PAPER

High-Throughput Exact Matching Implementation on FPGA with Shared Rule Tables among Parallel Pipelines

Xiaoyong SONG[†,††a)], **Zhichuan GUO**[†,††b)], **Xinshuo WANG**[†,††c)], *and* **Mangu SONG**[†,†††d)], *Nonmembers*

SUMMARY In software defined network (SDN), packet processing is commonly implemented using match-action model, where packets are processed based on matched actions in match action table. Due to the limited FPGA on-board resources, it is an important challenge to achieve large-scale high throughput based on exact matching (EM), while solving hash conflicts and out-of-order problems. To address these issues, this study proposed an FPGA-based EM table that leverages shared rule tables across multiple pipelines to eliminate memory replication and enhance overall throughput. An out-of-order reordering function is used to ensure packet sequencing within the pipelines. Moreover, to handle collisions and increase load factor of hash table, multiple hash table blocks are combined and an auxiliary CAM-based EM table is integrated in each pipeline. To the best of our knowledge, this is the first time that the proposed design considers the recovery of out-of-order operations in multi-channel EM table for high-speed network packets processing application. Furthermore, it is implemented on Xilinx Alveo U250 field programmable gate arrays, which has a million rules and achieves a processing speed of 200 million operations per second, theoretically enabling throughput exceeding 100 Gbps for 64-Byte size packets.

key words: *field programmable gate arrays (FPGA), match-action table, exact matching, hash table, hash collision, CAM*

1. Introduction

In software defined network (SDN), most network functions are implemented based on match-action table (MAT) model. In MAT, the specific fields of data packets are extracted as key to probe the matching table, and the action instructions that should be executed are obtained after successful matching [1], [2]. Exact matching (EM) table plays an important role and is widely used in packet processing applications such as packet inspection [3], packet classification [4] and flow monitoring [5] etc. The processing speed of network packets and the scale of networks are increasing continuously, along with higher processing performance requirements for switch devices, which also demand higher performance and scalability to exact matching tables.

Field programmable gate arrays (FPGA) has significant advantages in terms of programmable flexibility and parallel processing, and various network functions are being offloaded to FPGAs for accelerated processing [6]. However, neither EM nor content addressable memory (CAM) is on an FPGA. User needs to design and implement the matching table based on on-board resources. On FPGA, the mainly methods to implement exact matching table include hash-based methods and CAM-based methods. The exact matching table based on CAM consumes huge resource and has a low memory efficiency [7]. EM table based on hash has higher memory efficiency, but there are problems such as hash collision, insertion difficulty, and nondeterministic worst case latency [8], [9]. Moreover, both methods will face difficulties in achieving a large depth or large width EM on FPGA with a high speed.

In order to improve the throughput of the matching table, some designs employ multiple parallel channels. However, it also brings the problem of memory replication [10], resulting in huge on-chip storage consumption. Some multi-channel designs [11], [12] without storage replication also have the problems of low hash table load factor. Moreover, different from the out-of-order execution of a general Key-Value System (KVS) in database, it is necessary to maintain the sequence order of packets in most of network packet processing applications. Hence, the issue of uncertain processing latency or packets out-of-order should also be considered in network matching table application.

To implement a large-scale high-throughput exact matching table and solve the problems of hashing collision and out-of-order among multiple pipelines, this paper proposes a multi-channel exact matching table with shared rule table, which can improve the processing speed of the matching table without memory replication. Especially, it would rearrange the out-of-order matching results after processing to ensure a correct sequence, which avoids packets out-of-order or error in network. The main contributions of this work are as follows:

- This paper proposed an FPGA-based exact matching implementation that leverages shared rule tables across parallel pipelines to enhance overall throughput without memory replication. The implemented EM table based on FPGA could insert a million rules, which has good scalability and achieves a processing speed of 200 million operations per second, theoretically enabling throughput exceeding 100 Gbps for 64-Byte size pack-

Manuscript received August 18, 2023.
Manuscript revised October 18, 2023.
Manuscript publicized January 30, 2024.

[†] The authors are with the National Network New Media Engineering Research Center, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China.

[††] The authors are with the University of Chinese Academy of Sciences, Beijing 100049, China.

[†††] The author is with Suzhou Haiwang Network Technologies Co., Ltd., China.

a) E-mail: songxy@dsp.ac.cn
b) E-mail: guozc@dsp.ac.cn (Corresponding author)
c) E-mail: wangxs@dsp.ac.cn
d) E-mail: songmg@dsp.ac.cn

DOI: 10.23919/transcom.2023EBP3140

ets.

- An out-of-order reordering function to recover the order of matching results within the pipelines to maintain packet sequence in network packet processing.
- A compact CAM-based exact match table is incorporated alongside the primary hash-based EM table within each pipeline to handle hash collisions, which ensures the important rules can be inserted into rule tables.

2. Exact Match Table Overview

2.1 Match-Action Model

As Fig. 1 shown, match-action model is the mainstream framework to process data packets in data plane of programmable devices. At each processing stage of data packets, the feature match filed in the packet is extracted as *Key* and used for MAT table lookup operation [2], [13]. The action engine then executes actions based on the result of the table lookup. Among the various types of MAT tables used in packet processing and pattern matching, the exact matching table is commonly employed.

There are two main ways to implement EM on FPGA, which is hash-based EM like [10] and CAM-based EM like [14]. Both hash-based and CAM-based exact matching tables have O(1) lookup performance. In contrast, hash-based EM has higher storage utilization efficiency, while the SRAM-based CAM has low storage utilization. However, CAM-based EM does not have the problem like hash collision or insert difficulty, etc.

2.2 Hash-Based Exact Match Table

The hash-based exact match table shown in Fig. 1(a) is a fast and efficient data structure that stores *Key-Value* pairs in the {*Vld, Key, Value*} data structure in each address space. During inserting or querying, the *Key* is hashed to generate the corresponding address index, and then the data structure is stored at the address or retrieved for comparison, ultimately yielding the corresponding value.

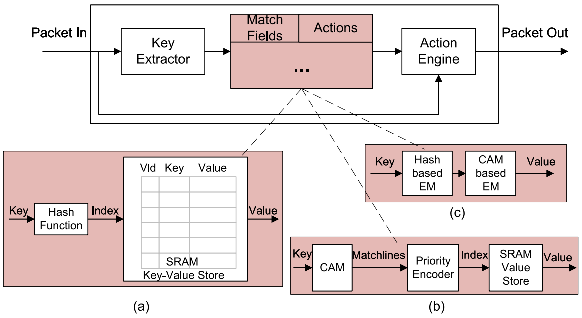


Fig. 1 Basic scheme of match-action model. (a) Architecture of hash-based exact match table. (b) Architecture of CAM-based exact match table. (c) Architecture of non-collision exact match table.

2.3 CAM-Based Exact Match Table

Content Addressable Memory (CAM) is a type of memory that enables fast content queries and has the advantage of fast search rate. As Fig. 1(b) shown, in CAM-based exact matching table, the *key* is entered into CAM to get the matching information *matchlines* which contains all match result of each address unit, and the match address index is encoder by *Priority Encoder*. Finally, this address is used to read the corresponding *Value* from the *Value Store*.

2.4 Non-Collision Exact Match Table

The probability of collision depends on the hash function, which is not possible to be perfect, especially in the case of random and frequently updatable data [7]. In order to solve the hash conflict, the cuckoo hash [9], multiple level hash table [10] or chaining [15], adding auxiliary storage [5], [12], [16], and other solutions have been proposed. Figure 1(c) shows a non-collision exact match table combined with hash-based EM and CAM-based EM. The rule entry is firstly inserted into hash-based EM table. If a collision occurs, the conflicted entry is then inserted into the CAM-based EM table. This hybrid structure leverages the benefits of both CAM and hash-based techniques to ensure efficient and collision-free matching.

3. Architecture

Although hash table has good scalability and high resource utilization, implementing a high-performance EM table on FPGA is still a challenge, especially when the size of table is large. It is difficult to perform matching with sufficient throughput for wire-speed processing. For instance, in a 100 Gbps high-speed network, at least 148.8 million of 64B size packets per second must be processed to meet processing speed requirements, which means the operation throughput of matching table should not be lower than 148.8 million.

The core idea of increasing the processing speed is to maximize the number of operations processed per clock cycle. Usually, the processing speed is increased by boosting the main frequency of system or utilization of multiple parallel pipelines [17]. It is not easy to improve the frequency on FPGA, especially when the entire system is complex and the table size is large. The problem faced by multiple parallel pipelines is that it requires multi-port memories or memory replication to store each rule several times, which consumes more storage resources. Due to the limited resources on FPGA, it is not feasible to use the method of memory replication when implementing a very large scale matching table.

3.1 Parallel Shared Hash Table with CAM Structure

To avoid storage replication and increase the number of entries processed in a single clock cycle, we optimize the multi-level hash pipeline structure to multiple parallel pipelines

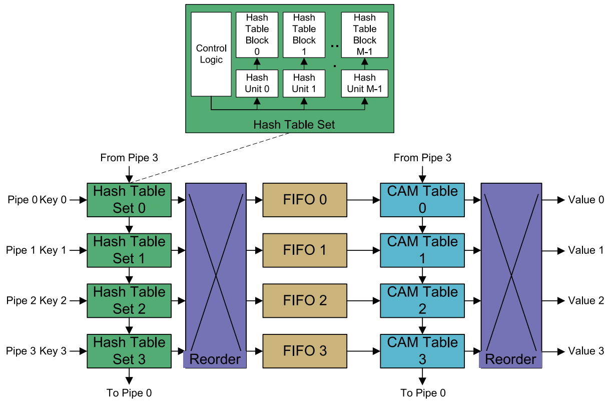


Fig. 2 Overall architecture of exact table with 4 parallel pipelines.

structure. Additionally, multiple CAM tables were adopted to handle hash conflicting.

All pipelines share all rules stored in the entire exact matching table, and each rule is only stored once time without backup. The table in each pipeline can be accessed by the operation from its neighbor pipeline if needed. For each key, there is a probability that they will be inserted or matched in the hash table set of each pipeline. If the operation succeeds in a hash table set, there is no need to access other tables in other pipelines. Meanwhile, the tables in other pipelines can process other operations. In the worst-case scenario, when all EM tables in all pipelines need to be accessed for each key, the throughput of the entire EM table is the same as that of a single pipeline. In the best scenario, all operations are succeed in the first hash table set they access, and the entire exact matching table could handle P operations in each clock cycle. However, for most cases, $1 \leq p \leq P$, where p is the number of operations EM can process in a clock cycle and P is the number of pipelines.

In the entire EM table, there are P parallel pipelines, each consisting of a set of hash table blocks as the main storage and a CAM-based EM table as auxiliary storage. Figure 2 shows the architecture of our EM table with four parallel pipelines. Multiple hash functions and hash table blocks are used in each hash table set to reduce the hash collision rate. To avoid the uncertainty in insertion time caused by cuckoo hashing, we select an empty address space from multiple alternative addresses in parallel, and the conflicted new entry would be inserted in other hash table sets instead of replacing existing entries if there is no empty alternative address in current hash table set. Entries failed inserted into all hash tables are eventually inserted into CAM. In order to maintain the sequence of packets and ensure a constant search latency, there is a *Reorder* module behind the hash table sets and CAM tables to return the searched key and matched result to its own input pipeline, and unify the latency of each search key according to its operation path.

3.2 Hash Table Block and Hash Table Set

In each pipeline, there is a set of hash-based EM table blocks, here called the *hash table set*. Each hash table set consists of M hash table blocks, and these hash table blocks are independent and store *Key-Value* pairs in their address units with the entry structure of $\{Vld, Key, Value\}$. If *Vld* is '1', it indicates the entry structure is valid and the slot in hash table block has been used.

3.2.1 Class *H3* Hash Function

The performance of a hashing scheme depends on the collision handling method and the hashing function chosen [18].

Class *H3* hash algorithm [8] was used to perform the hashing operation on the key, which has been demonstrated to be effective on distributing keys randomly [10]. Let i denotes the number of bits for input key, and j denotes the number of bits for hash index. Let Q denotes a $i \times j$ Boolean matrix. For a given $q \in Q$, let $q(m)$ be the bit string of the mth row of Q, and let $x(m)$ denote the mth bit of input key. The hashing function $h(x) : A \rightarrow B$ is defined as

$$h(x) = (x(1) \cdot q(1)) \oplus (x(2) \cdot q(2)) \oplus \ldots \oplus (x(i) \cdot q(i)). \quad (1)$$

Compared to other hashing algorithms like *Toeplitz* [19], the *H3* algorithm not only ensures uniformity and fast computation but also consumes fewer logic resources when implemented on an FPGA. The hardware which stores *H3* matrix can be organized in a bank of registers. The same hardware can realize any desired hashing function from this class and the hashing function can be changed dynamically by loading data into the bank of registers if needed [18]. To improve the clock frequency, the hashing operation is pipelined and completed within two clock cycles.

3.2.2 Hash Collision Handling

To reduce hash collisions, an independent *H3* hash matrix is set for each hash table block, and the entire EM table has $P \times M$ *H3* hash matrices and $P \times M$ hash function units. If a hash collision occurs during insertion, the new entry would select another empty slot to insert, instead of replacing the original entry like cuckoo hashing. In each hash table set, M hash function units perform hash calculations on the same key parallelly, and then choose an address without hash conflict from the M candidate addresses for insertion. With an increasing number of hash table blocks in each hash table set, the hash collision rate would be reduced significantly, which would be explained further in Sect. 4.1. If there is no empty candidate address in the current hash table set, the insertion operation proceeds to the hash table set of the next pipeline. If all hash tables fail to insert, the conflicting entries are stored in the CAM table finally.

3.2.3 Operations

- *Insert*: As illustrated in Fig. 3, for each hash table set, M hash function units in this set first generate M candidate addresses for the key during entry insertion. Then the

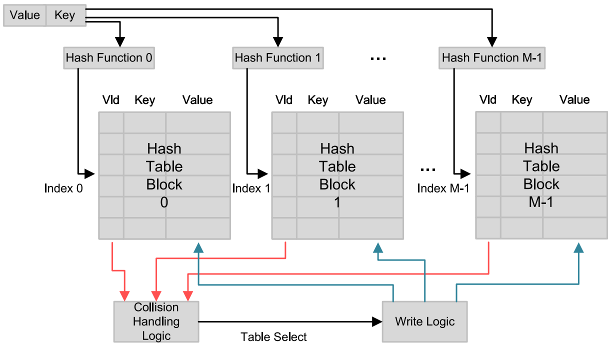


Fig. 3 Insert process of hash table set.

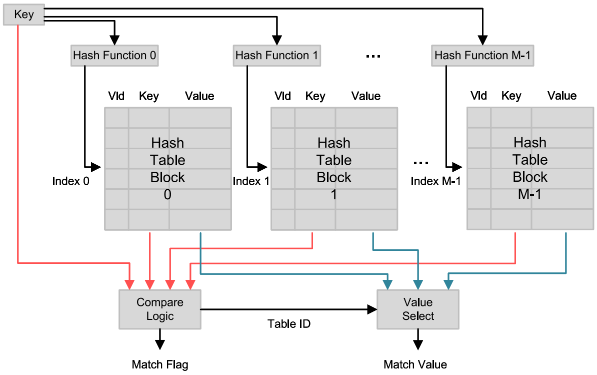


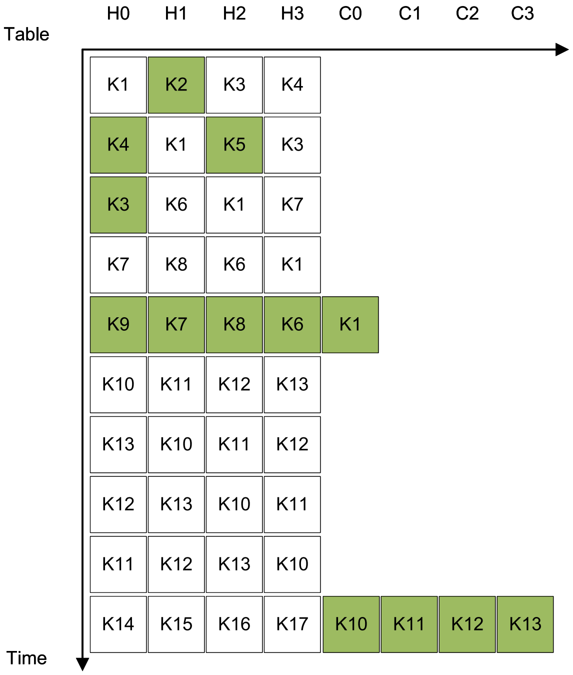
Fig. 4 Query process of hash table set.

entry structure stored in the candidate address of each hash table block is read. If the *Vld* bit in entry structure is ‘0’, the address is empty, and ‘1’ indicates that this address space has been already in use. The *Collision handling logic* selects a hash table block with empty candidate address to insert. *Write logic* writes the entry structure of new entry into the corresponding address of the selected table.

- *Query*: As illustrated in Fig. 4, after hashing calculation and entry structure reading, the queried key is compared with all keys in valid entry structures. After comparing, *Compare logic* encodes all comparison results and gets the matching address, and then the matching value is selected according to the matching address.
- *Delete*: Performs a query operation firstly. After the matching is successful, the content of the matching address is written to 0 to delete the entry.

3.3 Auxiliary CAM Tables

Even if we use multiple hash functions and multiple hash table blocks to reduce the probability of hash collisions, a perfect hash function does not exist. To avoid situations where important rules cannot be inserted into hash tables due to hash conflicts, we handle this problem by adding auxiliary storage, namely a small depth CAM-based EM table for storing entries that cannot be inserted into the hash table.



H0	H1	H2	H3	C0	C1	C2	C3
K1	🟩 K2	K3	K4				
🟩 K4	K1	🟩 K5	K3				
🟩 K3	K6	K1	K7				
K7	K8	K6	K1				
🟩 K9	🟩 K7	🟩 K8	🟩 K6	🟩 K1			
K10	K11	K12	K13				
K13	K10	K11	K12				
K12	K13	K10	K11				
K11	K12	K13	K10				
K14	K15	K16	K17	🟩 K10	🟩 K11	🟩 K12	🟩 K13

Fig. 5 Timeline of the operation (H: Hash Table Set. C: CAM Table. K: Operation Key. H0, H1, H2, H3 are the four hash table sets in Fig. 2, and C0, C1, C2, C3 are the four CAM tables in Fig. 2. K0, K1, $\cdots$, K13 are the operation keys into the tables).

The CAM here is implemented using the transposed SRAM method [20]. In the implementation of this method, key is used as write or read address, and *matchlines* containing entry address information are stored in SRAM. The width of *matchlines* is equal to the depth of CAM, with each address space corresponding to a single bit in *matchlines*. For a given key, if a particular bit in its corresponding *matchlines* is ‘1’, it means that the address is a matching address.

Theoretically, the depth of CAM depends on the probability of hash collision, and a detailed analysis will be provided in Sect. 4.2. Here, we assume that the total CAM depth requirement is C_{depth}. Figure 5 illustrates the timeline of the operation in the EM table. When a key K enters the matching table, it will appear in the timeline and the grid in Fig. 5. The white grid indicates that the operation of K is not completed yet, and it needs to continue entering the next hash table set or CAM table to try its operation with the loop order of $Pipe\ 0 \rightarrow Pipe\ 1 \rightarrow Pipe\ 2 \rightarrow Pipe\ 3 \rightarrow Pipe\ 0$. The green grid indicates that the operation of K has been successfully completed or all tables have been accessed. For each key, if its operation is failed in all hash tables, it would further enter into CAM table. The key failed to do operation in a CAM table would access the next CAM table with the same loop order of $CAM\ 0 \rightarrow CAM\ 1 \rightarrow CAM\ 2 \rightarrow CAM\ 3 \rightarrow CAM\ 0$. For instance, the operation of *K1* is failed in all hash table sets and it finally completes its operation in CAM table 0. To avoid the worst-case scenario which shown in Fig. 5, when all hash table sets in multiple pipelines need to insert conflicting entries into CAM simultaneously (*K10*, *K11*, *K12*,

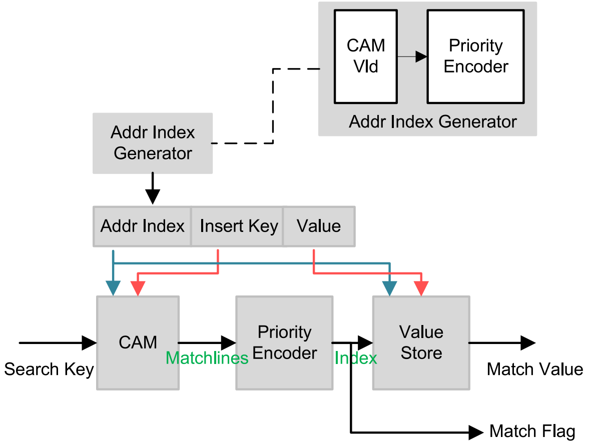


Fig. 6 Architecture of CAM-based EM table.

and *K13*), we place a CAM table after each hash table set in each pipeline. The key of *K10*, *K11*, *K12*, and *K13* are failed to do their operation after traversing all hash tables in four pipelines. After the first Reorder module behind the hash tables in Fig. 2, *K10* ~ *K13* enter into the CAM tables with their corresponding pipelines. The depth of the CAM in each pipeline $CD = \frac{C_{depth}}{P}$, where P is the number of pipelines or the number of hash table sets.

By combining CAM-based EM tables with hash-based EM tables, it can address potential hash conflicts and ensure that important rules are correctly inserted into the match table.

3.3.1 Address Spaces Management

As shown in Fig. 6, for each CAM table there is a bitmap *CAM Vld* vector which records the usage status of each CAM address space. '0' means that the space is already in use, while '1' means that it is available for use. When inserting an entry into CAM, the address index generator allocates an empty address space to this entry by the *vld* bitmap and a priority encoder. After an entry is deleted, the corresponding address *vld* bit corresponding would be set to '1' again, indicating this address can be reallocated.

3.3.2 Operations

- *Insert*: After generating the write index, the *new content* of the entry is generated based on the write index ($new\ content = 1 << write\ index$). In the same time, the original content in the address of writing key should be read out as *old matchiline*. The *new content* and *old matchiline* perform the ***or*** operation to generate the *new matchiline* written into CAM. *New matchiline* = *old matchlines* | *new content*. Use the entry key as write address, write this *new matchiline* into the CAM. At the same time, write the corresponding value into value store at the write index. Set the corresponding bit in the *vld* bitmap to '0' to indicate the address is now in use.

To CAM (1b)	Pipeline State (4b)	Option Code (2b)	Succeed (1b)

Fig. 7 State structure across EM pipelines.

- *Query*: During a query, the key is used as the read address to read the *matchlines* stored in CAM. A priority encoder is used to encode the *matchlines* and obtain the matching index. If the address space is in use, read the corresponding value stored in this address space from the value store.
- *Delete*: After completing the query operation, clear the content of corresponding bit in matchlines at the key's address in CAM and also clear the content at the corresponding index of the value store. Set corresponding bit in bitmap to '1', indicating that it can be used again.

3.4 State Structure across Pipelines

In addition to entry structure stored in EM table, a custom data structure is maintained and transferred among pipelines to record the status and data path of each operation, here we call it *state structure*. As shown in Fig. 7, this state structure has a length of 8 bits, and each sub-field is defined as follows:

- [0]: *Succeed* – Indicates whether the operation of an entry is successful. It is set to 1 if the entry has been successfully inserted or if a query has been successful.
- [2:1]: *Option Code* – Specifies the operations to be performed on this entry. 2'b00 indicates no operation, 2'b01 indicates insertion, 2'b10 indicates deletion, and 2'b11 indicates query.
- [6:3]: *Pipeline State* – A 4-bit bitmap represents the status of whether pipelines or tables have been accessed. 1'b0 indicates that the table has been accessed, and 1'b1 indicates that table has not been accessed. For example, 4'b1011 indicates that the entry is entered from *Pipe* 2 and the table in *Pipe* 2 has been accessed. If further access is needed, the next step is to jump to *Pipe* 3 → *Pipe* 0 → *Pipe* 1 for access until operate succeed or all pipelines have been accessed. For table entries that have not been successfully operated in hash tables, this field will be restored to its initial state of 4'b1111 before entering CAM tables.
- [7]: *To CAM* - Determines whether to insert an entry into the CAM when insertion fails in all hash tables. 1'b0 means the entry can be directly discarded when insertion fails in hash tables, while 1'b1 means the entry should still be inserted into the CAM if needed.

As a key is queried or a *Key-Value* is inserted into the EM, the hash table or CAM table performs the operation based on its *Option Code*. If the current pipeline executes successfully, it directly jumps out of the table and enters the *Reorder* module. Otherwise, if there is another pipeline table that has not been accessed according to the *Pipeline State*, it continues to jump to the next pipeline table for corresponding operation.

Once the hash table or CAM table of each pipeline completes its operation, it modifies the corresponding bit in the *Pipeline State* of the pipeline for that pipeline and updates the *Succeed* bit based on the operation success. Additionally, *Reorder* would control the exit time of each entry according to its *Pipeline State*, which ensures the processing latency of each entry is equal.

For a query entry, the state structure is propagated along the query entry until the matching result gets used. However, when inserting an entry, once the insertion is successful, the structure will not propagate backwards further.

3.5 Rearrange Out-of-Order

According to the previous design, when an entry completes the query operation in a hash table set or a CAM table, it will carry its state structure away from the table to allow more new entries to access the matching table for processing. Since each entry may not need to access all hash table sets or CAM tables, the operation latency vary from entry to entry. This would result in out-of-order and congestion at the out-ports of the EM table. Additionally, in network packet processing applications, it is usually necessary to maintain packet sequence.

To ensure that the sequence of packets entering and leaving the pipeline is not disrupted, and that the query latency of each entry is consistent, a *reorder* module is introduced to restore the order of processed entries and make corresponding delay for each entry.

As shown in Fig. 8, there are four channels in *reorder* module, and each corresponds to one channel in the EM table. Taking the hash table as an example, each queried entry carries its status structure from the current hash table set into the *reorder* module. The *parsing unit* parses its pipeline state field to find out which pipeline the entry enters the matching table from and how many hash table set it has been accessed. The module of *path select* dispatches the matching result back the pipeline it entered. Then *delay select* module selects an appropriate additional delay for this entry and outputs it from the corresponding outport of *reorder* module.

After being processed by the *reorder* module, the query latency of each entry is consistent and it is equal with the worst-case delay, and the corresponding matching results can still be returned to its own pipeline after cross-pipeline lookups, which ensures the sequence of packets in each pipeline in the later processing.

As illustrated in Fig. 9, the girds with the same color enter into the table simultaneously, and the colored grid means a key has completed its operation. For example, *K1*, *K2*, *K3* and *K4* are all green because they all entered into the EM table at the first time. *K1* completed its operation in its first table set *H0*, but *K2*, *K3* and *K4* not. They went through 2, 3 and 4 tables to complete the operation respectively. Then they entered the reorder module from the channel which they complete operation. After reordering, these four keys exited the EM table simultaneously.

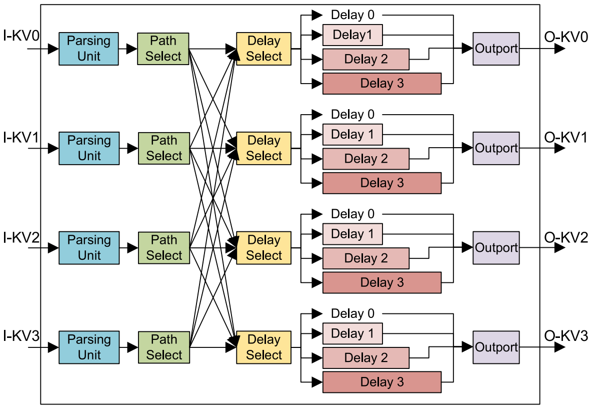


Fig. 8 Architecture of reorder.

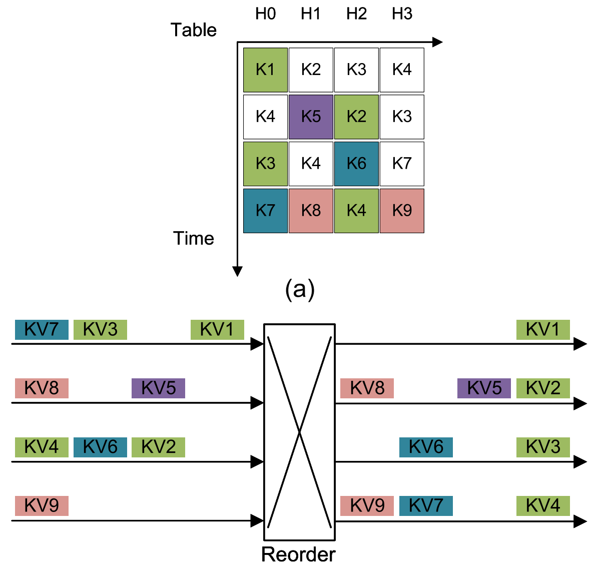


Fig. 9 Timeline of reorder operation.

The keys entered from a same pipeline also keep their sequence after reordering. *K4*, *K7* and *K9* entered from *H3* table set at different time. Although they have different processing latency in hash tables, they still maintain their sequence after reordering.

4. Analysis

Here we analyze the hash collision rate, the capacity CAM required and the issue of consistency. Table 1 lists the variable abbreviations that will be used later and their meanings.

4.1 Hash Collision Rate

There are P hash table sets in the entire EM table, and each hash table set contains M hash table blocks with the depth of

Table 1 Tabel of abbreviation.

Abbr	Description
$r_{collision}$	Hash collision rate of a hash table set.
$R_{collison}$	Hash collision rate of the entire hash-based EM table.
$N_{collision}$	The number of conflicted entries of the entire hash-based EM table.
C_{depth}	The depth of entire CAM-based EM table.
H_{depth}	The depth of entire hash-based EM table.
K	Width of key.
V	Width of value.
HD	The depth of each hash table block.
M	The number of hash table blocks in each hash table set.
P	The number of pipelines.
CD	The depth of each CAM table in each pipeline.
$HTSUN$	The number of URAMs used by each hash table set.
$HTBUN$	The number of URAMs used by each hash table block.
$CAMBN$	The number of BRAM36Ks used by each CAM-based EM table.
TUN	The number of URAMs used by entire hash-based EM table.
TBN	The number of BRAM36Ks used by entire CAM-based EM table.
Ep	The expected hash table set number of each entry needs to query.

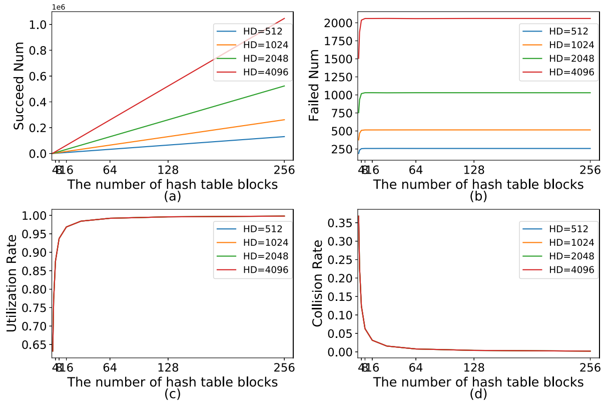


Fig. 10 Simulation results under different hash table block's number and depth. (a) The number of entries inserted into hash tables successfully. (b) The number of failed entries which are not inserted into hash tables. (c) The utilization rate of the entire hash table. (d) The collision rate of the entire hash table.

HD. Hence, there are $P \times M$ hash table blocks. We simulated the hash collision rate under different hash table block's number and depth. The probability of each key hashing to a particular location is uniform [21], so uniform random function acts as hash function unit to generate insert index in simulation. 1000 simulation experiments were conducted for each case and calculated the mean value. In each time, $P \times M \times HD$ entries are inserted into the hash tables.

The simulation results are shown in Fig. 10. The Fig. 10(a) shows the number of successful inserted entries and Fig. 10(b) shows the number of failed entries under different table numbers and different table depths, which provide us a reference to choose the depth of CAM under different cases. It can be seen from Fig. 10(c) and Fig. 10(d) that with the increase of the number of hash table blocks, the hash table utilization rate (load factor) would increase and the collision rate would decrease. When the number of entries inserted into all hash tables is the same as the total number of address spaces in hash tables, the collision rate of entire hash table is almost unaffected by the depth of each hash table block, but mainly determined by the number of hash table blocks. When the number of hash table blocks exceeds 64, the hash collision rate drops below 1%.

4.2 CAM Capacity

Assuming that the hash collision rate is $R_{collison}$, there are P hash table sets, and each hash table set has M hash table blocks with the depth of HD. Then there are H_{depth} (= $P \times M \times HD$) address spaces in the entire hash table. After inserting H_{depth} entries into hash tables, there would be $N_{collison}$ (= $R_{collison} \times H_{depth}$) entries cannot be inserted into the hash table finally.

Hence, the capacity of CAM C_{depth} required should be equal the number of conflicted entries.

$$C_{depth} = N_{collision} = R_{collision} \times (P \times M \times HD). \quad (2)$$

The CAM on each channel should be CD, and

$$CD = \frac{C_{depth}}{P}. \quad (3)$$

4.3 Consistency

Due to the latency in hash computation and SRAM read/write operations, there are two extreme scenarios where consistency issues may occur: (i) a queried key is the same as an inserting key; (ii) in a hash table set, a key is being inserted, the new inserted key takes the insert address of the inserting key as its candidate address.

For the former, if the same search key accesses the matching table during the writing process of an entry, it may result in incorrect query results. For the latter, the status of the inserted address is updated to the occupied state only after the entry is inserted completely. During this insertion period, the address space is still considered to be selected for subsequent insert entries, which may lead to a collision between two entries when selecting an address.

However, the probability and impact of these two scenarios are negligible. Firstly, compared with query operation, insertion operation is generally infrequent. Moreover, it is extremely rare for multiple collisions of a single address to occur in such a vast depth of table, especially within a very short period of time. The first case has been discussed to be negligible in prior work [10]. In the second case, entries can be inserted from different pipelines in the way of round-robin, or new entries can be inserted after confirming that the previous entry has completed the insertion operation.

5. Implementation and Evaluation

5.1 Implementation

We implement our design on a Alveo U250 [22] FPGA device, which has 1,728,000 LUTs, 3,456,000 Flip-flops, 2688 BRAM36K memory blocks and 1280 URAM memory blocks. In order to make a trade-off between latency and throughput, the EM table has a total of 4 channels in our implementation, each channel has a hash table set and an auxiliary CAM based EM. Each hash table set has 64 hash table blocks with a depth of 4K. Based on the hash collision rate, it can be determined that 4K CAM entries are sufficient to store the collision entries of the hash table. Therefore, the CAM depth of each pipeline is 1K. The total EM can store 1048K (4*64*4096=1048576) entries.

5.2 Memory Utilization

The SRAM storage resources on FPGA are independent units, and each SRAM unit must be used by block. We utilized URAMs to implement hash table and transposed BRAMs to implement CAM. Here, each URAM block is configured as a 4K*72b SRAM and each BRAM36K block is configured as a 512*72b SRAM. The symbol of $\lceil * \rceil$ represents rounding up.

The number of URAMs used by each hash table block is

$$HTBUN = \left\lceil \frac{HD}{4096} \right\rceil \times \left\lceil \frac{K+V+1}{72} \right\rceil. \tag{4}$$

The number of URAMs used by each hash table set is

$$HTSUN = M \times \left(\left\lceil \frac{HD}{4096} \right\rceil \times \left\lceil \frac{K+V+1}{72} \right\rceil \right). \tag{5}$$

The number of BRAM36Ks used by each CAM based EM table is

$$CAMBN = \left\lceil \frac{K}{log_2^{512}} \right\rceil \times \left\lceil \frac{CD}{72} \right\rceil + \left\lceil \frac{CD}{512} \right\rceil \times \left\lceil \frac{V}{72} \right\rceil. \tag{6}$$

The total memory blocks consumed by the entire EM table is *TUN* URAM blocks and *TBN* BRAM36K blocks, in which

$$TUN = P \times \left(M \times \left(\left\lceil \frac{HD}{4096} \right\rceil \times \left\lceil \frac{K+V+1}{72} \right\rceil \right) \right), \tag{7}$$

and

$$TBN = P \times \left(\left\lceil \frac{K}{log_2^{512}} \right\rceil \times \left\lceil \frac{CD}{72} \right\rceil + \left\lceil \frac{CD}{512} \right\rceil \times \left\lceil \frac{V}{72} \right\rceil \right). \tag{8}$$

Table 2 shows the resource utilization for different widths of key and value in our implementation. In entire EM Tables, there are totally 1048K address spaces in hash tables and 4K address spaces to store conflicted entries in CAM-based EM tables. In general, the logical resource occupation remains within a reasonable range, which reserves enough resource and frequency space for the implementations of other on-board applications. The utilization of storage resources is directly proportional to the width of keys or values. However, in some cases, because of the SRAM must be used in the unit of an entire block, there may be storage waste, which is inevitable in FPGA implementations. In addition, when the depth of a matching table remains constant, increasing the width may lead to a decrease in achievable frequency. This is because increasing the width requires more on-board resources and may result in more complex routing and longer signal propagation paths. This increases wire delay and limits the operating frequency of the entire matching table.

Table 2 Resource utilization of 1048K entries exact matching table on U250 FPGA.

Value Width (bits)	**Key Width (bits)**		**LUT (%)**	**FF (%)**	**BRAM (%)**	**URAM (%)**	**Frequency (MHz)**
64	16	Total EM	3.45	1.66	4.61	40	200
		Hash Part	2.29	1.06	0	40	
		Cam Part	1.13	0.57	4.61	0	
	32	Total EM	4.09	1.96	8.93	40	200
		Hash Part	2.43	1.08	0	40	
		Cam Part	1.63	0.84	8.93	0	
	64	Total EM	5.69	2.28	17.56	40	192.308
		Hash Part	3.09	1.13	0	40	
		Cam Part	2.56	1.1	17.56	0	
	128	Total EM	7.86	2.45	32.66	60	163.934
		Hash Part	3.98	1.22	0	60	
		Cam Part	3.82	1.15	32.66	0	
128	16	Total EM	4.52	2.29	4.91	60	185.185
		Hash Part	3.23	1.64	0	60	
		Cam Part	1.18	0.6	4.91	0	
	32	Total EM	5.5	2.56	9.23	60	185.185
		Hash Part	3.69	1.62	0	60	
		Cam Part	1.76	0.88	9.23	0	
	64	Total EM	7.18	2.88	17.86	60	178.571
		Hash Part	4.39	1.66	0	60	
		Cam Part	2.74	1.15	17.86	0	
	128	Total EM	9.18	3.04	32.96	80	150.015
		Hash Part	5.28	1.74	0	80	
		Cam Part	3.84	1.21	32.96	0	

5.3 Performance Evaluation

For each hash table set, its collision rate is $r_{collison}$, which also means the probability of transferring from one hash table set to its adjacent channel after the insertion failure. As can be seen from Fig. 10(d), when there are a larger number of hash table blocks, almost all entries can be successfully inserted into its first hash table set. Taking an example when each hash table set has 64 hash table blocks (i.e., M=64), the collision rate $r_{collison}$ is approximately 0.00785, which means the entries rarely moves to other channels to insert. Hence, the speed of insertion would be effectively enhanced by multiple parallel pipeline channels.

For queries, the total number of address spaces in each hash table set is equal, so the depth of each hash table set is $\frac{1}{P}$ of the entire table (here do not consider the minimal number of entries in CAM), and the probability of a successful query for each table entry in the current table is $\frac{1}{P}$. On average, the expected number of hash table sets to be queried for each table entry can be denoted as Ep. Therefore, in average cases, it is equivalent to having $\frac{P}{Ep}$ channels working in parallel in our EM table.

The expected number of tables to be queried for each entry is

$$\begin{aligned} E_p &= \frac{1}{P} + 2\frac{1}{P}\left(1-\frac{1}{P}\right) + \cdots + i\frac{1}{P}\left(1-\frac{1}{P}\right)^{i-1} + \cdots \\ &\quad + (P-1)\frac{1}{P}\left(1-\frac{1}{P}\right)^{P-2} + P\left(1-\frac{1}{P}\right)^{P-1}, \qquad (9) \\ &= \sum_{i=1}^{P-1}\left(\frac{i}{P}\left(1-\frac{1}{P}\right)^{i-1}\right) + P\left(1-\frac{1}{P}\right)^{P-1}. \qquad (10) \end{aligned}$$

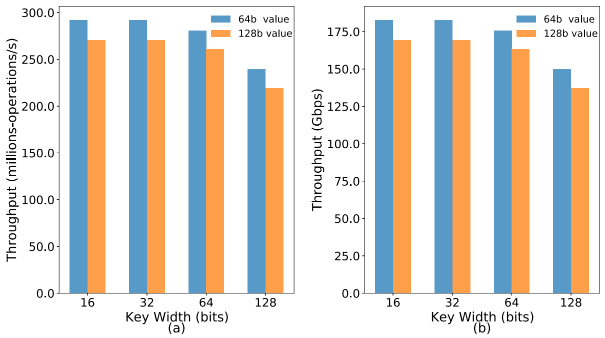


Fig. 11 Throughput under different EM configuration. (a) Throughput of operations. (b) Throughput of packets with length of 64B.

For the entire EM table, although each operation has a certain latency, both insert operations and query operations are pipelined. Therefore, the EM table can do $\frac{P}{Ep}$ operations per clock cycle on average. Therefore, the operation throughput of the EM table is

$$Operation\,Throughput = \frac{P}{Ep} * Freq. \tag{11}$$

For packets, the theoretical throughput that can be achieved is

$$Throughput = \frac{P}{Ep} * Freq * (Pkt\,Len + 20)B. \tag{12}$$

The additional 20 bytes are the extra overhead of packet transferring in network, which includes: 12 bytes inter frame gap (IFG) which is the minimum frame gap of Ethernet packets (IEEE 802.3), 7 bytes preamble for clock synchronization and 1 byte start of frame delimiter (SFD) for identifying the start of the frame.

In our implementation, the number of pipelines is 4. When P is 4, $Ep(P = 4) \approx 2.73$, and $\frac{P}{Ep} \approx 1.46$. Therefore, the EM table can handle 1.46 operations per clock cycle on average. According to the implementation frequency of EM in Table 2, we can calculate the operation throughput and supported packet throughput of EM table in different cases.

Figure 11 shows the number of query operations that EM can handle per second and the corresponding 64B packet throughput under different conditions. A smaller EM table is easy to achieve higher throughput because it could reach a higher working frequency. Overall, the entire EM table can process more than 200 million of operations per second, and can reach a throughput of about 125 Gbps for 64B packets.

6. Related Work and Discussion

Exact matching table is a research hotspot and is widely used in database, key-value store and packet classification etc., and hash-based exact matching table is a mainstream method on FPGA implementation. Researchers mainly focusing on scale expansion, hash collision handling and throughput enhancement of hash table on FPGA, which are also our main work in this paper.

With the continuous expansion of the network scale, the size of the matching table is also increasing. Although hash table is a storage efficient structure, the implementation of huge matching tables on FPGA would still encounter problems such as implement difficulties, resource constraints, and frequency reduction etc. Besides this, solving the hash collision problem is one of the key challenges to achieve accurate matching tables. Researchers use different methods to reduce collisions, such as using better hash functions, open addressing methods, chain methods, etc. Implementing collision resolution algorithm on FPGA needs to consider the balance between hardware resource utilization efficiency and throughput. To increase throughput, the researchers explored a variety of approaches. For example, parallel access is achieved by querying and manipulating multiple hash buckets in parallel.

Y.Z. Li [16] proposed a non-collision hash scheme using bloom filter (BF) and CAM to ensure that each lookup accesses memory at most once. An additional CAM is used to store the conflicting entries of hash table. And a bloom filter to pre-detect if an entry is in hash table ensures that each lookup accesses hash table or CAM at most once. It achieves better worst-case performance and has greater flexibility to quickly insert or query entries. However, bloom filter has some problems such as the difficulty of deleting, and it consumes a lot of resources when implementing a large matching table, which is not feasible in practice.

M. Sha [5] proposed to solve the cuckoo hash conflicts by using a set of distributed RAM as auxiliary storage, which is actually a small CAM implemented by distributed RAMs and registers. However, it has limited scalability especially under a bigger depth requirement of CAM when the matching table has huge depth. Additional, there may be uncertainty in the insertion time in this design because of the cuckoo hashing and the entries in extended table may be rewritten back into the hash table.

Yang et al. [10] proposed FASTHash to optimize hash table throughput through multiple parallel pipeline designs. In this design, it carries out memory replication on each pipeline, which means the tables in each pipeline are the same. Although it improves the throughput of the hash table, it consumes great storage resources to store the same rules multiple times, and it is infeasible to do memory replication on resource-limited FPGA when the size of the matching table is very huge. In addition, although the design reserves multiple slots for each address space to avoid hash collision, it does not solve the hash conflict more thoroughly. In some cases, there will still be entries that cannot be inserted successfully.

Salvatore Pontarelli Pedro Reviriego et al. [11] compared serial, parallel and parallel-pipeline hash table implementations, and proposed parallel d-pipeline implementation which increases the throughput by accessing the tables in parallel. However, the hash collision in cuckoo hashing does not further be solved in this design.

W.Q. Wu et al. [12] introduced CAM into d-Pipeline to address hash collision further and kept the high throughput

Table 3 Comparison with other methods.

Architecture	Collision Free	Constant latency	Auxiliary CAM/Memory	Multiple Pipelines	Memory Replication	Throughput	Load Factor of Hash Table	Out-of-Order Recovery
BF-Hash-CAM[16]	Yes	Yes	Yes	No	No	Low	Low	No need
FASTHash [10]	No	Yes	No	Yes	Yes	High	High	No need
d-Pipeline [11]	No	No	No	Yes	No	High	Mid	No
[12]	Yes	No	Yes	Yes	No	High	Mid	No
Proposed	Yes	Yes	Yes	Yes	No	High	High	Yes

of parallel hash tables. However, the structure has only one CAM unit after multiples hash tables. When load factor of hash table is high, the insertion of hash table is difficult and multiple conflicted entries need to access CAM simultaneously. Moreover, the CAM it used needs 16 clock cycles to finish a write operation. Limited by the writing speed of CAM, if there is entry to be written to CAM, the hash table needs to stall and wait its completion. The CAM cannot be adapted to the parallel hash tables with high throughput. In addition, the out-of-order problem is not considered in the design, and it is not applicable in some scenarios that require the sequence of network packets.

Table 3 shows the comparison of our work with existing methods. Based on resource considerations, we did not adopt the bloom filter in our design like BF-HASH-CAM [16]. Compared to [5], it does not replace the existing entry in insertion when collision occurs in our design, which avoids the uncertain insertion latency caused by cuckoo hashing. By sharing the rule matching table among multiple pipeline channels, our method avoids storage replication in FASTHash [10] and improves throughput. At the same time, the load factor is enhanced by increasing the number of hash table blocks, and hash conflicts are handled by auxiliary CAM units. In addition, for the network packet processing scenario, this paper specially considers the out-of-order recovery in the multiple parallel pipeline channels, which is not considered in the design of d-Pipeline [11] and [12].

Actually, there are several dedicated SmartNICs products or solutions to offload SDN packet processing these years, such as Nvidia's ConnectX series [23], Xilinx's Alveo U25 [24] and SN1000 [25] SmartNICs, Microsoft's Bluebird [26] etc. The proposed matching table is an platform-independent module, it could be embedded into these systems to support SDN packet processing as well.

7. Conclusions

In summary, this paper presented a large-scale high-throughput collision-free EM table which shares rule tables and with out-of-order recovery among multiple parallel pipelines for the network packet application based on FPGA. By multiple channels working parallelly and sharing their rule tables, the throughput of the entire table is increased by about 1.5 times without storage replication. All matching results would be reordered to ensure the operation sequence and the constant processing latency in each pipeline. Moreover, it reduces the collision rate through multiple hash table blocks, and stores conflicted entries into auxiliary CAM tables.The implemented exact match table supports 200 million query operations per second, which is enough to support exceeding 100 Gbps throughput even for 64B packets.

Acknowledgments

This research was funded by National Key Research and Development Program of China: Software-defined interconnecting chip and supporting software kit development (Project.No. 2022YFB2901004).

References

[1] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," SIGCOMM Comput. Commun. Rev., vol.44, no.3, pp.87–95, July 2014.

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," ACM SIGCOMM Comput. Commun. Rev., vol.43, no.4, pp. 99–110, 2013.

[3] R. Shubbar and M. Ahmadi, "Fast 2D filter with low false positive for network packet inspection," IET Networks, vol.6, no.6, pp.224–231, 2017.

[4] P. Reviriego, G. Levy, M. Kadosh, and S. Pontarelli, "Algorithmic tcams: Implementing packet classification algorithms in hardware," IEEE Commun. Mag., vol.60, no.9, pp.60–66, 2022.

[5] M. Sha, Z. Guo, K. Wang, and X. Zeng, "A high-performance and accurate FPGA-based flow monitor for 100 Gbps networks," Electronics, vol.11, no.13, p.1976, 2022.

[6] M. Sha, Z. Guo, and M. Song, "A review of FPGA's application in high-speed network processing," J. Network New Media, vol.10, pp.1–11, 2021.

[7] M. Irfan, A.I. Sanka, Z. Ullah, and R.C. Cheung, "Reconfigurable content-addressable memory (CAM) ON FPGAs: A tutorial and survey," Future Generation Computer Systems, vol.128, pp.451–465, 2022.

[8] J. Carter and M. Wegman, "Universal classes of hash functions (extended abstract)," Proc. ninth Annual ACM Symposium on Theory of Computing, STOC'77, pp.106–112, 1977.

[9] R. Pagh and F.F. Rodler, "Cuckoo hashing," J. Algorithms, vol.51, no.2, pp.122–144, 2004.

[10] Y. Yang, S.R. Kuppannagari, A. Srivastava, R. Kannan, and V.K. Prasanna, "FASTHash: FPGA-based high throughput parallel hash table," Proc. 35th International Conference, ISC High Performance 2020, High Performance Computing, Frankfurt/Main, Germany, pp.3–22, Springer, June 2020.

[11] S. Pontarelli, P. Reviriego, and J.A. Maestro, "Parallel d-pipeline: A cuckoo hashing implementation for increased throughput," IEEE Trans. Comput., vol.65, no.1, pp.326–331, 2015.

[12] W.-Q. Wu, M.-T. Xue, T.-Q. Zhu, Z.-G. Ma, and F. Yu, "High-throughput parallel SRAM-based hash join architecture on FPGA," IEEE Trans. Circuits Syst. II, Exp. Briefs, vol.67, no.11, pp.2502–2506, 2020.

[13] M. Kekely and J. Korenek, "Mapping of P4 match action tables to FPGA," 2017 27th International Conference on Field Programmable Logic and Applications (FPL), IEEE, pp.1–2, 2017.

[14] D.-H. Le, K. Inoue, and C.-K. Pham, "Design a fast CAM-based exact pattern matching system on FPGA and 0.18 μm CMOS process," IEICE Trans. Fundamentals, vol.E96-A, no.9, pp.1883–1888, Sept. 2013.

[15] Z. István, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10 GBPS key-value stores on FPGAS," 2013 23rd International Conference on Field Programmable Logic and Applications, IEEE, pp.1–8, 2013.

[16] Y. Li, "Non-collision hash scheme using Bloom filter and CAM," 2009 Second Pacific-Asia Conference on Web Mining and Web-based Application, IEEE, pp.55–58, 2009.

[17] M. Kekely, L. Kekely, and J. Korenek, "Memory aware packet matching architecture for high-speed networks," 2018 21st Euromicro Conference on Digital System Design (DSD), IEEE, pp.1–8, 2018.

[18] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," IEEE Trans. Comput., vol.46, no.12, pp.1378–1381, 1997.

[19] H. Krawczyk, "LFSR-based hashing and authentication," Proc. 14th Annual International Cryptology Conference, Advances in Cryptology — CRYPTO'94, Santa Barbara, California, USA, pp.129–139, Springer, Aug. 1994.

[20] W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs," Architectures for Networking and Communications Systems, IEEE, pp.71–82, 2013.

[21] G.H. Gonnet, "Expected length of the longest probe sequence in hash code searching," J. ACM (JACM), vol.28, no.2, pp.289–304, 1981.

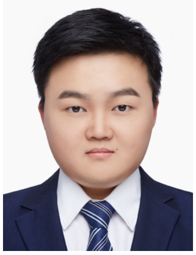
[22] Xilinx, "Alveo u200 and u250 data center accelerator cards data sheet (ds962)," Online, 2023, https://docs.xilinx.com/r/en-US/ds962-u200-u250

[23] Nvidia, "ConnectX-7 400G Adapters," Online, 2023, https://nvdam.widen.net/s/csf8rmnqwl.infiniband-ethernet-datasheet-connectx-7-ds-nv-us-2544471

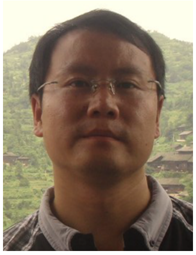
[24] Xilinx, "Alveo U25 Product Brief," Online, 2023, https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/alveo-u25-product-brief.pdf

[25] Xilinx, "Alveo SN1000 SmartNICs Data Sheet (DS989)," Online, 2023, https://docs.xilinx.com/v/u/en-US/ds989-sn1000

[26] M. Arumugam, D. Bansal, N. Bhatia, J. Boerner, S. Capper, C. Kim, S. McClure, N. Motwani, R. Narasimhan, U. Panchal, T. Pimpo, A. Premji, P. Shrivastava, and R. Tewari, "Bluebird: High-performance SDN for bare-metal cloud services," 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), Renton, WA, pp.355–370, USENIX Association, April 2022.



Xiaoyong Song received the B.S. degree from Beijing University of Technology, Beijing, China, in 2019. He is currently pursuing the doctor's degree with the School of Electronic, Electrical and Communication Engineering, University of Chinese Academy of Sciences, Beijing. His current research interest includes FPGA network acceleration, and matching table etc.



Zhichuan Guo received the B.S. degree from Wuhan University in 1996, and the Ph.D. degree from the University of Science and Technology of China in 2006. From 1996 to 2003, he served as an Electronics Engineer with the 13th Research Institute of China Electronics Technology Group Corporation and a SDH hardware R&D system engineer of optical networks at Huawei. In 2006, he joined with the Institute of Acoustics, Chinese Academy of Sciences, Beijing, China. Now he is a Professor of CAS engaging in field programmable gate array (FPGA)-based code acceleration, VLSI, and security.

Xinshuo Wang received the B.E. degree from Chongqing University of Posts and Telecommunications, Chongqing, China, in 2021. At present, he is studying for a doctorate degree in the school of electronic. electrical and communication engineering of the University of Chinese Academy of Sciences in Beijing, focusing on the field of FPGA network acceleration.

Mangu Song is currently working at the Institute of Acoustics, Chinese Academy of Sciences (IACAS), as a research assistant. She received her M.Sc degree in electronics and communication engineering from the School of Microelectronics, Chinese Academy of Science, Beijing, China in 2017. Her current research interests include FPGA-based code acceleration and network security.