High-performance architecture for flow-table lookup in SDN on FPGA

Rashid Hatami1 · Hossein Bahramgiri1

Published online: 2 January 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

We propose range-based ternary search tree (RTST), a tree-based approach for flow-table lookup in SDN. RTST builds upon flow-tables in SDN switches to provide a fast lookup among flows. We present a parallel multi-pipeline architecture for implementing RTST that benefits from high throughput and low latency. The proposed RTST and architecture achieve a memory efficiency of 1 byte of memory for each byte of flow. We also present a set of techniques to support dynamic updates. Experimental results reveal that RTST can be used to improve the performance of flow-lookup. It achieves a throughput of 670 million packets per second (MPPS), for a 1K 15-tuple flow-table, on a state-of-the-art FPGA (Virtex 6 XC6VLX760).

Keywords Software-defined networking (SDN) · Flow-table · Range-based ternary search tree (RTST) · Field-programmable gate array (FPGA) · Pipeline architecture

1 Introduction

Software-defined networking (SDN) defines a new paradigm that separates the control plane and data plane. Network-control applications, including routing, balancing traffic load, access control, and security policies, are defined as software in the control plane [1]. The control plane instructs data plane to forward the packets. The forwarding is determined by the rules that data plane applies to packets. The OpenFlow [2, 3] can be used as a viable interface for SDN to manage the network traffic between the control plane and data plane (network devices) [4]. A network device, specifically called the OpenFlow switch [5, 6] or switch, uses one or more flow-tables to execute its functionality. Lookup is one of the main functions on the flow-tables during which the switch identifies the corresponding rule for each incoming packet.
packet. It compares a prefix of the incoming packet fields with a table of stored rules (flow-table) and returns the matching rule.

To improve performance in SDN, the switch must provide a fast lookup on the flow-tables. Moreover, since switch memory may be needed to handle the always-growing number of services [7], it is important to consider a memory-efficient lookup solution. Current solutions can be classified into three categories: ternary content-addressable memory (TCAM) [8–10], bit vector (BV)-based approaches [11–14], and the decision tree approaches [15–18].

TCAM is not scalable with respect to flow-table size; also it is a scarce, expensive, and power-hungry resource [19, 20]. Furthermore, the time of adding new rules with random priority is significantly higher than with single priority [21]. The rules in TCAM are stored from top to bottom in decreasing priority, and for this reason, inserting a new TCAM entry usually entails rearranging the existing entries that yields high overhead for each update.

The BV approaches perform the lookup on each field individually and return the combination of the results by bit vectors or tables. These approaches focus on maximizing classification speed while sacrificing memory usage. It is difficult to implement a BV-based design for large rule sets in hardware because the size of bit vectors and tables grows with the size of rule sets.

Decision tree approaches partition the large search space into a small number of regions and construct a tree for each one. Decision tree performs the lookup by traversing from root to leaf. Unfortunately, to perform updates, often it will be necessary to reconstruct the tree because too many modifications are required for the decision tree. Also, decision tree incurs substantial cost regarding resources because there is a need to store child pointers at each node.

FPGA can perform reconfigurable architecture and offer abundant parallelism. Using FPGA makes it feasible to process scientific computing applications [22] and do parallel processing utilizing various types of strategies [23–25]. Moreover, a wide range of internet applications that require some hardware to perform frequent updates and adapt to different processing have been proposed on FPGA [26, 27]. However, the FPGA-based implementation requires some attention due to the limitation of the hardware resources and architectural optimization.

In this paper, we present RTST, a high-speed and memory-efficient lookup solution. RTST reduces the search space to achieve a fast lookup on the flow-tables. It takes advantage of memory efficiency using the pointer elimination technique. In addition, RTST needs not to be rebuilt for new updates, and therefore obtains a fast update. To implement RTST, we present a parallel multi-pipeline architecture on FPGA. This architecture is high throughput and low latency, and supports dynamic updates. The main contributions of this paper are as follows:

- We propose an RTST as a data structure to provide fast lookup on the flow-tables.
- We present an RTST-based pipeline architecture which benefits from high throughput even if the length of the packet header is scaled up.
- We employ an optimization technique to achieve the high memory efficiency of 1 byte of memory per byte of flow.
We present a set of techniques for supporting dynamic updates, including flow modification, deletion, and insertion.

The rest of the paper is organized as follows: Sect. 2 introduces the background and problem definition. Section 3 shows details of the RTST design. Section 4 describes the proposed architecture. Section 5 presents update techniques on this architecture. Section 6 describes our design scalability and the performance evaluations. Section 7 concludes the paper.

2 Background and problem definition

2.1 Background

An equivalent class of packets that includes a subset of packet header fields is denoted as a flow [1]. The ultimate target of the control plane is to specify flows and write in the switch’s flow-table through OpenFlow. The flow’s behaviors are determined by the control plane in the form of rules. A rule is matched with an incoming packet if all the header fields of the packet are matched with that rule. A set of actions will be taken when a packet matches a rule. The flow-table lookup is one of the packet processing functions in SDN used for services requirements such as quality of service.

2.2 Problem definition

The problem of flow-table lookup is defined as follows: Given that a flow-table \( F \) consists of \( N \) flows, a high-throughput lookup engine is designed which also supports memory efficiency and dynamic updates. It compares the incoming packet fields against the flow-table and returns the matching flow(s). Among all the matched flows, the highest priority flow is considered. In SDN, a large number of fields are required to be examined to support sophisticated network applications. For example, in the recent specification of the OpenFlow protocol [3], the number of fields is 15, which consists of 356 bits. Two kinds of matches are allowed in each field: prefix match and exact match. In the flow-table lookup, only the source address (SA) and destination address (DA) fields require the prefix match, while all the other fields require the exact match.

To achieve a high-performance lookup engine, there are three existing challenges that need to be resolved:

- **Large scale** Lookup upon a large number of flows in large-scale networks requires high processing.
- **Memory efficiency** To support larger flow-tables, memory-efficient lookup solutions are necessary.
– Dynamic update Flows may be dynamically changed and updated, and this affects the performance. Dynamic update carried out in the flow-table includes three operations: (1) modification, (2) deletion, and (3) insertion.

3 RTST design

In this section, we propose RTST, a special tree-based data structure that focuses on high memory efficiency and fast lookup. We consider RTST with order three because hardware implementation results show that orders greater than three are very difficult to be efficiently implemented. It inflicts poor uniformity and constructs a wide variety of fat nodes in the data structure. In this case, wide memory access is required for each fat node [18]. In addition, trees with order greater than three increase the number of bits for storing pointers. RTST comprises the following properties:

– The tree is a type of range-based ternary tree.
– The tree includes two different types of node: 3-node and leaf-node. All the nodes contain up to two data fields, left and right data field, where the value of the right data field is greater than the left one. Figure 1 shows the types of nodes in RTST.
– The 3-node references to three children including left, middle, and right, using pointers. The left child contains values less than the left data field, the middle child contains values between the two data fields, and the right child contains values greater than the right data field.
– All the leaf nodes contain one or two data fields and are at the lowest level.

A complete RTST is a tree in which all the levels are fully occupied except possibly the last level. In a complete RTST, each element can be found in \(1 + \lceil \log_3 N \rceil\), where \(N\) is the total number of nodes in the tree.

In the first step, the root of the tree, which consists of two values, must be determined. The locations of the root values can be calculated as follows:

![Leaf - node](image)

![3 - node](image)

**Fig. 1** Different types of nodes in RTST
where \( L_L \) is the location of the left data, and \( L_R \) is the location of the right data. After unveiling the root, three sub-trees, \( T_L, T_M, \) and \( T_R, \) are obtained. The \( T_L \) is the left sub-tree which has locations \( 0 \leq T_L < L_L, \) the \( T_M \) is the middle sub-tree which has locations \( L_L < T_M < L_R, \) and the \( T_R \) is the right sub-tree that has locations \( L_R < T_R \leq N. \) For each sub-tree, this procedure is repeated recursively. Let \( n \) be the number of levels of the RTST and \( \gamma \) be the number of nodes in the last level. The total number of nodes in each level, excluding the last level, is \( 3^{n-1} \) and \( \gamma = N - \sum_{i=1}^{n} 3^{i-1}. \) The steps to build an RTST are described in Algorithm 1.

**Algorithm 1 Complete RTST**

**Input:** List \( A \) consist of \( N \) sorted elements  
**Output:** Complete RTST

1. \( L_L = \left\lfloor \frac{1}{3} N \right\rfloor, \) \( L_R = \left\lfloor \frac{2}{3} N \right\rfloor \)
2. \( \text{Root} = [L_L, L_R] \)
3. \( \text{Pick elements} = (A[L_L], A[L_R]) \)
4. \( \text{Left sub-tree = complete RTST (left of } L_L \text{ sub-list);} \)
5. \( \text{Middle sub-tree = complete RTST (middle of } L_L \text{ and } L_R \text{ sub-list);} \)
6. \( \text{Right sub-tree = complete RTST (right of } L_R \text{ sub-list);} \)

We divide a flow-table into \( k \) groups to achieve significant performance, which will be described in Sect. 6.3. Two groups of flows are said to be disjoint if and only if any two flows in the groups do not overlap with each other. An RTST is built for each group of disjoint flows. The RTST need not be rebuilt for inserting new flows, and the flows can be added in the tree in any order. To add a new flow \( F \) into RTST, \( F \) is compared with node’s data fields and moved down the correct path while the leaf node is not reached. The following cases are defined to insert a new flow upon RTST:

- In the leaf node, if it has one data item, then insert \( F \) into the leaf node as the left or right data field.
- If the leaf node has two data items and the parent node has one data item, then compare the two data items of the leaf node with \( F. \) Afterward, select the middle value from the comparison and insert it in the parent node. If the leaf node is the left child, then move the data in the parent node to the right as the right data field and insert the middle value in the parent node as the left data field. If the leaf node is the right child, then the middle value is inserted in the parent node as the right data field, and the two leftover values become two leaf nodes.
- If the leaf and parent nodes are full, then develop the middle value of the leaf node to its parent node. Afterward, develop the middle value of the current node while the parent node is full and the current node is not root. If the current node
is root, then create a new root node from the middle value to its parent. Finally, update the left and right data fields of the new root.

Figure 2 shows a simple structure of RTST in which disjointed prefixes of a group are implemented. Depending on the comparison at each node, the search in the tree is performed. If the incoming prefix is smaller than the left data field, then the traversal forwards to the left child through the left pointer, and if the prefix is greater than the right data field, then it forwards to the right child through the right pointer and the traversal forwards to the middle child, otherwise. The comparison is begun from the root, and the decision regarding the traversing path is made based on the data values of each node. For example, assume that a packet with a prefix $P_p = 11,001,000$ arrives. $P_p$ is compared with node values 00,110,110 and 10,000,101, yielding no match and a “right” result. Therefore, the packet traverses to the right sub-tree. The comparison with the input value in this node leads to a “match” result.

### 3.1 Improving memory efficiency by pointer elimination

To build the RTST, each child pointer needs to be stored as a reference at each node. These pointers may incur cost regarding memory consumption. The memory efficiency can be improved if these pointers are eliminated.

To eliminate pointers, a logical relationship between parent and child nodes must be provided. The nodes in each level of the tree are accommodated in a contiguous memory block. Let $i$ denote the index of node $x$; then, $3i$ is the index of the left child of node $x$, and its middle child and right child are located at $3i + 1$ and $3i + 2$ indexes, respectively. The address of the next node could simply be calculated on the fly based on the current node address. Thus, there is no need to store pointers at each node. The memory allocation and address calculation are described in Fig. 3. Consider a node with address $A_{ij}$, where $i$ and $j$ are the level and the node number of tree, respectively; to access the left child, $(3 \times A_{ij})$ should be calculated, and $((3 \times A_{ij}) + 1)$ and $((3 \times A_{ij}) + 2)$ should be calculated in order to access the middle and right children, respectively.

![Fig. 2 A sample group of disjoint prefixes and its corresponding complete RTST](image.png)
Architecture

4.1 Overall architecture

Pipeline is an important technique to improve the performance of a design. In the ideal case, it takes one clock cycle for each operation. Throughput and delay are two criteria used for determining the performance of a pipeline. They are related but not identical. There are two approaches to increase the performance of pipeline architecture. The first one is to reduce the delay in each stage, while the second one is to increase the number of parallel operations in each stage. We employed both approaches to improve the throughput of our design. The overall architecture is shown in Fig. 4. In our multi-pipeline architecture, there are \( k \) pipelines, one for each group of the flow-table. To implement RTST on the pipeline, each level of the tree is mapped upon a separate pipeline stage. The number of pipeline stages is determined by the height of the tree, and the height is dependent on the number of elements of each group.
4.2 RTST architecture

The block diagrams of the architecture and a single stage of the pipeline are shown in Fig. 5. The pipeline architecture is dual line where each stage includes logical circuits and a memory block. Logical circuits are applied to perform the comparison operation and send the result to the next stage. The memory block is dual port so that two data can be read/write every clock cycle. In each stage of the pipeline, three data are forwarded from the previous stage: (1) input prefix (which is compared with the content of each entry in the memory to determine the matching status), (2) memory address, and (3) ready signal. The forwarded memory address is used to retrieve the content of each entry in the memory that is the left and right data of the node. Note that the memory access time is a major reason for the delay on each stage. It can affect the total performance. In our design, each entry in the memory includes two elements which are available simultaneously; hence, the access time delay will be decreased. When matching is made, the ready signal is set, and the memory access in the stages is turned off. Figure 6 shows the block diagram of the next address generator. Since the next address is calculated in parallel with memory access, there is no further delay in the address generation. Depending on the output of the comparator, the priority encoder provides four possible states including (I) $3 \times A_i$, if the input prefix is smaller than the left data, (II) $3 \times A_i + 1$, when the input prefix is between

![RTST Pipeline Stages]

Fig. 5 The proposed architecture for implementing RTST
the left and right data, (III) $3 \times A_i + 2$, provided that the input prefix is greater than the right data, (IV) \textit{Next hop}, if the match is made either with the left or right data. In this case, the ready signal is established to refer to the next stage. When a match is made, searching in the subsequent stages is not necessary, and the previous result can simply be forwarded.

5 Dynamic updates

Since updates and configurations in flow-tables occurred frequently, it requires the hardware to adapt to frequent updates during runtime. Given a flow-table consisting of $N$ flow entries $\{F_i|i = 0, 1, \ldots, N - 1\}$, the updates include three operations: (1) modification, (2) deletion, and (3) insertion.

- \textit{Modification} The primary purpose of the modification is to change the flow information in the flow-tables. For the modification requests, if a matching entry exists in the flow-table, the fields of this entry are updated with the new value from the request. Given a flow $F_i$, search $F_i$ in the flow-table, locate $ie\{0, 1, \ldots, N - 1\}$ where $F_i = F$; change flow $F_i$ in the flow-table into flow $F$.

- \textit{Deletion} The primary purpose of deletion is to remove the flows in the flow-tables. For the deletion requests, if a matching entry exists in the table, it must be deleted. Given a flow $F_i$, search $F_i$ in the flow-table, locate $ie\{0, 1, \ldots, N - 1\}$ where $F_i = F$; remove $F_i$ completely from the flow-table.
– *Insertion* The primary purpose of insertion is to add the flows in the flow-tables. Two flow entries are overlapped if a single packet match both, and both entries have the same priority. If an overlapping conflict exists, the insertion must be rejected. In the non-overlapping case, insert the flow entry in the flow-table. Given a flow $F$, search $F$ in the flow-table, if $\forall i \in \{0, 1, \ldots, N-1\}, F_i \neq F$; then insert $F_i$ into the flow-table.

The first step of the update operations is always searching $F$, which reports whether $F$ exists in the current flow-table. After searching $F$ is completed, we present our main ideas as follows (Fig. 7).

### 5.1 Modification

Assume $F$ exists in the flow-table; hence, $\exists i \in \{0, 1, \ldots, N-1\}$ such that $F_i = F$. Flow modification can be performed by inserting a write bubble, as introduced in [28]. This is illustrated in Fig. 8. There is one write bubble in each stage that stores the update information. Each write bubble consists of two entries. Each entry includes: (1) the memory address to be updated in the next stage, (2) the new content, and (3) a write enable bit. When a flow modification is initiated, the memory content of the write bubble in each stage is modified, and a write bubble is inserted into the pipeline. The write bubble will modify the memory location in the next stage when the
write enable bit is set. Since the memory is dual-ported, two nodes can be simultaneously modified at each stage.

5.2 Deletion

Assume \( F \) exists in the table; hence, \( \exists i \in \{0, 1, \ldots, N - 1\} \) such that \( F_i = F \). For deletion, we consider a valid bit (flag) to keep the validity of the nodes. In this case, a node is valid only if its corresponding valid bit is set to ‘1’, and is invalid if its corresponding valid bit is reset to ‘0’. To delete a flow, we reset its corresponding valid bit to ‘0’. An invalid flow is not compared for any match result. The left part of Fig. 7 depicts the tree in which only \( F_8 \) is initially invalid. If we want to delete \( F_4 \) and \( F_7 \), the valid bits corresponding to \( F_4 \) and \( F_7 \) are reset to ‘0’. The right part of Fig. 7 shows the reset process of their corresponding valid bits.

5.3 Insertion

Assume that the flow \( F \) does not exist in the table; hence, \( \forall i \in \{0, 1, \ldots, N - 1\}, F_i \neq F \). For insertion, we insert a new flow by modifying an invalid node or adding a new node. In other words, we replace invalid flows by the new flows. If there is no invalid node, a new node is created upon RTST. The algorithm used to insert upon RTST follows a bottom-up approach. The first step is to find the parent \( p \) of the newly node \( n \) that will be inserted on RTST. There are two cases: (1) The node \( p \) has only one or two children (the parent node has one or two invalid children) and (2) the node \( p \) has three children (the parent node does not have invalid child). In the first case, the valid bit of a child node is set to ‘1’ and \( n \) is inserted as the appropriate child of \( p \). In the second case, if \( n \) is inserted as the appropriate child of \( p \), \( p \) has four children, thus violates the property of an RTST. Hence, an internal node \( m \) is created according to the mechanism of adding a new flow, presented in Sect. 3.

6 Performance evaluation

6.1 Scalability

A scalable architecture sustains high levels of performance even if the number of the packet fields and flow-table size are scaled up [12]. Our design benefits from high clock rate even if the length of each packet header is scaled up. Moreover, it is a fast lookup engine, even when the number of flows is increased. Since each level of the tree is mapped onto one stage of the pipeline, the number of stages is determined by the height of the tree. It has never a height greater than \( \lceil \log_3 N \rceil \). We can expect considerable resource saving in hardware implementation because the storage complexity is linear, implementation is simple, and the amount of logic resources is small.
6.2 Experimental setup

We conducted experiments to evaluate the performance of our design, including the FPGA prototype of the architecture. The architecture has been implemented in VHDL, using Xilinx ISE Design Suite 14.2, targeting the Virtex 6 XC6VLX760 FFG1760-2 FPGA [29]. The chip contains 118, 560 logic slices, 1200 I/O pins, 26 Mb BRAM (720 RAMB 36 blocks). A configurable logic block (CLB) on this FPGA consists of 2 slices, each having 4 LUTs and 8 flip-flops. Throughput and latency are reported using post-place-and-route results. We also generated random flows for OpenFlow-tables \( (d = 15, L = 356) \), in order to prototype our design, where \( d \) is the total number of fields for the flow and \( L \) is the total number of bits for the input packet header. The number of flows in a flow-table is chosen from 128 to 1K.

6.3 Scalability of latency

Latency is the processing delay of a single packet when no dynamic update is performed. Figure 9 shows the effect of various sizes of flow-tables \( (N = 128, 256, 512 \) and \( 1024) \) on the latency performance of our architecture. Experimental results show that our design achieves low latency for various values of \( N(L = 356) \). We used the prefix partitioning [18] to divide a given flow-table into \( k \) groups of disjoint flows. We store each group of flows in a balanced RTST leading to \( O(\log_3 N') \) lookup time per group, where \( N' \) is the number of flows in each group. Since increasing the number of groups can be inefficient in hardware implementation, we can achieve an acceptable number of groups by \( h = \log_3(\frac{N}{k}) \), where \( h \) is the height of the search tree. As can be seen, compared to the no prefix partitioning implementation, the prefix partitioning approach improves the latency performance in our architecture.

![Latency for each N = 128, 256, 512, 1024](image)

**Fig. 9** Latency for each \( N = 128, 256, 512, 1024 \)
The factors that lead to this result are the use of fewer trees and the reduction in the height of the tree.

Figure 10 shows the effect of the length of the packet header ($L = 104$ and $L = 356$) on the latency performance of our design. As it is shown, the clock rate for both $L = 104$ and $L = 356$ is high, and the difference in the latency is not large. Our experimental results show that the clock rate achieved for our design is 337 MHz that can provide a scalable and high-throughput architecture on FPGA.

### 6.4 Comparison of the throughput and latency with the state of the art

We compare the throughput and latency performance of our approach with the existing hardware accelerators in Fig. 11. All the implementations and our approach support 1K flows, and each flow includes 15 fields. The TCAM can match packets in a single clock cycle [30]. As a result, the throughput and latency performance of TCAM are estimated based on the assumption that packets can be matched within a single clock cycle and clock rate can be at 360 MHz. In the FSBV [13], it is assumed that 7 pipeline stages are employed, and the clock rate can be sustained at 167 MHz. We assume a single pipeline is employed in the StrideBV [14], which consists of 89 stages running at 105 MHz. For the decision-tree-based implementation [17], we assume 16-stage pipeline is used, and clock rate can be at 125 MHz. In the BV-based pipelined architecture [12], the 2-dimensional pipelined architecture is employed, which for a flow-table consisting of $N$ flows and an $L$-bit packet header, requires $N$ rows and $L$ columns while the clock rate can be sustained at 324 MHz. As can be seen, the StrideBV and BV-based pipelined architecture have higher latency than other designs. These results are due to the deeply pipelined architecture. Our design achieves better latency compared to the StrideBV, decision tree, and BV-based pipelined.
architecture. Our architecture achieves high clock rate (337 MHz) since the processing delay in each stage of the pipeline is low. It also benefits from high throughput (674 MPPS) compared to the other designs.

6.5 Memory efficiency

The memory efficiency is defined as the amount of memory required to store one flow entry. We compared our design with TCAM [30], FSBV [13], and StrideBV [14] with respect to the average memory requirement per flow in Table 1. The memory consumption of the BV-based pipelined architecture [12] is unclear, but usually, the BV algorithms can provide high throughput at the cost of low memory efficiency. As can be seen, our design consumed 44.5 bytes per flow, achieved a memory efficiency of 1 byte of memory for each byte of flow. TCAM also consumes 44.5 bytes per flow, but is not scalable in terms of circuit area, as compared to SRAMs [31].

| Approach       | Memory req.   | Memory req.   |
|----------------|---------------|---------------|
|                | $L = 104$ bits | $L = 356$ bits |
| TCAM [30]      | 13 bytes/flow | 44.5 bytes/flow |
| FSBV [13]      | 29 bytes/flow | 99.23 bytes/flow |
| Stride BV [14] | 156 bytes/flow | 534 bytes/flow |
| Our approach   | 13 bytes/flow | 44.5 bytes/flow |
7 Conclusion

In this paper, we proposed RTST as a data structure to achieve fast lookup for the flow-tables. We also presented a parallel multi-pipeline architecture to sustain both high throughput and low latency. This architecture provides high clock rate and fast lookup, as well as supporting dynamic updates. The proposed architecture is simple, and resource consumption is small. We could also achieve a memory efficiency of 1 byte of memory for each byte of the flow entry. Experimental results revealed that for a 1 K 15-tuple flow-table, a state-of-the-art FPGA could sustain a throughput of 670 MPPS. With these advantages, our algorithm and design can be used to improve the performance of flow-lookup.

References

1. Wang H, Qian C, Yu Y, Yang H, Lam SS, Wang H, Qian C, Yu Y, Yang H, Lam SS (2017) Practical network-wide packet behavior identification by AP classifier. IEEE/ACM Trans Netw (TON) 25(5):2886–2899
2. McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J (2008) OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput Commun Rev 38(2):69–74
3. OpenFlow Switch Specification V1.3.1. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onspecifications/openflow/openflow-spec-v1.3.1.pdf
4. Chang S-H, Mao-Sheng H (2017) A novel software-defined wireless network architecture to improve ship area network performance. J Supercomput 73(5):2886–2899
5. Kang S, Yoon W (2016) SDN-based resource allocation for heterogeneous LTE and WLAN multi-radio networks. J Supercomput 72(4):1342–1362
6. Tu R, Wang X, Yang Y (2014) Energy-saving model for SDN data centers. J Supercomput 70(3):1477–1495
7. Reitblatt M, Canini M, Guha A, Foster N (2013) Fattire: declarative fault tolerance for software-defined networks. In: Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN ’13, (New York, NY, USA), ACM, pp 109–114
8. Yu F, Katz RH, Lakshman T (2005) Efficient multimatch packet classification and lookup with TCAM. Micro IEEE 25(1):50–59
9. Lakshminarayanan K, Rangarajan A, Venkatachary S (2005) Algorithms for advanced packet classification with ternary CAMs. ACM SIGCOMM Comput Commun Rev 35(4):193–204
10. Vamanan B, Vijaykumar TN (2011) Treecam: decoupling updates and lookups in packet classification. In: Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologies, CoNEXT ’11, (New York, NY, USA), ACM, pp 27:1–27:12
11. Zhou S, Qu Y, Prasanna V (2014) Multi-core implementation of decomposition-based packet classification algorithms. J Supercomput 69(1):34–42
12. Qu YR, Prasanna VK (2016) High-performance and dynamically updatable packet classification engine on FPGA. IEEE Trans Parallel Distrib Syst 27:197–209
13. Jiang W, Prasanna VK (2009) Field-split parallel architecture for high performance multi-match packet classification using FPGAs. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA ’09, (New York, NY, USA), ACM, pp 188–196
14. Ganegedara T, Prasanna VK (2012) StrideBV: single chip 400G+ packet classification. In: 2012 IEEE 13th International Conference on High Performance Switching and Routing, pp 1–6
15. Gupta P, McKeown N (2000) Classifying packets with hierarchical intelligent cuttings. IEEE Micro 20:34–41
16. Singh S, Baboescu F, Varghese G, Wang J (2003) Packet classification using multidimensional cutting. In: Proceedings of the 2003 Conference on Applications, Technologies, Architectures,
and Protocols for Computer Communications, SIGCOMM ’03, (New York, NY, USA), ACM, pp 213–224

17. Jiang W, Prasanna VK (2012) Scalable packet classification on FPGA. IEEE Trans Very Large Scale Integr (VLSI) Syst 20(9):1668–1680

18. Le H, Prasanna VK (2012) Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning. IEEE Trans Comput 61(7):1026–1039

19. Meiners CR, Liu AX, Torng E (2007) TCAM Razor: a systematic approach towards minimizing packet classifiers in TCAMS. In: 2007 IEEE International Conference on Network Protocols, pp 266–275

20. Dong X, Qian M, Jiang R (2018) Packet classification based on the decision tree with information entropy. J Supercomput, 1–15

21. Jin X, Liu HH, Gandhi R, Kandula S, Mahajan R, Zhang M, Rexford J, Wattenhofer R (2014) Dynamic scheduling of network updates. In: Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM ’14, (New York, NY, USA), ACM, pp 539–550

22. Thapliyal H, Jayashree H, Nagamani A, Arabnia HR (2013) Progress in reversible processor design: a novel methodology for reversible carry look-ahead adder. In: Transactions on Computational Science XVII, Springer, pp 73–97

23. Arabnia HR, Oliver MA (1986) Fast operations on raster images with SIMD machine architectures. In: Computer graphics forum, vol 5. Wiley, New York, pp 179–188

24. Arabnia HR, Taha TR (1998) A parallel numerical algorithm on a reconfigurable multi-ring network. Telecommun Syst 10(1–2):185–202

25. Yang MQ, Athey BD, Arabnia HR, Sung AH, Liu Q, Yang JY, Mao J, Deng Y (2009) High-throughput next-generation sequencing technologies foster new cutting-edge computing techniques in bioinformatics. BMC Genom 10(S1):I1

26. Bonesana I, Paolieri M, Santambrogio MD (2008) An adaptable FPGA-based system for regular expression matching. In: 2008 Design, Automation and Test in Europe, pp 1262–1267

27. Frigerio L, Marks K, Krikelis A (2010) Timed coloured petri nets for performance evaluation of DSP applications: the 3GPP LTE case study. In: Piguet C, Reis R, Soudris D (eds) VLSI-SoC: design methodologies for SoC and SiP. Springer, Berlin, pp 114–132

28. Basu A, Narlikar G (2005) Fast incremental updates for pipelined forwarding engines. IEEE/ACM Trans Netw (TON) 13(3):690–703

29. Virtex-6 FPGA family. http://www.xilinx.com/products/virtex6

30. Zane F, Narlikar G, Basu A (2003) CoolCAMs: power-efficient TCAMS for forwarding engines. Proc IEEE INFOCOM 1:42–52

31. Jiang W, Prasanna VK (2009) Large-scale wire-speed packet classification on FPGAs. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA ’09, (New York, NY, USA), ACM, pp 219–228