximize data cache utilization.

We implemented this NDN data plane with a software forwarding engine on an Intel Xeon-based line card in the Cisco ASR 9000 router. By simulation with names extracted from the IR-Cache traces, we demonstrated that our forwarding engine achieves

a promising performance of 8.8 MPPS and our NDN router can forward the NDN traffic at 20Gbps or higher. Our future work includes full implementation of our NDN router including a hierarchical Content Store, further system-level optimization to improve the performance, and large-scale experimentation of forwarding with our NDN routers.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] S. Arianfar, P. Nikander, and J. Ott. On content-centric router design and implications. In *Proceedings of the Re-Architecting the Internet Workshop*, ReARCH '10, pages 5:1–5:6, New York, NY, USA, 2010. ACM.

[2] J.-P. Aumasson and D. J. Bernstein. Siphash: a fast short-input PRF. Cryptology ePrint Archive, Report 2012/351, 2012. http://eprint.iacr.org/.

[3] CCNx. http://www.ccnx.org.

[4] Cisco Systems. http://www.cisco.com/en/US/prod/collateral/routers/ps9853/data_sheet_c78-663164.html.

[5] H. Dai, B. Liu, Y. Chen, and Y. Wang. On pending interest table in named data networking. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 211–222, New York, NY, USA, 2012. ACM.

[6] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[7] ECHO Design Team. http://crypto.rd.francetelecom.com/ECHO/sha3/AES.

[8] M. Fukushima, A. Tagami, and T. Hasegawa. Efficiently looking up non-aggregatable name prefixes by reducing prefix seeking. In *Proceedings of the 2nd IEEE International Workshop on Emerging Design Choices in Name-Oriented Networking*, NOMEN '13, 2013.

[9] Google Inc. http://code.google.com/p/cityhash/.

[10] Grøstl Team. http://www.groestl.info.

[11] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[12] IRCache Traces. ftp://ircache.net/Traces/DITL-2007-01-09/.

[13] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, 2009.

[14] B. Jenkins. http://www.burtleburtle.net/bob.

[15] B. Kittridge. http://byteworm.com/2010/10/13/crc32.

[16] M. Krasnyansky. UIO-DMA. http://opensource.qualcomm.com/wiki/UIO-DMA.

[17] M. Krasnyansky. UIO-IXGBE. http://opensource.qualcomm.com/wiki/UIO-IXGBE.

[18] NDN project team. Named data networking (NDN) project. Technical Report NDN-0001, Oct. 2010.

[19] D. Perino and M. Varvello. A reality check for content centric networking. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, ICN '11, pages 44–49, New York, NY, USA, 2011. ACM.

[20] The OpenSSL Software Foundation. http://www.openssl.org/docs/crypto/md5.html.

[21] URLBlacklist. http://urlblacklist.com.

[22] M. Varvello, D. Perino, and J. Esteban. Caesar: a content router for high speed forwarding. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, ICN '12, pages 73–78, New York, NY, USA, 2012. ACM.

[23] M. Varvello, D. Perino, and L. Linguaglossa. On the design and implementation of a wire-speed pending interest table. In *Proceedings of the 2nd IEEE International Workshop on Emerging Design Choices in Name-Oriented Networking*, NOMEN '13, 2013.

[24] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen. Scalable name lookup in NDN using effective name component encoding. In *Proceedings of the 32nd International Conference on Distributed Computing Systems*, ICDCS '12, 2012.

[25] Y. Wangand, Y. Zuand, T. Zhangand, K. Pengand, Q. Dongand, B. Liu, W. Mengand, H. Daiand, X. Tianand, Z. Xuand, H. Wuand, and D. Yang. Wire speed name lookup: A GPU-based approach. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, 2013.

[26] Wikipedia. Birthday attack, 2013. [Online; accessed 23-May-2013].

[27] C. Yi, A. Afanasyev, L. Wang, B. Zhang, and L. Zhang. Adaptive forwarding in named data networking. *SIGCOMM Comput. Commun. Rev.*, 42(3):62–67.

[28] W. You, B. Mathieu, P. Truong, J. Peltier, and G. Simon. DiPIT: A distributed Bloom-filter based PIT table for CCN nodes. In *Proceedings of the 21st International Conference on Computer Communications and Networks (ICCCN)*, pages 1–7, 2012.

[29] H. Yuan, T. Song, and P. Crowley. Scalable NDN forwarding: Concepts, issues and principles. In *Proceedings of the 21st International Conference on Computer Communications and Networks*, ICCN '12, 2012.