Computer Networks 57 (2013) 3165–3177

Contents lists available at ScienceDirect

Computer Networks

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

GPU-accelerated name lookup with component encoding ☆

Yi Wang, Huichen Dai, Ting Zhang, Wei Meng, Jindou Fan, Bin Liu *

Tsinghua University, Beijing 100084, China

ARTICLE INFO

Article history:
Available online 25 August 2013

Keywords:
Name lookup
GPU
Name component encoding
Named data networking

ABSTRACT

Named Data Networking (NDN) aims at redesigning the current Internet: using names to identify the wanted contents instead of using IP addresses to locate the end hosts, with the goal of substantially improving the data retrieval efficiency. Different from IP routers, NDN routers forward packets by names. An NDN name is composed of a number of length-variable components, causing the name to be tens or even hundreds of characters in length. Meanwhile, NDN routing tables could be several orders of magnitude larger than the current IP routing tables. This kind of complex name constitution plus the huge-sized name table makes wire speed name lookup an extremely challenging task.

In pursuit of overcoming this challenge, we propose a Name Component Encoding (NCE) solution that assigns codes (integers) to name components. Along with an elaborate one-dimensional transition array and a local code allocation algorithm, NCE performs every node transition by a single memory access to boost lookup speed besides greatly compressing storage space. Moreover, we implement the name lookup engine on a GPU platform to exploit GPU's massive parallel processing power; furthermore, pipeline and CUDA multi-stream techniques are applied to GPU to increase lookup throughout while reducing lookup latency. Experimental results demonstrate that, under the constraint of 100 μs latency, our GPU-based name lookup engine can achieve 51.78 million searches per second on a name table containing 10 million prefixes. NCE also saves 59.57% memory cost comparing with the character trie and supports around 900K prefix insertions and 1.2 million prefix deletions per second.

1. Introduction

The current Internet, with its core ideas shaped between the 1960s and 1970s, has already been developed for around 50 years. Centered on IP addressing, the Internet was designed for conversations between communication endpoints and facilitated ubiquitous interconnectivity. The Internet was successful when providing effective, global-scale communications, which was unique and groundbreaking. However, it is now overwhelmingly used for content distribution, which has been gradually shown to be a poor match between the Internet architecture and today's dominant use – information/content-intensive retrieval and sharing. Internet usage exhibits a content-centric rather than an IP address-centric property. This kind of functionality transition from host-to-host conversation to content distribution calls for a brand new Internet architecture, and such an idea has also been observed by [1–4].

To address this transition, the clean slate Named Data Networking (NDN) [5], an instance of the Content-Centric Networking (CCN) [1] paradigm, was recently proposed for this purpose and widely regarded as a promising architecture for future networks. Quite different from the current IP-based network, this new paradigm is characterized by name-based routing and forwarding. NDN names, used to route and forward packets, have hierarchical

☆ This work is supported by NSFC (61073171), Tsinghua University Initiative Scientific Research Program (20121080068), 863 project (2013AA013502).

* Corresponding author. Tel.: +86 1062773441.
E-mail address: liub@tsinghua.edu.cn (B. Liu).

1389-1286/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comnet.2013.07.006

structure and variable-length. This content name-based longest prefix match (LPM) presents difficulties when implementing wire speed lookup: (1) ***the content names are far more complex than the IP addresses.*** Unlike the fixed-length IP addresses, the content names have variable and unbounded length which may be composed of tens, or even hundreds of characters. Thus, an NDN name is much longer than an IPv4/IPv6 address, making the individual name search a time-consuming operation; (2) ***An NDN name table could be several orders of magnitude larger than a today's IP forwarding table.*** As of August 2013, there have been 716,822,317 websites reported [6], leading to an NDN name table containing up to tens of millions of name prefixes even if we adopt a hierarchical name structure, while the current largest IP routing table has less than 500K IP prefixes. Further, given that NDN name aggregation can only happen in a unit of a component, inefficient prefix aggregation could be expected. (3) ***High update rate.*** In addition to the changes of network topology and routing policy, an NDN router has to handle frequent updates brought by content publishing and deletion, which makes name prefix updates much more frequent than IP prefix updates in today's Internet. Definitely, the designed name lookup mechanism should support fast update operations including insertion and deletion while keeping fast name lookup speed.

Facing these challenges, the goal of this paper is to explore effective solutions to achieve high lookup throughput while greatly compressing the name table's memory occupation and supporting fast name prefix updates for NDN. To reach the above goal, encoding the name components to integers is potentially a promising solution to reduce the memory occupation of a name table. Compared with the components, the codes are smaller in size and faster for searching. Obviously, an encoded name table will have a more constrictive data structure with less memory size while boosting the search speed.

Ahead of the table lookup, we need to first encode the incoming names in a real-time fashion and this will inevitably incur extra cost. Our study [7] shows that both encoding the arrival names and searching the codes against the encoded name table are computation-intensive tasks. This naturally makes us think of using Graphic Processor Unit (GPU) to speed up the name lookup operation. GPU offers extremely high thread-level parallelism with hundreds of slim cores [8,9]. Its data-parallel execution model fits nicely with parallel name lookup to achieve high throughput. However, GPU's massive parallel processing power can only be fully exploited by loading a large amount of input names in batches, and this usage will potentially lead to a longer per-packet delay because of accumulating the arrival names for batch processing. In this paper, we also strive to elaborate the GPU implementation to balance the lookup throughput and the search latency. Especially, we make the following major contributions:

1. We propose an efficient encoding method, i.e., Name Component Encoding (NCE), to assign codes (integers) to name components. Along with an elaborate one-dimensional transition array and a local code allocation algorithm, NCE performs every node transition by a single memory access to boost lookup speed, and compresses storage space. Compared with the traditional character trie, NCE can achieve a lookup speedup of 5 times, while saving around 59.57% memory space.
2. We combine the NCE scheme with GPU's massive parallel processing power to implement a GPU-based name lookup engine, in which CPU and GPU are virtually configured as the control plane and data plane of a router, respectively. The CPU is used to build specially designed data structures off-line from the NDN name table and loads them into the data plane, while GPU is responsible for encoding the incoming names on-line and searching against the name table. Besides this, the CPU executes the name prefix updates and downloads them to the device memory of the GPU incrementally to support fast insertions and deletions.
3. We take the advantage of pipeline and CUDA stream techniques to effectively reduce the name lookup latency while keeping high throughput by fully exploiting the parallel property of the GTX590 GPU. Intensive experiments show that, under the strict latency constraint of 100 μs, our GPU-based name lookup engine can achieve 51.78 MSPS throughput, equivalent to 103.56 Gbps wire speed lookup (assuming the average NDN packet size is 250 bytes).

The rest of this paper is organized as follows: the background and challenges of NDN name lookup are described in Section 2. Section 3 introduces the idea of name component encoding, as well as its algorithm and data structure. Section 4 optimizes NCE's lookup latency. Section 5 presents the experimental results. After surveying the related work in Section 6, we conclude our work in Section 7.

2. NDN name lookup: background and challenges

2.1. Packet forwarding process

There are two kinds of packets in NDN: Interest packet and Data packet [5]. As shown in Fig. 1, the forwarding processes in a router's data plane for these two kinds of packets are different. **Interest packet forwarding:** (1) When a router receives an Interest packet, it first searches the content name against the Content Store (CS) table (exact match). If hit, this means the router has stored the required content, then a Data packet with the required content is returned; (2) If not hit, the router looks up the content name against the Pending Interest Table (PIT). If matched, it means that the router has received the Interest packet(s) requesting the same content earlier, but not responded yet. Then, the information of the arrival port from which the Interest packet comes is appended to the corresponding PIT entry, and this Interest packet is discarded and would not be forwarded to upstream routers; (3) If not matched in the PIT, the router looks up the content name against the Forwarding Information Base (FIB), i.e., name table, which complies with the Longest Prefix Match (LPM). If this name matches its LPM prefix in the FIB, the router will forward this Interest packet to the corresponding next-hop port, and create an associative PIT entry;

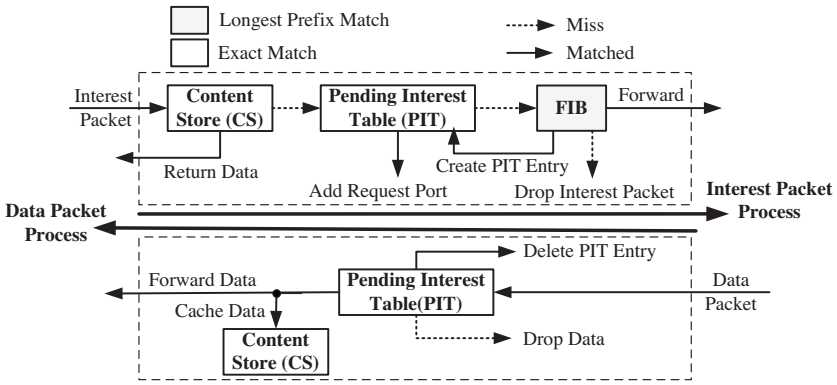


Fig. 1. NDN forwarding process.

otherwise, it will discard this Interest packet. **Data packet forwarding:** When a Data packet arrives at a router, the router looks up the content name against the PIT based on exact match [1,10]. If matched, the router will forward this Data packet to the port(s) recorded in the matched PIT entry; otherwise, it will discard this Data packet.

2.2. Name lookup challenges

Among the above forwarding processes, CS and PIT lookups are exact match, while name lookup against the FIB is Longest Prefix Match. LPM will not only complicate the name lookup, but also create the performance bottleneck of packet forwarding. In this paper, we focus on wire speed name lookup against the FIB.

In order to implement high-speed NDN packet forwarding against large-scale FIBs, an NDN name lookup engine should satisfy the following requirements.

1. *Wire speed name lookup*: Nowadays, OC-768 (40 Gbps) links are increasingly deployed in the Internet backbone, and OC-3072 (160 Gbps) technology is emerging on the horizon of the Internet. To implement NDN routing and forwarding in an increasingly large-scale, high-speed Internet, name lookup engines have to keep up with the accelerating wire speed.
2. *Memory efficiency*: An NDN router's FIB can contain as many as tens of millions of name prefixes, or even more; and each name prefix consists of tens, or even hundreds, of characters, resulting in large FIBs that have tens of gigabytes. To be practical, name lookup engines need to implement name lookup against such large-scale FIBs with reasonable memory occupation.
3. *Fast updates*: In addition to network topology changes and routing policy modifications, NDN routers have to handle one new type of FIB update — when the contents are published/deleted, the name prefixes need to be inserted into or deleted from the FIBs, which makes the updates of the FIBs much more frequent than that of today's Internet. Fast FIB update, therefore, should be well handled, especially for large-scale FIBs.
4. *Scalability*: The growth rate of NDN FIB size not only depends on the number of routers and end hosts, but also on the number of contents, which grows much faster than the number of routers and end hosts. Therefore, satisfying the above performance requirements represents a much tougher challenge than the scalability issue in the traditional IP lookup.
5. *Latency*: Generally, a high-end router should achieve low packet forwarding delay. Typically, an ISDN switch forwards a voice slot with no more than 450 μs. Based on this reasoning, we expect that the lookup latency[1] should be less than 100 μs. Thus it will be a challenge to achieve low latency while keeping the high speed throughput, especially when we implement the name lookup engine on GPU's platform.

In summary, compared with the traditional IP lookup, NDN name lookup is a much more challenging task. Practical name lookup engine design and implementation, therefore, requires substantial innovation and re-engineering.

3. Name component encoding: algorithm and data structure

This section introduces the Name Component Encoding (NCE) algorithm and its corresponding data structures. In Section 3.1, a name table is organized as a Name Component Trie (NCT), which is of the component granularity and reduces memory space substantially compared with the name character trie. Then, in Section 3.2, we encode each component in the Name Component Trie to an integer code, making the Name Component Trie into an Encoded Name Component Trie (ENCT). In Section 3.3, ENCT is further reconstructed as a one-dimensional transition array, where a transition from one node to its child node on the ENCT is reached by a single memory access, which markedly increases the lookup speed. To further reduce the memory cost of ENCT, in Section 3.4, we improve the way of assigning codes from the Global-code allocation mechanism to the Local-code allocation mechanism. Given that the coded trie solution asks for encoding the content names in real-time, in Section 3.5, we optimize the data structure of the Component Hash Table to achieve fast

[1] The lookup latency is defined as the time interval between inputting the name to and outputting the next-hop port from the search engine.

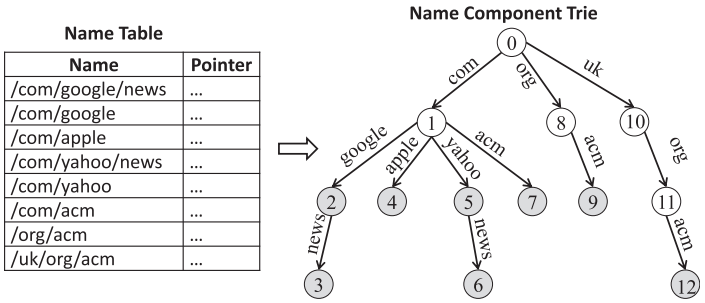


Fig. 2. Name component trie.

component-code mapping while reducing memory consumption. At last, we describe the incremental update scheme of NCE in Section 3.6.

3.1. Name Component Trie (NCT)

NDN names are of component granularity. By obeying the LPM rule, a prefix match can only happen at the end of a component. For instance, a name */com/google* can only be matched after completely matching two components: *com* and *google*.

Based on the above basic properties, we can represent the names in the FIB using a Name Component Trie, which is shown in the right part of Fig. 2. The transition condition from one node to another depends upon completely matching a component. For example, the search path for */com/google/orange* is $0 \rightarrow 1 \rightarrow 2$, where the transitions $0 \rightarrow 1$ and $1 \rightarrow 2$ correspond to match the components of *com* and *google*, respectively. Since component *orange* does not meet any transition condition, the lookup process terminates at the prefix of */com/google*.

Obviously, if we do not consider the hierarchy of names and treat them as flat strings, names can be represented traditionally by a character trie, here we name it as Traditional Character Trie (TCT). Compared with TCT, NCT not only reduces the number of nodes to save memory space, but also decreases the depth of the trie to accelerate lookup speed. For the name table illustrated in Fig. 2, there are only 13 nodes totally in NCT with a depth of 3, while TCT contains 56 nodes with a depth of 13.

3.2. Basic name component encoding

Compared with TCT, NCT notably reduces the number of nodes and the depth of the trie. However, a transition is matched only when a whole component is matched. In fact, this kind of complicated work can be released by encoding the components. In NCT, the edges (transitions[2]) leaving a node represent the variable-length components. Suppose that all the transitions leaving from the same node form a transition set. If a transition set is organized as a list, the matching process will traverse the list to find the matched component, and the time complexity will be $O(NW)$, where N is the length of the list and W is the average number of characters over all the components in the transition set. If we adopt a hash table to store the transition set, the time complexity of matching components can be improved to $O(W*(1+\alpha/2))$ [11] when chaining is used to resolve conflicts, but the memory space will be increased by a factor of $1/\alpha$, where α is the load factor of the hash table.

Instead, we propose to assign a code to each component, and the transitions in the NCT are then turned to be represented by the codes. We name this solution Name Component Encoding. NCE first employs a hash table to keep track of all the actual components in the NCT, and assigns each component a code according to our code allocation algorithms which are depicted in Section 3.4. Then NCE transforms the NCT to the Encoded Name Component Trie to increase lookup speed and reduce memory consumption. For instance, for the NCT in Fig. 2, after encoding, the corresponding hash table and ENCT are illustrated in Fig. 3.

When looking up a name, say */com/google/orange*, we first decompose the name into three components: *com, google, orange*; then we encode them by looking up each component for a code in the hash table. In this example, the encoding results are: *com* = 3, *google* = 6 and *orange* = −1 (−1 means that *orange* does not match any component in the hash table). At last, the code string */3/6* is used to look them up against the ENCT. Finally, the matching process terminates at node 2, and a pointer to this FIB entry is returned.

3.3. Transition array for ENCT

ENCT can further be improved by the Encode Transition Array (ETA), which directly stores all the transitions of an ENCT in a one-dimensional transition array, and the node information is implied in the ETA element as well. The ETA markedly improves lookup speed, since it enables a transition on the ENCT to be implemented by only a single memory access.

Each element in the ETA has two fields: {*code, offset*}. A *code* is an integer that is assigned to the corresponding component; and an *offset* stands for the address of the ETA. As illustrated in the left part of Fig. 3, ETA[0]–ETA[4] store the transitions of Node_0. The offset of ETA[3], which is 5, indicates that the transitions of the next node (Node_1 in this case) are stored beginning at ETA[5]. Formally, one specific transition T_{ij} (the edge from Node N_i to Node N_j) of Node N_i is stored in ETA at the location $offset_i + code_{ij}$,

[2] In this paper, *edge* and *transition* have the same meaning, and we use it according to the context.

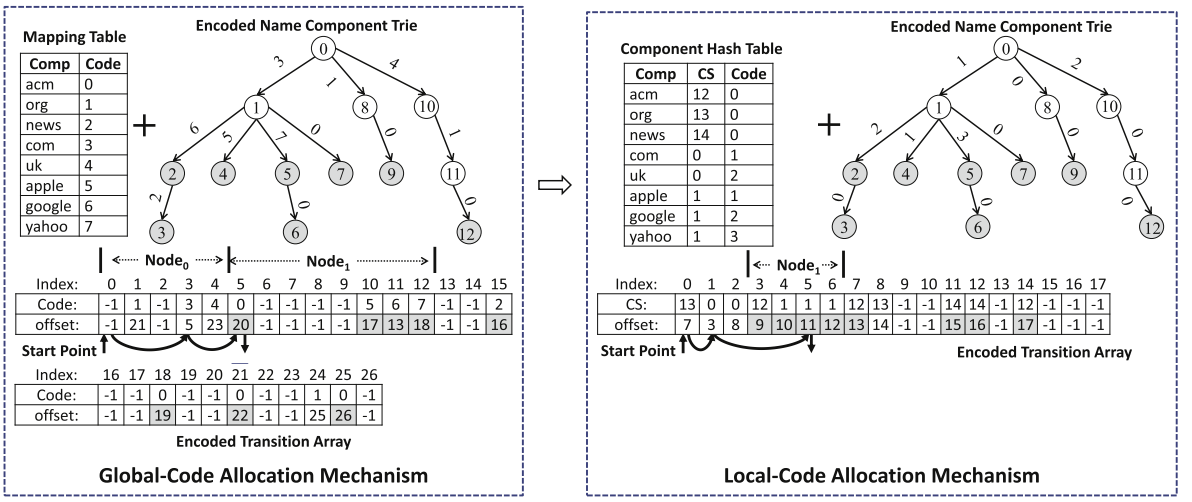


Fig. 3. Name component encoding solution.

where $offset_i$ is the offset (relative to the first element of ETA) of $Node_i$, and $code_{ij}$ represents the code value of T_{ij}.

Here, the ETA needs to store the code values to ensure the correctness. For example, ETA[5] in the left part of Fig. 3 stores a transition (with its code = 0) of $node_1$. If the ETA has no code information, a transition of $node_0$ ($offset_0 = 0$) with a code = 5 will point to the 5th position in the ETA, which will cause an error. This problem can be formalized as if $a + b = 5 \Rightarrow a = ?,\ b = ?$. Obviously, there is more than one answer. Consider a more illustrative example: */com/acm* is encoded to */3/0*, and */apple* is encoded to */5*. Their search paths are $0 \rightarrow 3 \rightarrow 5$ and $0 \rightarrow 5$, respectively. At last, */com/acm* and */apple* both reach the same node $Node_7$. Consequently, an error occurs. With the help of code information, such an error can be avoided. The corresponding search path of */com/acm* is $0 \rightarrow 3 \rightarrow 5$ remains unchanged, while the search path of */apple* is $0 \rightarrow NULL$.

3.4. Code allocation algorithm

The coding methodology directly influences the size of the ETA. Therefore, we design the Local-code allocation mechanism to reduce the memory consumption. To clarify our encoding mechanism, we first present the following three definitions.

Definition 1. A component (edge) C_i **belongs to** a node N_j when C_i leaves N_j for another node in the trie.

Definition 2. Original Collision Set (OCS) is a set of components, and all the components **belong to** a given Node N_j. Different components in an OCS should be encoded with different codes to avoid collision.

Definition 3. If a component **belongs to** multiple OCSes, these OCSes form a new set, called **Collision Set** (CS), i.e., each OCS is an element of CS, and CS is a set of set.

3.4.1. Global-code allocation mechanism

The simplest approach, Global-code allocation mechanism (global encoding for short), assigns each distinct component a unique code, i.e., a component and a code has a one-to-one mapping relationship. We can conclude that: $C_i \neq C_j (i \neq j) \Rightarrow E_i \neq E_j$, here E_i is the code allocated to C_i. The Global-code allocation mechanism will cause ETA's memory space expansion. For example, in the left part of Fig. 3, the $Node_2$'s *offset* is 13, and its transitions occupy ETA[13]–ETA[15]. However, $Node_2$ only has one transition and is stored in ETA[15], thus 66.7% memory is wasted. Global encoding, though easy to implement, requires $O(N_c^2)$ space complexity (N_c is the number of distinct components). Therefore, global encoding is insufficient to support large-scale FIBs.

3.4.2. Local-code allocation mechanism

To address the problem of large memory consumption of global encoding, we propose the Local-code allocation mechanism (local encoding for short). As mentioned above, an OCS is a set of all the transitions leaving from the same node. Hereby, the transitions belonging to an OCS should be assigned different codes; while the different components belonging to the different OCSes can share the same code. In other words, $C_{ij} \in OCS_i$, $C_{ik} \in OCS_i$, $j \neq k \Rightarrow E_{ij} \neq E_{ik}$. If $C_{ij} \in OCS_i$, $C_{km} \in OCS_k$, $C_{ij} \neq C_{km}$, $i \neq k$, then E_{ij} is allowed to be equal to E_{km}. For instance, in the right part of Fig. 3, *com* belongs to OCS_0 and *apple* belongs to OCS_1. They can both be encoded as 1, while still guaranteeing the correctness.

Local encoding is equivalent to the function f: $\{OCS_i - \times C_{ij} \rightarrow E_{ij}\}$, where $\times$ means Cartesian product. f implies that with both the node number and the component information, the codes can be correctly assigned to the components. However, the name lookup engine can only get the components of a content name, rather than the lookup state (node) information. Therefore, the code of C_i should be consistent with multiple OCSes to support encoding only based on the component itself without the node information. For

example, in the right part of Fig. 3, *acm* is assigned the same code 0 in all the OCSes: OCS_1, OCS_8 and OCS_{11}.

Assigning a code to a component is not an easy task. In fact, finding an encoding scheme that minimizes the memory requirement of the ETA is an NP hard problem. Therefore, we utilize the following greedy algorithm to implement the local encoding:

- Step 1, we build the NCT based on the name table.
- Step 2, we traverse the NCT to build a hash table for all the distinct components, and keep track of the information to which node a component belongs (i.e., belongs to which OCS).
- Step 3, we transform the component hash table into a component array which is sorted in a descending order based on the number of OCSes that a component C_i belongs to. If the numbers of OCSes are equivalent, the component array is sorted in an ascending order based on the number of transitions that the OCS contains.
- Step 4, we encode the component C_i in the component array from $i = 0$ by selecting the smallest code as the code of C_i among the OCSes that C_i belongs to.

3.4.3. False positive

Given that the different components C_i and C_j can share the same code, i.e., $E_i = E_j$, merely using code is inadequate to prevent false positives. For example, in the right part of Fig. 3, */com/google/news* and */com/google/acm* are encoded as the same code sequence */1/2/0*. In this way, */com/google/acm* is matched with the prefix */com/google/news*, while the correct one is */com/google*.

In order to avoid this kind of error, we revise the information that an ETA element contains from {*Code, offset*} to {*CS, offset*}, where CS (refer to Definition 3) field stores the ID of the CS that this component belongs to. If C_i only belongs to one node (OCS), we set CS_i=OCS_i; otherwise, $CS_i = \max\{\max\{CS_j\},\max\{OCS_k\}\} + 1$. Meanwhile, the original Mapping Table entry {component → Code} is revised to be {component → ⟨CS, Code⟩}. In this way, the encoding result of */com/google/acm* is */⟨0, 1⟩/⟨1, 2⟩/⟨12, 0⟩*, which has the search path: */⟨0, 1⟩/⟨1, 2⟩*. Therefore, the correct longest prefix */com/google* is found.

The following two Theorems prove that the Local-code allocation mechanism keeps the correctness of longest prefix match.

Theorem 1. *The Local-code allocation mechanism, along with the ETA that implements the ENCT, preserves the trie property that a child node can only be directly reached by its parent node, if the transfer condition holds.*

Proof 1. Let O_i be the offset of $Node_i$, the component C_i's corresponding code and collision set are E_i and CS_i, respectively. C_i matches the transition leaving from $Node_i$ to $Node_{i+1}$. Assume that $C_i \neq C_j$ and $Node_{i+1} = Node_{j+1}$, according to the Local-code allocation mechanism, we have $O_{i+1} = O_{j+1}$, $CS_i = CS_j$, $O_{i+1} = O_i + E_i$, $O_{j+1} = O_j + E_j$. If $E_i \neq E_j \Rightarrow O_i \neq O_j \Rightarrow CS_i \neq CS_j$, contradicts with $CS_i = CS_j$, therefore $E_i = E_j$. If $E_i = E_j \Rightarrow O_i = O_j$ and $C_i = C_j \Rightarrow Node_i = Node_j$. □

Theorem 2. The Local-code allocation mechanism keeps the correctness of longest prefix match.

Proof 2. Suppose that two names $A = C_{a1}, \ldots, C_{ai}, \ldots, C_{am}$ and $B = C_{b1}, \ldots, C_{bj}, \ldots, C_{bm'}$, their prefixes are $Prefix_A = C_{a1}, \ldots, C_{ai}$ and $Prefix_B = C_{b1}, \ldots, C_{bj}$, respectively. Moreover, $Prefix_A \neq Prefix_B$, but A and B match the same prefix $O_{ai+1} = \text{-}O_{bj+1}$. Suppose two names A and B are encoded to two code sequences $E_{a1}, \ldots, E_{ai}, \ldots, E_{am}$ and $E_{b1}, \ldots, E_{bj}, \ldots, E_{bm'}$. The search paths of prefix A and B are $O_{a1}, \ldots, O_{ai}O_{ai+1}$ and $O_{b1}, \ldots, O_{bj}O_{bj+1}$, respectively. Because $O_{ai+1} = O_{bj+1}$, based on Theorem 1, we arrive at $O_{ai} = O_{bj}$ and $C_{ai} = C_{bj} \Rightarrow O_{a1} = \text{-}O_{b1}$ and $i = j \Rightarrow O_{ak} = O_{bk}, C_{ak} = C_{bk}, 0 \leqslant k \leqslant i$, i.e., $Prefix_A = Prefix_B$, contradicts with $Prefix_A \neq Prefix_B$. □

3.5. Component hash table

In the NCE scheme, the name lookup engine first decomposes the content name into components and encodes the components into codes; then it looks up the codes against ENCT to find the LPM prefix and forwarding port. The components encoding process and the codes lookup process both contribute search time to the name lookup process. The ETA and the Local-code allocation mechanism effectively accelerate lookup speed and decrease memory space.

Here, we adopt a hash table to store the mapping table to speed up the encoding operations. Though multiple components in a name can be encoded in parallel, the encoding speed of a single component directly influences the throughput of the entire lookup engine. To achieve fast mapping, the hash value of a component is calculated by the BKDR [12] hash function, and the hash conflicts are resolved by a chaining, as illustrated in Fig. 4. The Component Hash Table (CHT), which maps a component to a ⟨CS, code⟩ pair, is composed of two parts: the index table and the content table. The index table stores the index of a component's profile, which is stored in the content table. A content table entry has five fields: {*CS, code, length, component, next*}. *CS* and *code* have been introduced above; *length* represents the number of characters of the component, and the following {(*length* − 1)/4 + 1} integers store the component (an integer is 4 bytes); *next* is a pointer that stands for the next position in the content table for the different components which have the same hash value.

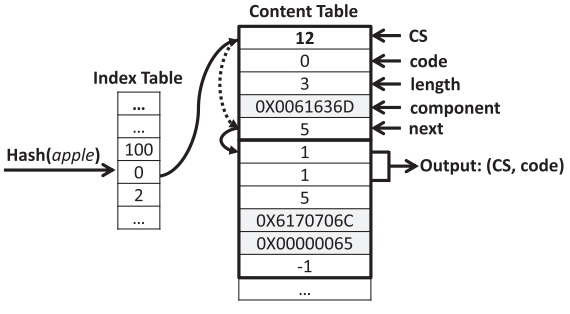


Fig. 4. Hash table data structure: component hash table.

Fig. 4 illustrates the lookup process of *apple*: (1) compute the hash value of *apple* as the address of the index table, and access that entry to obtain the address (of the content in this example Table); (2) access the content table from the address of 0. The length of *apple* is 5, while the length of the component stored in the current position is 3 — they do not match; then we jump to position 5 (indicated by the *next* pointer), *apple* matches the component in current position; finally the output is $CS = 1$, $Code = 1$.

The data structure of a hash table brings the following benefits:

1. The separation between the index table and the content table enables us to allocate a relatively large index table to reduce collision rate, while costing a small storage overhead. Moreover, we can re-construct the index table to dynamically re-construct the hash table based on the load factor, without modifying the content table.
2. The content table can compactly store the variable-length keys (components).
3. The hash table is represented by arrays, which are easy to implement on the GPU platform.

3.6. Update

There are two types of name prefix updates: deletion and insertion, which are both easy to handle with our design. We first update the NCT in the control plane (CPU); and then incrementally deliver the updating results to the ETA in the data plane (GPU).

3.6.1. Delete

Suppose that we want to delete a prefix $C_1, \ldots, C_j, \ldots, C_k$. The steps are as follows: (1) First, we find this prefix in the NCT and record the components (supposed to be $C_j, \ldots, C_k$) and nodes which need to be deleted; (2) Second, we search the components in the CHT to obtain the code string $E_1, \ldots, E_j, \ldots, E_k$ of the components $C_1, \ldots, C_j, \ldots, C_k$; (3) Third, we delete the corresponding transitions in the ETA, while deleting the edges and nodes in the NCT. If a component C_m no longer exists in the NCT after deletion, we delete the corresponding entries of C_m in the CHT.

3.6.2. Insert

Suppose prefix $C_1, \ldots, C_j, \ldots, C_k$ is inserted into the name table. The operations are as follows: (1) First, we insert the prefix into the NCT. Suppose that the added components are $C_j, \ldots, C_k$; (2) Second, we insert $C_m (j \leqslant m \leqslant k)$ into the CHT. If the CHT does not store these components, an entry for C_m is added in the CHT. Otherwise, we fetch the corresponding code E_m of C_m, and judge whether this E_m conflicts with the existing codes in the OCS to which C_m belongs. If not, we do nothing; otherwise, all the influenced nodes will be re-encoded; (3) Third, all the $E_m (j \leqslant m \leqslant k)$ are inserted into the ETA. If the entries are occupied, we move all the transitions belonging to this node to a new position, and update the offset value of its parent node; otherwise, we store the E_m in the available entries.

4. Implementation

In this section, we present the implementation details of our GPU-based name lookup engine, as well as the optimization strategies used to boost the lookup performance, e.g. throughput and lookup latency. First in Section 4.1, we introduce the hardware platform, operating system and development tools with which we implement our name lookup engine. Then in Section 4.2, we give the framework of our GPU-based name lookup engine. Finally, in Section 4.3, we describe the workflow of our name lookup engine.

4.1. Platform, environment and tools

Our GPU-based name lookup engine is implemented on a commodity PC with an NVIDIA Geforce GTX590 GPU board which contains two GPUs. In our current implementation, we only use one of them. The PC we used is equipped with a 6-core CPU (Intel Xeon E5645x2), with 2.4 GHz clock frequency. The GPU communicates with the CPU via a PCI Express ($\times 16$) bus. The relevant hardware configuration is listed in Table 1.

We run the 2.6.41.9-1.c15.x86_64 version of Linux Operating System (Fedora release 15) on the CPU of this PC, and the x86_64-285.05.09 NVIDIA-Linux operating system version of CUDA on the GPU.

The entire name lookup system is written in about 8500 lines of code, including the code written in C/C++ for the CPU part and the code written in CUDA C for the GPU part. The CPU part is responsible for preparation and preprocessing of the name table, while the GPU part handles the real per-packet name lookup, which is developed using NVIDIA CUDA SDK 4.0.

4.2. Overall structure

As shown in Fig. 5, our GPU name lookup engine consists of three major functional sub-systems: the *NCE Builder* and *Test Bench* residing in the CPU side, and the high-performance *Kernel Search Engine* responsible for all the GPU-related tasks.

The *NCE Builder* serves as the control-plane of the name lookup system. It takes a name table as its input, and outputs two tables used to perform the fast lookup on the GPU. One table is the CHT, which is essentially a hash table in which the encoding information of name components is grouped; the other one is the ETA, which is a compact one-dimensional table recording all the transitions in the ENCT.

The *Kernel Search Engine* loads the two control tables (CHT and ETA) generated by the Builder to device memory. Then it takes the name traces in the *Test Bench* as input, and encodes each name in the name traces into codes,

Table 1
Hardware configuration.

Item	Specification
CPU	Intel Xeon E5645 × 2 (6 cores, 2.4 GHz)
GPU	NVIDIA GTX590 (2 × 512 cores, 2 × 1536 MB)
Motherboard	ASUS Z8PE-D12X (INTEL S5520)
RAM	DDR3 ECC 48 GB (1333 MHz)

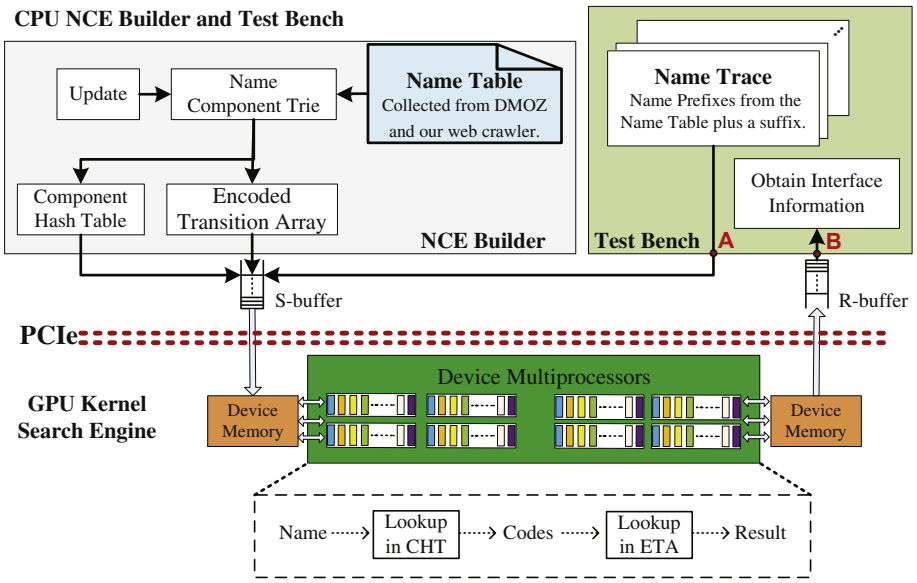


Fig. 5. Framework of the name lookup system.

searches the codes against the ETA and returns the lookup result. Take a closer look at the *Kernel Search Engine*: there are hundreds or even thousands treads running the same instruction concurrently on the GPU. Each thread deals with one specific name lookup request, translates the name into codes and searches the codes against the transition array. The lookup results are temporarily stored in the global memory of GPU, and returned to one module running on the CPU. The module will obtain the next-hop information based on the GPU's output.

In addition to the *NCE Builder*, there is also a daemon managing the update of the name table and the corresponding CHT and ETA.

Besides this, an off-line name trace generator also serves the whole system as the synthesized traffic provider that generates the name traces used to test the functionality and performance of our GPU name lookup engine.

4.3. Workflow

Fig. 5 also presents the basic workflow in our GPU encoded name lookup system. Performing name lookup in a per-packet manner is expensive because this will introduce significant overhead caused by PCIe transaction, as well as under-utilization of GPU resource. In order to achieve high throughput and take full use of GPU, we group packets in a batch manner. All the computations related to name lookup are done at the granularity of a batch, not a single packet.

We divide the name lookup workflow into three steps: pre-processing, search-processing, and post-processing. Pre-processing and post-processing run on the CPU side while search-processing does GPU-related tasks. Each step works as follows:

4.3.1. Pre-processing

During this process, the CPU reads names from the input name traces and stores them in the S-buffer (as shown in Fig. 5). At the same time, names are stored in one row in the memory if there is still available space in the current row; otherwise they are stored in the next row. A special number (e.g. 0x00000000) is used as the delimiter of names in the same row. A number of rows are organized as a batch. Once the CPU has read in enough names or the GPU is ready for the next batch of names, the system runs to the search-processing step and starts the next round of pre-processing at the CPU side again.

4.3.2. Search-processing

All GPU related tasks are finished in this process. At first, pre-processed names, as input names in the S-buffer, will be sent to the GPU's global memory via PCIe bus. As names are already grouped in rows, each thread in GPU will take one row of input names for processing. Each component is hashed by the BKDR [12] hash function so that we can address its code in the CHT. Once the thread has fetched the name's codes, it looks up the codes in the ETA till the end or a failure. If a failure of code matching is detected, the search of the current name is terminated and the next name in its assigned row will be handled. The search result of each name is recorded in the global memory. After all threads have finished their tasks, the results are sent back to the R-buffer in the CPU side via PCIe bus.

4.3.3. Post-processing

Once the CPU gets the results from the GPU, it will translate the results into port numbers and sends out packets with synthetic contents to the corresponding ports.

4.4. Streaming

By dividing jobs into smaller batches, the latency will be reduced. Inspired by this finding, we have found another approach to dividing jobs, after carefully reviewing the architecture of NVIDIA GPU and CUDA. In addition to

thread-level parallelism, CUDA also supports task-level parallelism, which is achieved by exploiting the CUDA streams. With the introduction of the CUDA streams, we successfully reduce the per-packet lookup latency, while keeping high lookup throughput.

4.4.1. CUDA stream

According to our test, we find that besides the searching step which is indeed kernel execution, the transmission of names and their lookup results via PCIe bus also contributes to the per-packet latency. While waiting for names to be transferred to the GPU's global memory and results to be written back to the CPU's memory, the GPU's computation resource is under-utilized. If we could run the kernel while the data is transferred via PCIe bus, the latency caused by PCIe transmission could be effectively hidden, resulting in reduced per-packet lookup latency and increased throughput.[3] In order to realize this objective, we need the help of the CUDA streams.

A CUDA stream represents an operation queue (a task queue) in the GPU. Operations in the same CUDA stream are executed in the order that they are inserted into the operation queue. The CUDA streams could execute concurrently under the permission of hardware. For example, devices with compute capability of 1.1 or above support device overlap, which means during the execution of kernel, data could also be copied between the page-locked host memory and the device memory. (GTX590 has a compute capability of 2.0.) Devices with only one copy engine could only support one-directional data copy while devices with two copy engines support bi-directional data copy.

4.4.2. Stream scheduling algorithm

Algorithm 1 (*Stream Scheduling Algorithm*).

```
1:  procedure Stream Scheduling
2:    i ← 1;
3:    offset ← i*data_size/N;
4:    NameFetch (offset, streams[i]);
5:    Kernel (offset, streams[i]);
6:    for i ← 2 to N do
7:      offset ← i*data_size/N;
8:      NameFetch (offset, streams[i]);
9:      Kernel (offset, streams[i]);
10:     wb_offset (i − 1)*data_size/N;
11:     WriteBack (wb_offset, streams[i-1]);
12:   end for
13:   WriteBack (wb_offset, streams[i-1]);
14: end procedure
```

From the view of a GPU programmer, the operation queues are ordered as the CUDA streams. However, the practical view is totally different. There is no stream in the hardware, but only two parallel operation engines: the kernel execution engine and the memory copy engine. The operations managed by the same engine are scheduled in the order that they are inserted into the engine's operation queue. However, the dependency of operations belonging to the same CUDA stream is still kept. Thus, we need to very carefully use the CUDA streams. Some algorithms appearing to be functional are likely to cause serialized execution of streams.

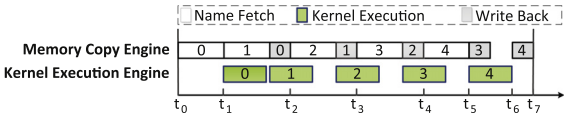


Fig. 6. An example of CUDA stream scheduling.

We design the Algorithm 1 to schedule our lookup tasks, where *NameFetch*, *Kernel* and *WriteBack* represent name copy from CPU to GPU, kernel execution, and result writing back from GPU to CPU, respectively. Here, *N* is the number of CUDA streams used for optimization.

As shown in Fig. 6, the *Kernel* task of stream 2 runs (on the kernel execution engine) in parallel with the *WriteBack* task of stream 1 followed by the *NameFetch* task of stream 3 (both running on the memory copy engine).

5. Evaluation

5.1. Experiments setup

5.1.1. Name tables

We collect two name tables for performance measurement, which include 2,763,780 entries and 10,000,000 entries, respectively. For brevity, we refer to them as the "3M name table" and the "10M name table", respectively. Each name table entry is composed of an NDN-style name and a next-hop port number.

5.1.2. Name traces

We generate name traces from name tables to simulate the content names carried in NDN packets. Each name trace is formed by concatenating a name prefix selected from the name table and a randomly generated suffix. We generate two types of name traces, simulating average lookup workload and heavy lookup workload, respectively. Each name trace contains 200M names.

The average workload trace is generated by randomly choosing names from the name table, while the heavy work load trace is generated by randomly choosing from the top 10% longest names in the name table. Intuitively, the longer the input names are, the more state transition operations the GPU will perform for their lookups, meaning heavier workload.

5.2. Experimental results

For the GPU-based name lookup engine, we measure the performance of the TCT as the base reference, and compare this performance with that of the NCE proposed in Section 3. In TCT, the transitions of each node are sorted by the values of the characters to support binary search.

In Section 5.2.1 and Section 5.2.2, we evaluate the lookup throughput, lookup latency and memory requirement

[3] The pipeline reduces the queuing time for each name and therefore the overall processing time for each name is reduced accordingly.

of the TCT and NCE methods for comparison. The scalability of our lookup engine design is evaluated in Section 5.2.3. Finally, in Section 5.2.4, we evaluate the performance of NCE on handling name prefix updates.

5.2.1. Throughput and latency

Firstly, we measure the throughput and latency of the TCT and NCE approaches on the 3M and 10M name tables. For each method, the experiments are conducted under different parameters: doubling block number of a grid from 8 to 4096, doubling thread number of a block from 32 to 1024, doubling stream number from 1 to 4096. The experimental results are shown in Fig. 7–10. For brevity, we only show the statistics with latency less than 1000 μs.

From Fig. 7 and Fig. 8, we can derive that with a latency of 100 μs, by NCE, the lookup performance under average workload and heavy workload of the 3M name table can achieve 51.78 MSPS and 47.82 MSPS, respectively, while TCT cannot even satisfy the requirement of 100 μs latency. With a latency of 200 μs, TCT can arrive at lookup throughput of 12.16 MSPS and 11.98 MSPS, under average and heavy workload, respectively; while NCE can achieve 59.33 MSPS and 53.79 MSPS, under average and heavy workload, respectively. The above results demonstrate that, NCE has a five times speedup over TCT, as well as better latency performance.

For the larger 10M name table, we obtain similar lookup and latency performance. Assuming that the average packet length is 250 bytes, with the latency required to be 100 μs, NCE achieves 103.56 Gbps and 95.64 Gbps throughput under average and heavy workload, respectively.

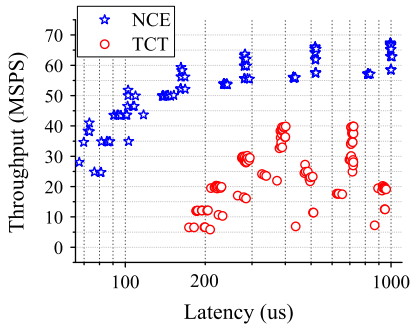


Fig. 7. Throughput and latency on 3M name table (average workload, $\alpha = 0.5$).

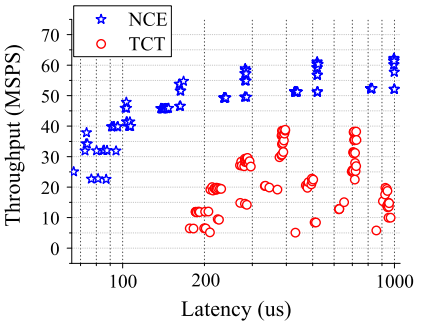


Fig. 8. Throughput and latency on 3M name table (heavy workload, $\alpha = 0.5$).

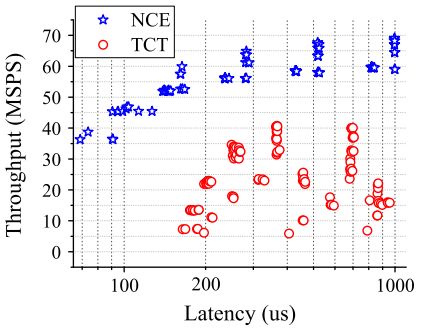


Fig. 9. Throughput and latency on 10M name table (average workload, $\alpha = 0.5$).

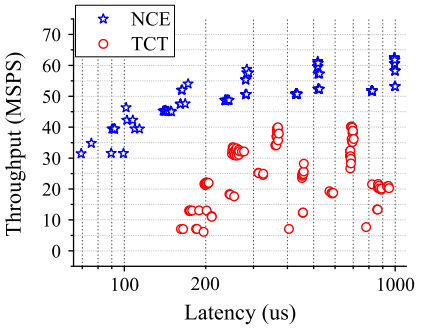


Fig. 10. Throughput and latency on 10M name table (heavy workload, $\alpha = 0.5$).

5.2.2. Memory space

We also measure the memory cost of these methods for comparison, based on the 3M and 10M name tables.

Table 2 shows the memory consumption of TCT and NCE. For the 3M name table, the CHT and ETA of NCE require 60.14 MBytes and 60.72 MBytes respectively, and the total memory consumption is 120.86 MBytes. Compared with the memory consumption (370.87 MBytes) of TCT, NCE reduces memory cost by 67.41%. For the 10M name table, the CHT and ETA require 186.16 MBytes and 343.47 MBytes, 529.63 MBytes in total, while TCT consumes 1310.02 MBytes. 59.57% memory reduction is achieved by NCE. Therefore, our proposed methods reduce storage space effectively over TCT.

5.2.3. Scalability

While our GPU-based name lookup engine demonstrates good performance on the 3M and 10M name tables, we are also interested in foreseeing its performance trend as the name table size grows. For that, we partition each name table into ten equal-sized subsets, and progressively generate ten name tables for each of them; the kth generated name table consists of the first k equal-sized subsets. Experiments are then conducted on these 20 generated name tables derived from the 3M name table and 10M name table. Experimental results on lookup throughput, memory requirement and lookup latency are presented in Figs. 11 and 12, respectively.

As the name table size grows, lookup throughput and lookup latency tend to stabilize around 50 MSPS and

Table 2
Comparison of NCT and NCE's processing performance.

Name table	Average name length (Byte)	TCT size (MByte)	Component number	Distinct component number	NCE (MByte)			Compression ratio (TCT vs. NCE)
					CHT size ($\alpha = 0.5$)	ETA size	Total size	
3M	21.27	370.87	6,223,396	2,505,706	60.14	60.72	120.86	67.41%
10M	24.48	1310.02	12,637,894	7,756,762	186.16	343.47	529.63	59.57%

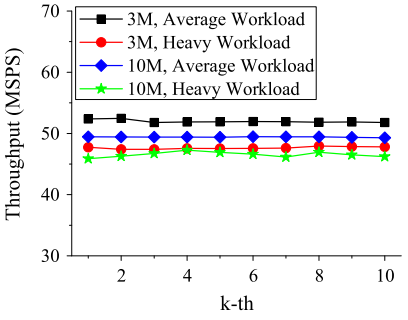


Fig. 11. Growth trend of lookup speed ($\alpha = 0.5$).

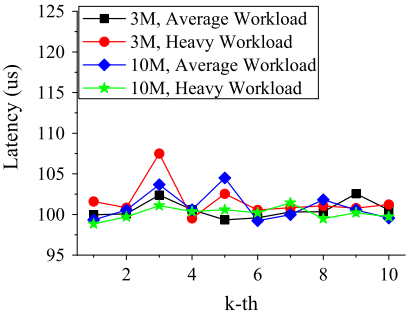


Fig. 12. Growth trend of lookup latency ($\alpha = 0.5$).

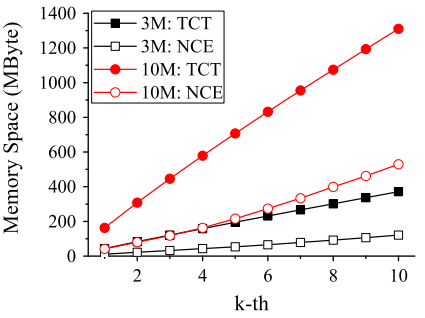


Fig. 13. Growth trend of memory size ($\alpha = 0.5$).

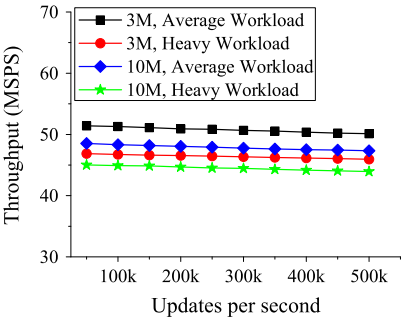


Fig. 14. The lookup throughput with different update rates.

100 μs, respectively. Fig. 13 further depicts the memory consumption under different name table scales. Regarding the name table scales, the slope of NCE is smaller than that of the TCT, which makes NCE more scalable to large-scale name tables.

5.2.4. Name table update

Finally, we measure the performance of NCE on handling name prefix updates, both insertions and deletions. In general, our design performs well on handling name table updates. On both name tables, we can consistently handle more than 900K insertions per second. As we have described in Section 3, deletions are much easier to implement than insertions; we can steadily handle around 1.2M deletions per second.

Fig. 14 further demonstrates the name lookup speed with different update rates (50% insertions and 50% deletions). As the update rate grows, lookup throughput tends to decrease slowly and stabilize around 48 MSPS on the 10M name table under average workload. In our system, the updates are handled in the host memory (*NCE Builder*) firstly; and then the modified parts of a FIB are loaded into the device memory to update the name lookup engine. Only the memory copy operation of update process competes with the name lookup process for the copy engine; therefore, the name lookup speed is only slightly affected by the update operations.

6. Related work

Many recent research efforts [1,5,13–16] have figured out that we should move the Internet away from its current reliance on purely point-to-point primitives. Therefore, they have proposed pioneering detailed designs that make the Internet more data-oriented or content-centric. Meanwhile, routing and forwarding based on hierarchical names have been studied in recent research [17–20], revealing the feasibility of using name instead of IP address from the viewpoint of working principle under small name table sizes. However, longest prefix match for hierarchical names can potentially slow down the lookup process. Thus, many software-based or hardware-based solutions are proposed to improve name lookup performance.

Software-based solutions: Hash-based methods have been intensively studied in order to quickly find the matched prefix by leveraging the exact match ability of hash tables. Z. Genova et al. [21] hash URLs to fixed-length signatures, and look up the signature in the hash table. This method regards the URL as a whole entity and cannot support longest prefix match. B. Michel et al. designed an incremental hash function called Hash Chain [22] to aggregate URLs sharing common prefixes. The logical data structure of Hash Chain is similar to our name component trie, which effectively reduces the memory consumption. Further, Zhou et al. [23] use a CRC-32 hash function to compress URL components, and utilize a multi-string matching algorithm for URL lookup. However, these hash-based algorithms all have a drawback – false positives due to hash collisions. Without additional remedies, any possibility of false positive will undermine the integrity of the fundamental function of packet forwarding in NDN routers. In Connectivity Server [24], all the URLs are sorted lexicographically and stored as a delta-encoded text file. Each entry only stores the different parts (delta) between the current URL and previous one. This scheme especially reduces the storage requirement significantly when the adjacent URLs share the same common prefix as long as possible. However, this scheme cannot support fast incremental update and thus fails to handle the frequent updates in NDN.

Hardware-based solutions: TCAM is well-known for its fast speed. A TCAM-based name lookup mechanism in NDN proposal [5] is presented. Unfortunately, from the cost point of view, it can hardly be a practical option, given its low memory density, high price and excessive power consumption.

Recently, GPU as a powerful and programmable parallel processing platform has attracted intensive interests in high speed packet processing: IP lookup [25,26], packet classification [27,28] and pattern matching [29–31]. Name lookup is more complex than IP lookup or packet classification. Essentially, name lookup is a pattern matching problem, in which throughput, not latency, is a key criterion. Therefore, previous research on GPU-based packet processing schemes do not provide a complete solution for reference.

We first proposed the component encoding mechanism in our previous work [7], in which the lookup speed can only reach 1.5 MSPS because it is limited by its imperfect component encoding algorithm and CPU-based implementation. In this work, we propose a Local-code allocation mechanism to reduce the memory space and boost the lookup speed. Meanwhile, we leverage the GPU to accelerate NDN name lookup by addressing all these performance issues, which include high-speed lookup and memory efficiency, as well as small latency and fast incremental update simultaneously.

7. Conclusions

NDN/CCN as an emerging technology is currently receiving a lot of attention; a broad range of technical topics are open for innovative research, and have yet to be standardized. Among them, NDN name lookup is one of the fundamental functions that promote the actual deployment of NDN. However, the questions of wire speed name lookup, memory cost, update support with off-the-shelf technologies under extremely large name tables remain to be solved. In this paper, we propose a Name Component Encoding solution that implements a GPU-accelerated name lookup engine with a number of innovative techniques. In NCE, each component is assigned a code to improve the name lookup performance, as well as reduce the memory consumption required by the NDN name table. Extensive experiments on real name tables collected from the Internet, using one chip of GTX590, demonstrate that NCE can achieve up to 51.78 MSPS on a name table containing 10 millions of name prefixes while keeping the latency less than 100 μs and supporting almost 1M updates per second.

Acknowledgments

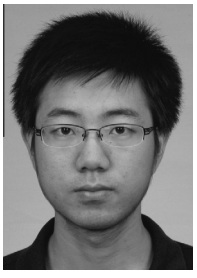
We thank Harry Rudin (the Editor in Chief), the anonymous reviewers and the guest editors of this special issue on ICN computer networks — Yanghee Choi, Andrea Detti, Mario Gerla, Diego Perino — for their help and invaluable comments.

References

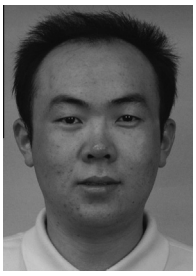
[1] V. Jacobson, D.K. Smetters, J.D. Thornton, M. Plass, N. Briggs, R. Braynard, Networking named content, in: Proc. of ACM CoNEXT, 2009.
[2] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, M. Walfish, A layered naming architecture for the internet, in: Proc. of ACM SIGCOMM, 2004.
[3] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, J. Lilley, The design and implementation of an intentional naming system, in: Proc. of ACM SOSP'99, 1999.
[4] M. Gritter, D.R. Cheriton, An architecture for content routing support in the internet, in: Proc. of USENIX USITS'01, 2001.
[5] L. Zhang, D. Estrin, V. Jacobson, B. Zhang, Named Data Networking (NDN) Project, in: Technical Report, NDN-0001, 2010.
[6] http://news.netcraft.com/archives/category/web-server-survey/.
[7] W. Yi, H. Keqiang, D. Huichen, M. Wei, J. Junchen, L. Bin, C. Yan, Scalable name lookup in ndn using effective name component encoding, in: Proc. of IEEE ICDCS'12, 2012.
[8] K. Fatahalian, M. Houston, A closer look at gpus, Communications of the ACM 51 (2008) 50–57.
[9] K.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÂijger, A.E. Lefohn, T.J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (2007) 80–113.
[10] V. Jacobson, D.K. Smetters, N.H. Briggs, M.F. Plass, P. Stewart, J.D. Thornton, R.L. Braynard, Voccn: voice-over content-centric networks, in: Proceedings of the 2009 Workshop on Re-architecting the Internet, ReArch '09, ACM, New York, NY, USA, 2009, pp. 1–6.
[11] D.E. Knuth, Art of Computer Programming, volume 1/Fundamental Algorithms; volume 3/Sorting and Searching, Addison-Wesley, 1973.
[12] http://www.partow.net/programming/hashfunctions/#BKDRHashFunction.
[13] D. Cheriton, M. Gritter, Triad: a new next-generation internet architecture, 2000, <http://www-dsg.stanford.edu/triad>.
[14] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K.H. Kim, S. Shenker, I. Stoica, A data-oriented (and beyond) network architecture, in: Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '07, ACM, New York, NY, USA, 2007, pp. 181–192.
[15] L.M. Correia, H. Abramowicz, M. Johnsson, K. Wnstel, Architecture and Design for the Future Internet: 4WARD Project, 1st ed., Springer Publishing Company, 2011. Incorporated.
[16] N. Fotiou, P. Nikander, D. Trossen, G.C. Polyzos, Developing Information Networking Further: From Psirp to Pursuit, vol. 66, Springer, Berlin Heidelberg, 2012. pp. 1–13.

[17] L. Popa, A. Ghodsi, I. Stoica, Http as the narrow waist of the future internet, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, ACM, New York, NY, USA, 2010, pp. 1–6.
[18] C.A. Shue, M. Gupta, Packet forwarding:name-based vs. prex-based, in: Proc. of IEEE INFOCOM'07, 2007.
[19] H. Hwang, S. Ata, M. Murata, A feasibility evaluation on name-based routing, in: Proc. of IPOM2009, 2009.
[20] Y. Wang, H. Dai, J. Jiang, K. He, W. Meng, B. Liu, Parallel name lookup for named data networking, in: IEEE Global Telecommunications Conference (GLOBECOM), 2011, pp. 1–5.
[21] Z.G. Prodanoff, K.J. Christensen, Managing routing tables for url routers in content distribution networks, International Journal of Network Management 14 (2004) 177–192.
[22] B.S. Michel, K. Nikoloudakis, P. Reiher, L. Zhang, Url forwarding and compression in adaptive web caching, in: Proc. of IEEE INFOCOM'00, 2000.
[23] Z. Zhou, T. Song, Y. Jia, A high-performance url lookup engine for url fitering systems, in: Proc. of IEEE ICC'10, 2010.
[24] K. Bharat, A. Broder, M. Henzinger, P. Kumar, S. Venkatasubramanian, The connectivity server: fast access to linkage information on the Web, in: Proc. of the Seventh International World Wide Web Conference, 1998.
[25] S. Han, K. Jang, K. Park, S. Moon, Packetshader: a gpu-accelerated software router, in: Proc. of ACM SIGCOMM'10, 2010.
[26] J. Zhao, X. Zhang, X.Wang, Y.Deng, X. Fu, Exploiting graphics processors for high-performance ip lookup in software routers, in: Proc. of IEEE INFOCOM'11, Mini-Conference, 2011.
[27] K. Kang, Y. Deng, Scalable packet classification via gpu metaprogramming, in: Design, Automation Test in Europe Conference Exhibition (DATE) 2011, 2011, pp. 1–4.
[28] A. Nottingham, B. Irwin, Parallel packet classification using gpu co-processors, in: Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists (SAICSIT '10), 2010, pp. 231–241.
[29] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, J.-M. Shyu, Accelerating string matching using multi-threaded algorithm on gpu, in: 2010 IEEE Global Telecommunications Conference (GLOBECOM 2010), 2010, pp. 1–5.
[30] N. Cascarano, P. Rolando, F. Risso, R. Sisto, infant: Nfa pattern matching on gpgpu devices, SIGCOMM Comput. Commun. Rev. 40 (5) (2010) 20–26.
[31] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, Q. Dong, Gpu-based nfa implementation for high speed memory efficient regular expression matching, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2012.

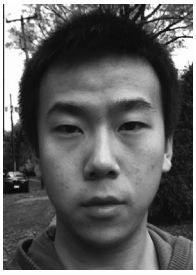
Yi Wang is a postdoctoral research fellow in the Department of Computer Science and Technology, Tsinghua University. He received the PhD degree in Computer Science and Technology from Tsinghua University in July, 2013. E-mail: wy@ieee.org. His research interests include router architecture design and implementation, greening the Internet, fast packet forwarding and information-centric networking.



Huichen Dai is a PhD candidate in the Department of Computer Science and Technology, Tsinghua University. He got his B.S. degree from Xidian University, Xi'an, China, in 2010. His research interests mainly lie in: router architecture, network processor architecture, Named Data Networking (NDN).



Ting Zhang is a PhD candidate in the Department of Computer Science and Technology, Tsinghua University. His recent work focuses on routing lookup, packet classification. His research interests encompass network security and VXworks OS.



Wei Meng received his B.S. degree in Computer Science and Technology, Tsinghua University in 2012. He is now a PhD student in School of Computer Science, Georgia Institute of Technology. His research interests include computer networks and security.

Jindou Fan is now a PhD student at the Department of Computer Science and Technology, Tsinghua University. His research interests include router/switch design, traffic management, greening the Internet and future Internet architecture.

Bin Liu was born in 1964. He is now a full Professor in the Department of Computer Science and Technology, Tsinghua University. His current research areas include high performance switches/routers, network processors, high speed network security and greening the Internet. He has received numerous awards from China and abroad including the Distinguished Young Scholar of China in 2006 and the inaugural Applied Network Research Prize sponsored by ISOC and IRTF in 2011.