Decision Forest: A Scalable Architecture for Flexible Flow Matching on FPGA

Weirong Jiang*, Viktor K. Prasanna*
*Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
Email: {weirongj, prasanna}@usc.edu

Norio Yamagaki†
†System IP Core Research Laboratories
NEC Corporation
Kawasaki, Kanagawa, Japan
Email: n-yamagaki@cj.jp.nec.com

***Abstract*—Next generation Internet requires processing rich and flexible flow information in the network infrastructure. Rapid growth in network traffic results in major challenge to support flexible flow matching at line rate. Most of the existing work focuses on functionality rather than performance, and simply adopts either power-hungry TCAM or performance-indeterministic hashing. This paper exploits the abundant parallelism and other desirable features provided by state-of-the-art FPGAs, and proposes a parallel architecture, named *decision forest*, for high-performance flexible flow matching. We develop a framework to partition a given table of flexible flow rules into multiple subsets each of which is built into a depth-bounded decision tree. The partitioning scheme is carefully designed to reduce rule duplication during the construction of the decision trees. Thus the overall memory requirement is significantly reduced. After such partitioning, the number of header fields used to build the decision tree for each rule subset is small. This leads to reduction in logic resource requirement. Exploiting the dual-port RAMs available in current FPGAs, we map each decision tree onto a linear pipeline to achieve high throughput. Our extensive experiments and FPGA implementation demonstrate the effectiveness of our scheme. Our design supports 1K flexible flow rules while sustaining 40 Gbps throughput for matching minimum size (40 bytes) packets. To the best of our knowledge, this is the first FPGA design for flexible flow matching to achieve over 10 Gbps.**

***Keywords*-flexible flow matching; FPGA; OpenFlow;**

I. Introduction

Emerging network requirements such as user-level and fine-grained security, mobility and reconfigurability have made network virtualization an essential feature for next-generation enterprise, data center and cloud computing networks. Major router vendors have recently initiated programs to provide open router platforms which allow users to develop software extensions for proprietary hardware [1]. This requires the underlying forwarding hardware be flexible and provide clean interface for software [2]. One such effort is the OpenFlow switch which manages explicitly the network flows using a flow table with rich definition as the software-hardware interface [3]. Most of the existing work in developing flexible forwarding hardware is focused on the functionality and simply adopts ternary content addressable memory (TCAM) or various hashing schemes. However, TCAMs are power-hungry, and do not scale well with respect to area and clock rate [4]. The power consumption per bit of TCAMs is 150 times that for static random access memories (SRAMs) [5]. Hashing cannot provide deterministic performance due to potential collision and is inefficient in handling wildcard or prefix matching [4]. While rapid growth of the Internet has posed great challenges on next generation network infrastructure for its performance including throughput and power consumption, few efforts have been made for flexible forwarding hardware to address these challenges. On the other hand, FPGA technology has become an attractive option for implementing real-time network processing engines [4], [6], due to its ability to reconfigure and to offer abundant parallelism. State-of-the-art SRAM-based FPGA devices such as Xilinx Virtex-6 and Altera Stratix-IV provide high clock rate, low power dissipation and large amounts of on-chip dual-port memory with configurable word width.

The kernel operation in flexible forwarding hardware is matching each packet against a table of flow rules. The definition of a flow rule can be as flexible as users want. Flexible flow matching can be viewed as an extension of the traditional five-field packet classification [4]. Taking the recent OpenFlow specification as an example, with more packet header fields to be matched, the total number of bits per packet for lookup increases from 104 to over 237 [3]. We adopt decision-tree-based algorithms, which are considered among the most scalable packet classification algorithms [4], [5]. However, existing decision-tree-based packet classification algorithms use all of the packet header fields to construct the tree. This results in large memory and resource requirements for flexible flow matching where different flow rules in a table specify few but different header fields. Moreover, the depth of a decision tree can be very large for flexible flow matching due to the increase in the number of header fields to be matched.

We propose a framework to partition a table of flow rules into multiple subsets so that each subset uses a small number of header fields to build a decision tree of bounded depth. We call the corresponding architecture the *decision forest*

This work is supported by the United States National Science Foundation under grant No. CCF-0702784. Equipment grant from Xilinx Inc. is gratefully acknowledged.

978-0-7695-4179-2/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.83

IEEE computer society

-based architecture where multiple (heterogeneous) decision trees are searched in parallel. High throughput is achieved by mapping each decision tree onto a linear pipeline. FPGA implementation results show that our design supports 1K OpenFlow-like flexible flow rules and sustains 40 Gbps throughput for matching minimum size (40 bytes) packets.

The rest of the paper is organized as follows. Section II introduces the OpenFlow switch as an example of flexible flow matching. Section III presents our decision forest architecture. Section IV evaluates the performance of the algorithms and the FPGA implementation. Section V concludes the paper.

II. Background

As the flexible forwarding hardware is recently proposed [2], most of existing work focuses on the functionality rather than performance. Few efforts have been made in exploiting the power of state-of-the-art FPGA technology to achieve high-performance flexible flow matching.

A. OpenFlow Switching

The recently proposed OpenFlow switch [3] brings programmability and flexibility to the network infrastructure by separating the control and the data planes of routers / switches, and managing the control plane at few centralized servers. The major processing engine in the OpenFlow switch is flexible flow matching, where up to 12-tuple header fields of each packet are matched against all the flow rules [3]. Table I shows a simplified example of OpenFlow rule table. The 12-tuple header fields supported in the current OpenFlow specification include the ingress port [1], 48-bit source / destination Ethernet addresses, 16-bit Ethernet type, 12-bit VLAN ID, 3-bit VLAN priority, 32-bit source / destination IP addresses, 8-bit IP protocol, 6-bit IP Type of Service (ToS) bits, and 16-bit source / destination port numbers [3]. Each field of a flow rule can be specified as an exact number or a wildcard. IP address fields can also be specified as a prefix. In the following discussion, we let SA / DA denote the source / destination IP addresses and SP / DP the source / destination port numbers. And we have following definitions:

- *Simple rule* is the flow rule of which all the fields are specified as exact values, e.g. R10 in Table I.
- *Complex rule* is the flow rule containing wildcards or prefixes, e.g. R1∼9 in Table I.

A packet is considered matching a rule if and only if its header content matches all the specified fields within that rule. If a packet matches multiple rules, the matching rule with the highest priority is used. In OpenFlow, a simple rule always has the highest priority. If a packet does not match any rule, the packet is forwarded to the centralized server. The server determines how to handle it and may register a new rule in the switches. Hence dynamic rule updating needs to be supported.

While OpenFlow switch technology is evolving, little attention has been paid on improving the performance of flow matching. Luo et al. [7] propose using network processors to accelerate the OpenFlow switching. Similar to the software implementation of the OpenFlow switching, hashing is adopted for simple rules while linear search is performed on the complex rules. When the number of complex rules becomes large, using linear search leads to low throughput. Naous et al. [6] implement the OpenFlow switch on NetFPGA which is a Xilinx Virtex-2 Pro 50 FPGA board tailored for network applications. A small TCAM is implemented on FPGA for complex rules. Due to the high cost to implement TCAM on FPGA, their design can support no more than few tens of complex rules. Though it is possible to use external TCAMs for large rule tables, high power consumption of TCAMs remains a big challenge. To the best of our knowledge, none of existing schemes for OpenFlow-like flexible flow matching can support more than hundreds of complex rules while sustaining throughput above 10 Gbps in the worst case where packets are of minimum size i.e. 40 bytes.

[1]The width of the ingress port is determined by the number of ports of the switch / router. For example, 6-bit ingress port indicates that the switch / router has up to 63 ports.

Table I
EXAMPLE RULE SET. (ETHERNET SRC/DST: 16-BIT; SA/DA:8-BIT; SP/DP: 4-BIT)

Rule	Ingress port	Ethernet src	Ethernet dst	Ethernet type	VLAN ID	VLAN priority	IP src (SA)	IP dst (DA)	IP Protocol	IP ToS	Port src (SP)	Port dst (DP)	Action
R1	*	00:13	00:06	*	*	*	*	*	*	*	*	*	act0
R2	*	00:07	00:10	*	*	*	*	*	*	*	*	*	act0
R3	*	*	00:FF	*	*	*	*	*	*	*	*	*	act1
R4	*	00:1F	*	0x8100	100	5	*	*	*	*	*	*	act1
R5	*	*	*	0x0800	*	*	*	01*	*	*	*	*	act2
R6	*	*	*	0x0800	*	*	001*	11*	TCP	*	10	15	act0
R7	*	*	*	0x0800	*	*	001*	11*	UDP	*	2	11	act3
R8	*	*	*	0x0800	*	*	100*	110*	*	*	5	6	act1
R9	5	00:FF	00:00	0x0800	4095	7	0011*	1100*	TCP	0	2	5	act0
R10	1	00:1F	00:2A	0x0800	4095	7	01000001	10100011	TCP	0	2	7	act0

B. Revisiting Packet Classification Schemes

Flexible flow matching can be viewed as an extension from five-field packet classification whose solutions have been extensively studied in the past decade. Comprehensive surveys for packet classification algorithms can be found in [5]. Among the existing packet classification engines, decision-tree-based solutions are considered the most scalable with respect to memory requirement [4], [5]. Traversal of the tree can be pipelined to achieve high throughput [4].

Decision-tree-based algorithms (e.g. HyperCuts [8]), take the geometric view of the packet classification problem. Each rule defines a hypercube in a D-dimensional space where D is the number of header fields considered for packet classification. Each packet defines a point in this D-dimensional space. The decision tree construction algorithm employs several heuristics to cut the space recursively into smaller subspaces. Each subspace ends up with fewer rules. The cutting process is performed until the number of rules contained by a subspace is small enough to allow a low-cost linear search to find the best matching rule. Such algorithms scale well and are suitable for rule sets where the rules have little overlap with each other. But they suffer from rule duplication which can result in $O(N^D)$ memory explosion in the worst case, where N denotes the number of rules. Moreover, the depth of a decision tree can be as large as $O(W)$, where W denotes the total number of bits per packet for lookup. $D = 12$, $W > 237$ in OpenFlow.

III. Decision Forest Framework

We aim to apply decision-tree-based algorithms to flexible flow matching while addressing their drawbacks, i.e. memory explosion and large tree depth.

A. Heuristic

We observe that different complex rules in a flexible flow table may specify only a small number of fields while leaving other fields to be wildcards. This phenomenon is fundamentally due to the concept of flexible forwarding hardware which was proposed to support various applications on the same substrate. For example, both IP routing and Ethernet forwarding can be implemented in OpenFlow. IP routing will specify only the destination IP address field while Ethernet forwarding will use only the destination Ethernet address.

B. Motivation

The memory explosion for decision-tree-based algorithms in the worst case has been identified as a result of rule duplication [4]. A less specified field is usually easier to cause rule duplication. For the example of OpenFlow table shown in Table I, if we consider only SA and DA fields, all the 10 rules can be represented geometrically on a 2-dimensional space shown in Figure 1. Decision-tree-based algorithms such as HyperCuts [8] cut the space recursively based on the values from SA and DA fields. As shown in Figure 1, no matter how to cut the space, R1∼4 will be duplicated to all children nodes. This is because their SA / DA fields are wildcards, i.e. not specified. Similarly, if we build the decision tree based on source / destination Ethernet addresses, R5∼8 will be duplicated to all children nodes, no matter how the cutting is performed. We will see in Section IV that the characteristics of flexible flow rules (discussed in Section III-A) cause severe memory explosion when the rule set becomes larger.

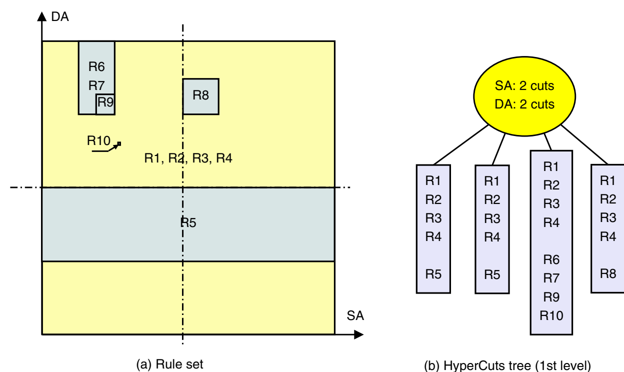


Figure 1. Rule duplication in HyperCuts tree.

Hence an intuitive idea is to split a table of complex rules into different subsets. The rules within the same subset specify nearly the same set of header fields. For each rule subset, we build the decision tree based on the specified fields used by the rules within this subset. For instance, the example rule table can be partitioned into two subsets: one contains R1∼4 and the other contains R5∼10. We can use only source / destination Ethernet addresses to build the decision tree for the first subset while only SA / DA fields for the second subset. As a result, the rule duplication will be dramatically reduced. Meanwhile, since each decision tree after such partitioning employs a much smaller number of fields than the single decision tree without partitioning, we can expect considerable resource savings in hardware implementation.

C. Algorithms

We develop the decision forest construction algorithms to achieve the following goals:

- Reduce the overall memory requirement.
- Bound the depth of each decision tree.
- Bound the number of decision trees.

Rather than perform the rule set partitioning and the decision tree construction in two phases, we combine them efficiently as shown in Algorithm 1. The rule set is partitioned dynamically during the construction of each decision tree. The function for building a decision tree i.e. $BuildTree(.)$ is shown in Algorithm 2.

Algorithm 1 Building the decision forest

Require: Rule set R.
Require: Parameters: $bucketSize$, $depthBound$, P.
Ensure: Decision forest: $\{T_i | i = 0, 1, \cdots, P-1\}$.

```
1: i ← 0, R_i ← R and split ← TRUE.
2: while i < P do
3:    if i == P − 1 then {The last subset / tree}
4:       split ← FALSE
5:    end if
6:    {T_i, R_{i+1}} ← BuildTree (R_i, split, bucketSize,
      depthBound)
7:    i ← i + 1
8: end while
```

Algorithm 2 Building the decision tree and the split-out set: $\{T, R_{ex}\} \leftarrow BuildTree(R, split, bucketSize, depthBound)$

Require: Rule set R.
Require: Parameters: $split$, $bucketSize$, $depthBound$.
Ensure: Decision tree T and the split-out set R_{ex}.

```
1: Initialize the root node: root.rules ← R.
2: Push root into nodeList.
3: while nodeList ≠ null do
4:    n ← Pop(nodeList)
5:    if n.numrules < bucketSize then
6:       n is a leaf node. Continue.
7:    end if
8:    if n.depth == depthBound then
9:       Assign to n the bucketSize most specified rules
         from n.rules. Push remaining rules of n.rules into
         R_ex. n is a leaf node. Continue.
10:   end if
11:   for f ∈ OptFields(n) do
12:      nCuts[f] ← OptNumCuts(n, f)
13:      n.numCuts *= nCuts[f]
14:   end for
15:   if split is TRUE then
16:      r ← PotentialDuplicatedRule(n, nCuts)
17:      Push r into R_ex.
18:   end if
19:   for i ← 0 to 2^{n.numCuts} − 1 do
20:      n_i ← CreateNode(n, nCuts, i)
21:      Push n_i into nodeList.
22:   end for
23: end while
```

The parameter P bounds the number of decision trees in a decision forest. We have the rule set R_i to build the ith tree whose construction process will split out the rule set R_{i+1}. $i = 0, 1, \cdots, P-1$. In other words, the rules in $R_i - R_{i+1}$ are actually used for building the data structure of the ith tree. The parameter $split$ determines if the rest of the rule set will be partitioned. When building the last decision tree ($i = P - 1$), $split$ is turned to be FALSE so that all the remaining rules are used to construct the last tree. Other parameters include $depthBound$ which bounds the depth of each decision tree, and $bucketSize$ which is inherited from the original HyperCuts algorithm to determine the maximum number of rules allowed to be contained in a leaf node.

Algorithm 2 is based on the original HyperCuts algorithm, where Lines 8∼10 and 15∼18 are the major changes. Lines 8∼10 are used to bound the depth of the tree. After determining the optimal cutting information (including the cutting fields and the number of cuts on these fields) for the current node (Lines 11∼14), we identify the rules which may be duplicated to the children nodes (by the $PotentialDuplicatedRule()$ function). These rules are then split out of the current rule set and pushed into the split-out rule set R_{ex}. The split-out rule set will be used to build the next decision tree(s). The rule duplication in the first $P-1$ trees will thus be reduced.

D. Architecture

To achieve line-rate throughput, we map the decision forest including P trees onto a parallel multi-pipeline architecture with P linear pipelines, as shown in Figure 2 where $P = 2$. Each pipeline is used for traversing a decision tree as well as matching the rule lists attached to the leaf nodes of that tree. The pipeline stages for tree traversal are called the *tree* stages while those for rule list matching are called the *rule* stages. Each tree stage includes a memory block storing the tree nodes and the cutting logic which generates the memory access address based on the input packet header values. At the end of tree traversal, the index of the corresponding leaf node is retrieved to access the rule stages. Since a leaf node contains a list of $bucketSize$ rules, we need $bucketSize$ rule stages for matching these rules. All the leaf nodes of a tree have their rule lists mapped onto these $bucketSize$ rule stages. Each rule stage includes a memory block storing the full content of rules and the matching logic which performs parallel matching on all header fields.

Each incoming packet goes through all the P pipelines in parallel. A different subset of header fields of the packet may be used to traverse the trees in different pipelines. Each

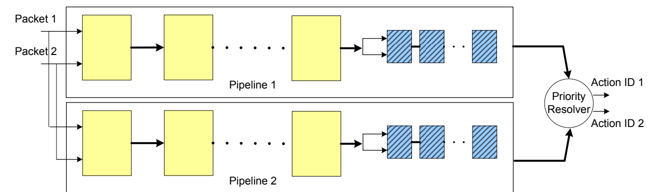


Figure 2. Multi-pipeline architecture for searching the decision forest (P = 2). The shaded blocks (i.e. rule stages) store the leaf-level rule lists while the tree nodes are mapped onto plain-color blocks (i.e. tree stages).

pipeline outputs the rule ID or its corresponding action. The priority resolver picks the result with the highest priority among the P outputs from the P pipelines. It takes $H + bucketSize$ clock cycles for each packet to go through the architecture, where H denotes the number of tree stages.

We adopt the similar scheme as [4] to map tree nodes onto pipeline stages while managing the memory distribution across stages. The cutting logic is generated based on the cutting information obtained from the tree construction process (Algorithm 2). To further improve the throughput, we exploit the dual-port RAMs provided by state-of-the-art FPGAs so that two packets are processed every clock cycle.

As in [4], our architecture supports dynamic rule updates by inserting *write bubbles* into the pipeline. Since the architecture is linear, all packets preceding or following the write bubble can perform their operations while the write bubble performs an update.

IV. Experimental Results

We conducted extensive experiments to evaluate the performance of our decision forest -based schemes including the algorithms and FPGA prototype of the architecture.

A. Experimental Setup

Due to the lack of large-scale real-life flexible flow rules, we generated synthetic 12-tuple OpenFlow-like rules to examine the effectiveness of our decision forest -based schemes for flexible flow matching. Each rule was composed of 12 header fields that follow the current OpenFlow specification [3]. We used 6-bit field for the ingress port and randomly set each field value. Concretely, we generated each rule as follows:

1) Each field is randomly set as a wildcard. When the field is not set as a wildcard, the following steps are executed.
2) For source / destination IP address fields, the prefix length is set randomly from between 1 and 32, and then the value is set randomly from its possible values.
3) For other fields, the value is set randomly from its possible values.

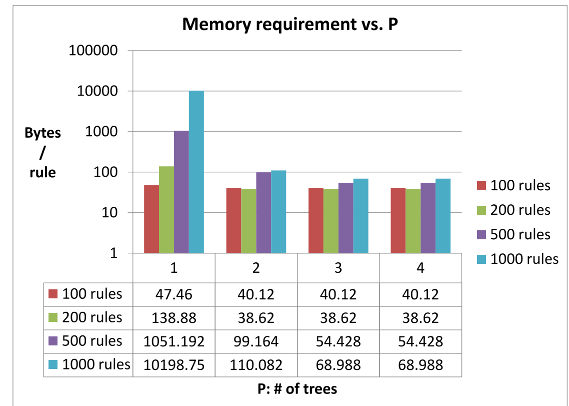
In this way, we generated four OpenFlow-like 12-tuple rule sets with 100, 200, 500, and 1K rules, each of which is independent of the others. Note that our generated rule sets include many impractical rules because each field value is set at random. But we argue that the lower bound of the performance of the decision forest scheme is approximated by using such randomly generated rule sets which do not match well the heuristic (Section III-A) observed in real-life flexible flow matching engines. Better performance can be expected using the decision forest scheme for large sets of real-life flexible flow rules which are to become available in the future.

B. Algorithm Evaluation

To evaluate the performance of the algorithms, we use following performance metrics:

- *Average memory requirement* (bytes) per rule. It is computed as the total memory requirement of a decision forest divided by the total number of rules for building the forest.
- *Tree depth*. It is defined as the maximum directed distance from the tree root to a leaf node. For a decision forest including multiple trees, we consider the maximum tree depth among these trees. A smaller tree depth leads to shorter pipelines and thus lower latency.
- *Number of cutting fields* (denoted N_{CF}) for building a decision tree. The N_{CF} of a decision forest is defined as the maximum N_{CF} among the trees in the forest. Using a smaller number of cutting fields results in less hardware for implementing cutting logic and smaller memory for storing cutting formation of each node.

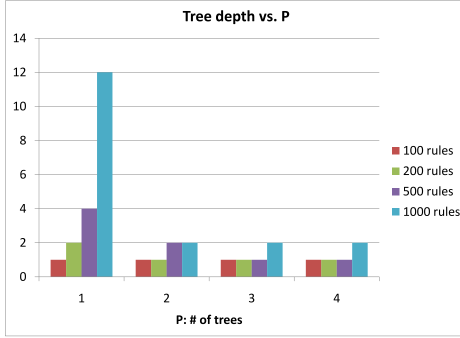
We set $bucketSize = 64$, $depthBound = 16$, and varied the number of trees $P = 1, 2, 3, 4$. Figure 3 shows the average memory requirement per rule, where logarithmic plot is used for the Y axis. In the case of $P = 1$, we observed memory explosion when the number of rules was increased from 100 to 1K. On the other hand, increasing P dramatically reduced the memory consumption, especially for the larger rule set. Almost 100-fold reduction in memory consumption was achieved for the 1K rules, when P was increased just from 1 to 2. With $P = 3$ or 4, the average memory requirement per rule remained on the same order of magnitude for different size of rule sets.



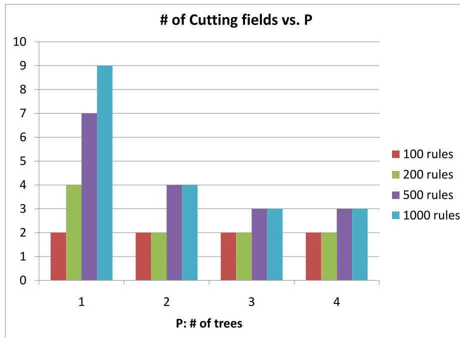
	1	2	3	4
100 rules	47.46	40.12	40.12	40.12
200 rules	138.88	38.62	38.62	38.62
500 rules	1051.192	99.164	54.428	54.428
1000 rules	10198.75	110.082	68.988	68.988

Figure 3. Average memory requirement with increasing P. ($bucketSize = 64$)

As shown in Figures 4(a) and 4(b), the tree depth and the number of cutting fields were also reduced by increasing P. With $P = 3$ or 4, 6-fold and 3-fold reductions were achieved, respectively, in the tree depth and the number of cutting fields, compared with using a single decision tree.



(a) Tree depth



(b) # of cutting fields

Figure 4. (a) Tree depth and (b) # of cutting fields, with increasing P. ($bucketSize = 64$)

C. Implementation Results

To implement the decision forest for 1K rules in hardware, we examined the performance results of each tree in a forest. Table II shows the breakdown with $P = 4$, $bucketSize = 32$, $depthBound = 4$.

Table II
BREAKDOWN OF A $P = 4$-TREE DECISION FOREST ($bucketSize = 32$)

Trees	# of Rules	# of Tree nodes	Memory (bytes/rule)	Tree depth	# of Cutting fields
Tree 1	712	545	78.70	2	3
Tree 2	184	265	84.70	2	5
Tree 3	65	17	41.78	1	2
Tree 4	39	9	45.23	1	2
Overall	1000	836	76.10	2	5

We mapped the above decision forest onto the 4-pipeline architecture. Since Block RAMs were not used efficiently for blocks of less than 1K entries, we merged the rule lists of the first two pipelines and used distributed memory for the remaining rule lists. BRAM utilization was improved at the cost of degrading the throughput to be one packet per clock cycle while dual-port RAMs were used. We implemented our design on FPGA using Xilinx ISE 10.1 development tools. The target device was Virtex-5 XC5VFX200T with -2 speed grade. Post place and route results showed that our design achieved a clock frequency of 125 MHz. The resulting throughput was 40 Gbps for minimum size (40 bytes) packets. The resource utilization of the design is summarized in Table III.

Table III
RESOURCE UTILIZATION ($bucketSize = 32$)

	Available	Used	Utilization
# of Slices	30,720	11,720	38%
# of 36Kb Block RAMs	456	256	56%
# of User I/Os	960	303	31%

V. CONCLUDING REMARKS

This paper proposed a FPGA-based parallel architecture, called *decision forest*, to address the performance challenges for flexible flow matching in next generation networks. We developed a framework to partition a set of complex flow rules into multiple subsets and build each rule subset into a depth-bounded decision tree. The partitioning scheme was designed so that both the overall memory requirement and the number of packet header fields for constructing the decision trees were reduced. Extensive simulation and FPGA implementation results demonstrate the effectiveness of our solution. The FPGA design supports 1K OpenFlow-like complex rules and sustains 40 Gbps throughput for minimum size (40 bytes) packets. Our future work includes porting our design into real systems and evaluating its performance under real-life scenarios such as dynamic rule updates.

REFERENCES

[1] J. C. Mogul, P. Yalagandula, J. Tourrilhes, R. McGeer, S. Banerjee, T. Connors, and P. Sharma, "API design challenges for open router platforms on proprietary hardware," in *Proc. HotNets-VII*, 2008.

[2] M. Casado, T. Koponen, D. Moon, and S. Shenker, "Rethinking packet forwarding hardware," in *Proc. HotNets-VII*, 2008.

[3] "OpenFlow Switch Specification, Version 1.0.0." [Online]. Available: http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf

[4] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *Proc. FPGA*, 2009, pp. 219–228.

[5] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.

[6] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proc. ANCS*. ACM, 2008, pp. 1–9.

[7] Y. Luo, P. Cascon, E. Murray, and J. Ortega, "Accelerating OpenFlow Switching with Network Processors," in *Proc. ANCS*. ACM, 2009.

[8] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. SIGCOMM*, 2003, pp. 213–224.