Scalable Packet Classification on FPGA

Weirong Jiang, *Member, IEEE*, and Viktor K. Prasanna, *Fellow, IEEE*

***Abstract*—Multi-field packet classification has evolved from traditional fixed 5-tuple matching to flexible matching with arbitrary combination of numerous packet header fields. For example, the recently proposed OpenFlow switching requires classifying each packet using up to 12-tuple packet header fields. It has become a great challenge to develop scalable solutions for next-generation packet classification that support higher throughput, larger rule sets and more packet header fields. This paper exploits the abundant parallelism and other desirable features provided by current field-programmable gate arrays (FPGAs), and proposes a decision-tree-based, 2-D multi-pipeline architecture for next-generation packet classification. We revisit the techniques for traditional 5-tuple packet classification and propose several optimization techniques for the state-of-the-art decision-tree-based algorithm. Given a set of 12-tuple rules, we develop a framework to partition the rule set into multiple subsets each of which is built into an optimized decision tree. A tree-to-pipeline mapping scheme is carefully designed to maximize the memory utilization while sustaining high throughput. The implementation results show that our architecture can store either 10K real-life 5-tuple rules or 1K synthetic 12-tuple rules in on-chip memory of a single state-of-the-art FPGA, and sustain 80 and 40 Gbps throughput for minimum size (40 bytes) packets, respectively.**

***Index Terms*—Decision tree, field-programmable gate array (FPGA), openflow, packet classification, pipeline, SRAM.**

I. Introduction

THE development of the Internet demands next-generation routers to support a variety of network functionalities, such as firewall processing, quality of service (QoS) differentiation, virtual private networks, policy routing, traffic billing, and other value added services. In order to provide these services, the router needs to classify the packets into different categories based on a set of predefined rules, which specify the value ranges of the multiple fields in the packet header. Such a function is called *multi-field packet classification*. In traditional network applications, packet classification problems usually consider the fixed 5-tuple fields: 32-bit source/destination IP addresses, 16-bit source/destination port numbers, and 8-bit transport layer protocol. Recently network virtualization emerges as an essential feature for next-generation enterprise, data center and cloud computing networks. This requires the underlying data plane be flexible and provide clean interface for control plane [1]. One such effort is the OpenFlow switch which manages explicitly the network flows using a rule set with rich definition as the software-hardware interface [2]. In OpenFlow, up to 12-tuple header fields are considered [3]. We call such OpenFlow-like packet classification the *next-generation packet classification* problems.

Due to the rapid growth of the Internet traffic, as well as the rule set size, multi-field packet classification has become one of the fundamental challenges to designing high speed routers. For example, the current link rate has been pushed beyond the OC-768 rate, i.e., 40 Gbps, which requires processing a packet every 8 ns in the worst case (where the packets are of minimum size, i.e., 40 bytes). Such throughput is impossible using existing software-based solutions [4]. Next-generation packet classification on more header fields poses an even bigger challenge. Most of the existing work in high-throughput packet classification is based on ternary content addressable memory (TCAM) [5]–[7] or a variety of hashing schemes such as Bloom Filters [8]–[10]. However, TCAMs are not scalable with respect to clock rate, power consumption, or circuit area, compared to SRAMs [11]. Most of TCAM-based solutions also suffer from range expansion when converting ranges into prefixes [6], [7]. Hashing-based solutions such as Bloom Filters have become popular due to their $O(1)$ time performance and high memory efficiency [12]. However, hashing cannot provide deterministic performance due to potential collision and is inefficient in handling wildcard or prefix matching [13]. A secondary module is always needed to resolve false positives inherent in Bloom Filters, which may be slow and can limit the overall performance [14].

As an alternative, our work focuses on optimizing and mapping state-of-the-art packet classification algorithms onto SRAM-based parallel architectures such as field-programmable gate array (FPGA). FPGA technology has become an attractive option for implementing real-time network processing engines [7], [10], [15] due to its ability to reconfigure and to offer abundant parallelism. State-of-the-art SRAM-based FPGA devices such as Xilinx Virtex-6 [16] and Altera Stratix-IV [17] provide high clock rate, low power dissipation and large amounts of on-chip dual-port memory with configurable word width. In this paper we exploit these desirable features in current FPGAs for designing high-performance next-generation packet classification engines.

The contributions of this paper are as follows.

- To the best of our knowledge, this work is among the first discussions of accelerating next-generation packet classification using FPGA. After revisiting the traditional fixed 5-tuple packet classification solutions, we adopt decision-tree-based schemes, which are considered among the most scalable packet classification algorithms [13], [18].

Manuscript received October 21, 2010; revised April 05, 2011; accepted June 21, 2011. Date of publication August 18, 2011; date of current version July 05, 2012. This work was supported by the U.S. National Science Foundation under Grant 1018801.

W. Jiang is with the Juniper Networks Inc., Sunnyvale, CA 94089 USA (e-mail: weirongj@acm.org).

V. K. Prasanna is with the Ming Hsieh Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90007 USA (e-mail: prasanna@usc.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2011.2162112

TABLE I
EXAMPLE 5-TUPLE RULE SET

Rule	SA	DA	SP	DP	Protocol	Priority	Action
R1	*	*	2 – 9	6 – 11	*	1	act0
R2	1*	0*	3 – 8	1 – 4	10	2	act0
R3	0*	0110*	9 – 12	10 – 13	11	3	act1
R4	0*	11*	11 – 14	4 – 8	*	4	act2
R5	011*	11*	1 – 4	9 – 15	10	5	act2
R6	011*	11*	1 – 4	4 – 15	10	5	act1
R7	110*	00*	0 – 15	5 – 6	11	6	act3
R8	110*	0110*	0 – 15	5 – 6	*	6	act0
R9	111*	0110*	0 – 15	7 – 9	11	7	act2
R10	111*	00*	0 – 15	4 – 9	*	7	act1

- We identify memory explosion to be the major challenge for handling 12-tuple packet classification. To address this challenge, we propose a decision-tree-based multi-pipeline architecture. We develop a framework, called *decision forest*, to partition a given set of 12-tuple rules into multiple subsets so that each subset uses a small number of header fields to build a decision tree of bounded depth.
- We tackle the problem of rule duplication when building the decision tree. Two optimization techniques, called *rule overlap reduction* and *precise range cutting*, are proposed to minimize the rule duplication. As a result, the memory requirement is almost linear with the number of rules.
- To map the tree onto the pipeline architecture, we introduce a fine-grained node-to-stage mapping scheme which allows imposing the bounds on the memory size as well as the number of nodes in each stage. As a result, the memory utilization of the architecture is maximized. The memory allocation scheme also enables using external SRAMs to handle even larger rule sets.
- We exploit the dual-port high-speed Block RAMs provided in state-of-the-art FPGAs to achieve a high throughput of two packets per clock cycle (PPC). On-the-fly rule update without service interruption becomes feasible due to the memory-based linear architecture. All the routing paths are localized to avoid large routing delay so that a high clock frequency is achieved.
- Implementation results show that our architecture can store 1K 12-tuple rules in a single Xilinx Virtex-5 FPGA, and sustain 40 Gbps throughput for matching minimum size (40 bytes) packets. To the best of our knowledge, this is the first FPGA design for 12-tuple packet classification to achieve over 10 Gbps. For traditional 5-field packet classification, our design can store 10K 5-tuple rules in a single Xilinx Virtex-5 FPGA, and sustain 80 Gbps throughput for minimum size (40 bytes) packets.

The rest of this paper is organized as follows. Section II states the problem we intend to solve, and summarizes several representative packet classification algorithms. Section III reviews the related work on FPGA-based multi-field packet classification engines. Section IV introduces our algorithms for partitioning the rule set and building the optimized decision tree. Section V presents the SRAM-based multi-pipeline architecture and the tree-to-pipeline mapping scheme. Section VI describes the FPGA implementation of our architecture in details.

TABLE II
HEADER FIELDS SUPPORTED IN CURRENT OPENFLOW

Header field	Notation	# of bits
Ingress port		Variable
Source Ethernet addresses	Eth src	48
Destination Ethernet address	Eth dst	48
Ethernet type	Eth type	16
VLAN ID		12
VLAN Priority		3
Source IP address	SA	32
Destination IP address	DA	32
IP Protocol	Prtl	8
IP Type of Service	ToS	6
Source port	SP	16
Destination port	DP	16

Section VII evaluates the performance of the algorithms and the architecture. Section VIII concludes this paper.

II. BACKGROUND

A. Problem Statement

In traditional 5-tuple packet classification, an IP packet is usually classified based on the 5 fields in the packet header: the 32-bit source/destination IP addresses (denoted $\mathbf{SA/DA}$), 16-bit source/destination port numbers (denoted $\mathbf{SP/DP}$), and 8-bit transport layer protocol. Individual entries for classifying a packet are called *rules*. Each rule can have one or more fields and their associated values, a priority, and an action to be taken if matched. Different fields in a rule require different kinds of matches: prefix match for SA/DA, range match for SP/DP, and exact match for the protocol field. Table I shows a simplified example, where each rule contains match conditions for 5 fields: 8-bit source and destination addresses (SA/DA), 4-bit source and destination port numbers (SP/DP), and a 2-bit protocol value.

Next-generation packet classification aims to match a larger number of header fields. The recently proposed OpenFlow switch [2], [3] enables network virtualization and brings programmability and flexibility to the network infrastructure. The major processing engine in the OpenFlow switch is packet classification, where up to 12-tuple header fields of each packet

[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 III
EXAMPLE OPENFLOW RULE SET (ETHERNET SRC/DST: 16-BIT; SA/DA:8-BIT; SP/DP: 4-BIT)

Rule	Ingr port	Eth src	Eth dst	Eth 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

are matched against all the rules [3]. The 12-tuple header fields supported in the current OpenFlow specification include the ingress port, source/destination Ethernet addresses, Ethernet type, VLAN ID, VLAN priority, source/destination IP addresses, IP protocol, IP Type of Service bits, and source/destination port numbers [3]. Table II shows the width of each field.[1]

Each field of an OpenFlow rule can be specified as either an exact number or a wildcard. IP address fields can also be specified as a prefix. Table III shows a simplified example of OpenFlow rule table, where we consider 16-bit Eth src/dst, 8-bit SA/DA, and 4-bit SP/DP. In the subsequent discussion, we have the following definitions.

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

A packet is considered matching a rule only if it matches all the fields within that rule. A packet can match multiple rules, but only the rule with the highest priority is used to take action.

B. Revisiting Packet Classification Algorithms

Next-generation packet classification can be viewed as a natural extension from traditional 5-tuple packet classification whose solutions have been extensively studied in the past decade. Comprehensive surveys can be found in [4] and [18]. Most of those algorithms fall into two categories: decomposition-based and decision-tree-based approaches.

Decomposition-based algorithms (e.g., parallel bit vector [19]), perform independent search on each field and finally combine the search results from all fields. Such algorithms are desirable for hardware implementation due to their parallel search on multiple fields. However, substantial storage is usually needed to merge the independent search results to obtain the final result. Decomposition-based algorithms have poor scalability, and work well only for small-scale rule sets.

Decision-tree-based algorithms (e.g., HyperCuts [20]), 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, which finally allows a low-cost linear search to find the best matching rule. After the decision tree is built, the algorithm to classify a packet is simple. Based on the value of the packet header, the algorithm follows the cutting sequence to locate the target subspace (i.e., a leaf node in the decision tree), and then performs a linear search on the rules in this subspace. Decision-tree-based algorithms allow incremental rule updates and scale better than decomposition-based algorithms. The outstanding representatives of decision-tree-based packet classification algorithms are HiCuts [21] and its enhanced version HyperCuts [20]. At each node of the decision tree, the search space is cut based on the information from one or more fields in the rule. HiCuts builds a decision tree using local optimization decisions at each node to choose the next dimension to cut, and how many cuts to make in the chosen dimension. The HyperCuts algorithm, on the other hand, allows cutting on multiple fields per step, resulting in a fatter and shorter decision tree. Fig. 1 shows the example of the HiCuts and the HyperCuts decision trees for a set of 2-field rules which can be represented geometrically. These rules are actually R1–R5 given in Table I, when only SP and DP fields are considered.

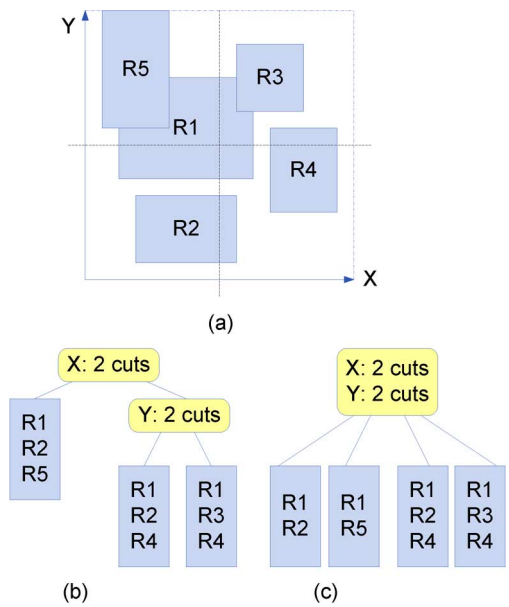


Fig. 1. Example of HiCuts and HyperCuts decision trees. (a) X and Y axes correspond to SP and DP fields for R1–R5 in Table I. (b), (c) A rounded rectangle in plain color denotes an internal tree node, and a rectangle in gray a leaf node.

In the worst case, both decision-tree-based and decomposition-based algorithms suffer from $O(N^d)$ memory explosion where N denotes the number of rules and d the number of fields in a rule [18].

III. Related Work

A. FPGA Designs for 5-Tuple Packet Classification

Although traditional 5-tuple packet classification is considered a saturated area of research, little work has been done on FPGAs. Most of existing FPGA implementations of packet classification engines are based on decomposition-based packet classification algorithms, such as BV [19] and DCFL [22].

Lakshman *et al.* [19] propose the Parallel Bit Vector (BV) algorithm, which is a decomposition-based algorithm targeting hardware implementation. It performs the parallel lookups on each individual field first. The lookup on each field returns a bit vector with each bit representing a rule. A bit is set if the corresponding rule is matched on this field; a bit is reset if the corresponding rule is not matched on this field. The result of the bitwise AND operation on these bit vectors indicates the set of rules that matches a given packet. The BV algorithm can provide a high throughput at the cost of low memory efficiency. Given N rules with D fields, since the projection of the N rules on each field may have $U = O(N)$ unique values and each value corresponds to one N-bit vector, the memory requirement of BV algorithms is $U * N * D = O(N^2)$. By combining TCAMs and the BV algorithm, Song *et al.* [7] present an architecture called BV-TCAM for multi-match packet classification. A TCAM performs prefix or exact match, while a multi-bit trie implemented in Tree Bitmap [23] is used for source or destination port lookup. [7] does not report the actual FPGA implementation results, though it claims that the whole circuit for 222 rules consumes less than 10% of the available logic and fewer than 20% of the available Block RAMs of a Xilinx XCV2000E FPGA. It also predicts the design after pipelining can achieve 10 Gbps throughput when implemented on advanced FPGAs.

Taylor *et al.* [22] introduce Distributed Crossproducting of Field Labels (DCFL), which is also a decomposition-based algorithm leveraging several observations of the structure of real filter sets. They decompose the multi-field searching problem and use independent search engines, which can operate in parallel to find the matching conditions for each filter field. Instead of using bit vectors, DCFL uses a network of efficient aggregation nodes, by employing Bloom Filters and encoding intermediate search results. As a result, the algorithm avoids the exponential increase in the time or space incurred when performing this operation in a single step. The authors predict that an optimized implementation of DCFL can provide over 100 million packets per second (MPPS) and store over 200K rules in the current generation of FPGA or application-specific integrated circuit (ASIC) without the need of external memories. However, their prediction is based on the maximum clock frequency of FPGA devices and a logic intensive approach using Bloom Filters. This approach may not be optimal for FPGA implementation due to long logic paths and large routing delays. Furthermore, the estimated number of rules is based only on the assumption of statistics similar to those of the currently available rule sets. Jedhe *et al.* [15] realize the DCFL architecture in their complete firewall implementation on a Xilinx Virtex 2 Pro FPGA, using a memory intensive approach, as opposed to the logic intensive one, so that on-the-fly update is feasible. They achieve a throughput of 50 MPPS, for a rule set of 128 entries. They also predict the throughput can be 24 Gbps when the design is implemented on Virtex-5 FPGAs. Papaefstathiou *et al.* [9] propose a memory-efficient decomposition-based packet classification algorithm, which uses multi-level Bloom Filters to combine the search results from all fields. Their FPGA implementation, called 2sBFCE [10], shows that the design can support 4K rules in 178 Kbytes memories. However, the design takes 26 clock cycles on average to classify a packet, resulting in low throughput of 1.875 Gbps on average. Note that both DCFL [22] and 2sBFCE [10] may suffer from false positives due to the use of Bloom Filters, as discussed in Section I.

Two recent works [24], [25] discuss several issues in implementing decision-tree-based packet classification algorithms on FPGA, with different motivations. Luo *et al.* [24] propose a method called *explicit range search* to allow more cuts per node than the HyperCuts algorithm. The tree height is dramatically reduced at the cost of increased memory consumption. At each internal node, a varying number of memory accesses may be needed to determine which child node to traverse, which may be infeasible for pipelining. Since the authors do not implement their design on FPGA, the actual performance results are unclear. To achieve power efficiency, Kennedy *et al.* [25] implement a simplified HyperCuts algorithm on an Altera Cyclone 3 FPGA. They store up to hundreds of rules in each leaf node and match them in parallel, resulting in low clock frequency (32 MHz reported in [25]). Since the search in the decision tree is not pipelined, their implementation can sustain only 0.47 Gbps in the worst cases where it takes 23 clock cycles to classify a packet for the rule set FW_20K.

B. Hardware Accelerators for OpenFlow

While OpenFlow switch technology is evolving, little attention has been paid on improving the performance of 12-tuple packet classification. Luo *et al.* [26] 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. Moreover, hashing-based solutions may suffer from hash collision and cannot produce deterministic throughput. Naous *et al.* [27] implement the OpenFlow switch on NetFPGA which is a Xilinx Virtex-2 Pro 50 FPGA board tailored for network applications. They use hashing for simple rules and a small TCAM 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 packet classification can sustain throughput above 10 Gbps in the worst case where packets are of minimum size i.e., 40 bytes.

Input: Rule set R.
Input: Parameters: $listSize$, $depthBound$, P.
Output: Decision forest: $\{T_i | i = 0, 1, \cdots, P-1\}$.
1: $i \leftarrow 0$, $R_i \leftarrow R$ and $split \leftarrow TRUE$.
2: **while** $i < P$ **do**
3: **if** $i == P-1$ **then** {The last subset / tree}
4: $split \leftarrow FALSE$
5: $\{T_i, R_{i+1}\} \leftarrow BuildTree\ (R_i,\ split,\ listSize,\ depthBound)$
6: $i \leftarrow i + 1$

Fig. 2. Building the decision forest.

IV. Motivations and Algorithms

A. Rule Set Partitioning

1) Motivation: Decision-tree-based algorithms (such as HyperCuts) usually 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 and d the number of fields in a rule. 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. For the example of OpenFlow table shown in Table III, if we consider only SA and DA fields, decision-tree-based algorithms such as HyperCuts [20] cut the search space recursively based on the values from SA and DA fields. 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.

Hence an intuitive idea is to partition 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 the rules 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.

2) Algorithm: We develop the rule set partitioning algorithm 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 in the algorithm shown in Fig. 2. The outcome of the algorithm is multiple decision trees, which we call *decision forest*. The rule set is partitioned dynamically during the construction of each decision tree. The function for building an optimized decision tree i.e., *BuildTree*($\cdot$) is detailed in Fig. 4 in Section IV-B.

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}. $i = 0, 1, \ldots, P-1$. In other words, the rules in $R_i - R_{i+1}$ are actually matched in the ith tree. The parameter *split* determines if the rest of the rule set will be partitioned. When building the last decision tree, *split* is disabled so that all the remaining rules are used to construct the tree. The rule duplication in the first $P-1$ trees will thus be reduced. Other parameters include *depthBound* which bounds the depth of each decision tree, and *listSize* which is inherited from the original HyperCuts algorithm to determine the maximum number of rules allowed to be contained in a leaf node.

The decision tree construction algorithm shown in Fig. 4 is based on the original HyperCuts algorithm, where Lines 6–7 and 17–19 are the major changes related to rule set partitioning. Lines 6–7 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, 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).

B. Optimizing HyperCuts

1) Motivation: After rule set partitioning, the rule duplication due to wildcard fields will be reduced. However, the HiCuts/HyperCuts algorithm may still suffer from rule duplication due to its own inefficiency. For example, as shown in Fig. 1, rules R1, R2, and R4 are replicated into multiple child nodes, in both HiCuts and HyperCuts trees.

We identify that the rule duplication when building the decision tree comes from two sources: 1) overlapping between different rules and 2) evenly cutting on all fields. Taking the rule set in Fig. 1 as an example, since R1 overlaps with R3 and R5, no matter how to cut the space, R1 will be replicated into the nodes which contain R3 or R5. Since each dimension is alway evenly cut, R2 and R4 are replicated though they do not overlap with any other rule. The second source of rule duplication exists only when cutting the port or the protocol fields of the packet header, since the prefix fields are evenly cut in nature. A prefix is matched from the most significant bit (MSB) to the least significant bit (LSB), which is equal to cutting the value space by half per step.

Accordingly, we propose the following two optimization techniques, called *rule overlap reduction* and *precise range cutting*.

- Rule overlap reduction: We store the rules (e.g., R1 shown in Fig. 3) which will be replicated into child nodes, in a list attached to each internal node. These rule lists are called *internal rule lists*.
- Precise range cutting: Assuming both X and Y in Fig. 3 are port fields, we seek the cutting points which result in the minimum number of rule duplication, instead of deciding the number of cuts for this field.

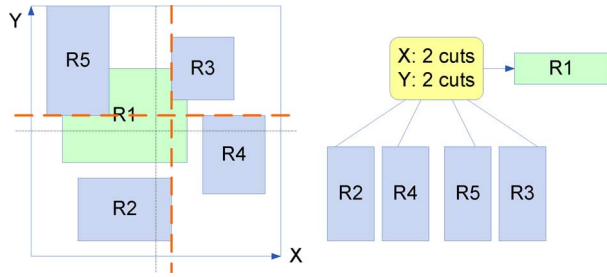


Fig. 3. Reducing rule duplication in HyperCuts tree.

As shown in Fig. 3, after applying the two optimizations, rule duplication is dramatically reduced, compared with Fig. 1(c). The memory requirement becomes linear with the number of rules. Section IV-B2 discusses the details about building the decision tree.

The proposed *rule overlap reduction* technique is similar to the *push common rule upwards* heuristic proposed by the authors of HyperCuts [20], where rules common to all descendant leaves are processed at the common parent node instead of being duplicated in all children. However, the *push common rule upwards* heuristic can solve only a fraction of rule duplication that is solved by our technique. Taking the HyperCuts tree in Fig. 1(c) as an example, only R1 will be pushed upwards while our technique allows storing R2 and R4 in the internal nodes as well. Also, the *push common rule upwards* heuristic is applied after the decision tree is built, while our *rule overlap reduction* technique is integrated with the decision tree construction algorithm.

2) Algorithm: Starting from the root node with the full rule set, we recursively cut the tree nodes until the number of rule in all the leaf nodes is smaller than a parameter named *listSize*. At each node, we need to figure out the set of fields to cut and the number of cuts performed on each field. We restrict the maximum number of cuts at each node to be 64. In other words, an internal node can have 2, 4, 8, 16, 32, or 64 children. For the port fields, we need to determine the precise cut points instead of the number of cuts. Since more bits are needed to store the cut points than to store the number of cuts, we restrict the number of cuts on port fields to be at most 2. For example, we can have 2 cuts on SA, 4 cuts on DA, 2 cuts on SP, and 2 cuts on DP. We do not cut on the protocol field since the first 4 fields are normally enough to distinguish different rules in real life [28].

We use the same criteria as in HiCuts [21] and HyperCuts [20] to determine the set of fields to cut (*ChooseField*()) and the number of cuts performed on SA and DA fields (*OptNumCuts*()). Our algorithm differs from HiCuts and HyperCuts in two aspects. First, when the port fields are selected to cut, we seek the cut point which results in the least rule duplication. Second, after the cutting method is determined, we pick the rules whose duplication counts are the largest among all the rules covered by the current node, and push them into the internal rule list of the current node, until the internal rule list becomes full. Fig. 4 shows the complete algorithm for building the decision tree, where n denotes a tree node, f a packet header field, and r a rule. Fig. 5 shows the decision tree constructed for the rule set given in Table I.

Input: Rule set R.
Input: Parameters: $split$, $listSize$, $depthBound$.
Output: Decision tree T and the split-out set R_{ex}.
1: Initialize the root node and push it into $nodeList$.
2: **while** $nodeList \neq null$ **do**
3: $n \leftarrow Pop(nodeList)$
4: **if** $n.numRules < listSize$ **then**
5: n is a leaf node. Continue.
6: **if** $n.depth == depthBound$ **then**
7: Assign to n the $listSize$ most specified rules from $n.rules$. Push remaining rules of $n.rules$ into R_{ex}. n is a leaf node. Continue.
8: $n.numCuts = 1$
9: **while** $n.numCuts < 64$ **do**
10: $f \leftarrow ChooseField(n)$
11: **if** f is SA or DA **then**
12: $numCuts[f] \leftarrow OptNumCuts(n, f)$
13: $n.numCuts$ *= $numCuts[f]$
14: **else if** f is SP or DP **then**
15: $cutPoint[f] \leftarrow OptCutPoint(n, f)$
16: $n.numCuts$ *= 2
17: **if** $split$ is TRUE **then**
18: $r \leftarrow PotentialDuplicatedRule(n, numCuts)$
19: Push r into R_{ex}.
20: Update the duplication counts of all $r \in n.ruleSet$: $r.dupCount \leftarrow$ # of copies of r after cutting.
21: **while** $n.internalList.numRules < listSize$ **do**
22: Find r_m which has the largest duplication count among the rules in $n.ruleSet \backslash n.internalList$.
23: Push r_m into $n.internalList$.
24: **if** All child nodes contain less than $listSize$ rules **then**
25: Break.
26: Push the child nodes into $nodeList$.

Fig. 4. Building the decision tree and the split-out set: $\{T, R_{ex}\} \leftarrow BuildTree(R, split, listSize, depthBound)$.

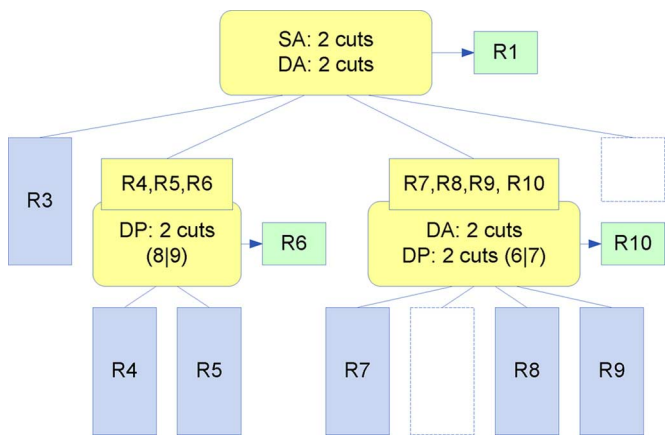


Fig. 5. Building the decision tree for the rule set given in Table I. (The values in parentheses represent the cut points on the port fields.)

V. Architecture

A. Overview

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 Fig. 6, 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

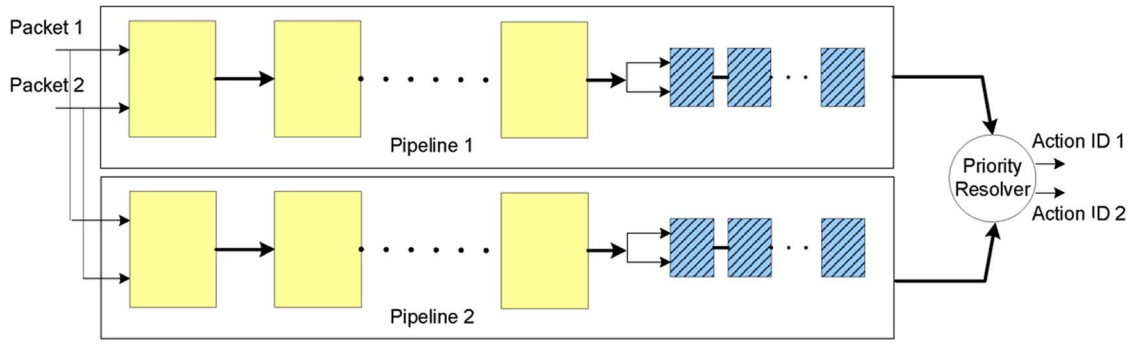


Fig. 6. Multi-pipeline architecture for searching the decision forest ($P = 2$). The shaded blocks (i.e., rule stages) store the leaf-level rule lists while the tree nodes are mapped onto plain-color blocks (i.e., tree stages).

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 *listSize* rules, we need *listSize* rule stages for matching these rules. All the leaf nodes of a tree have their rule lists mapped onto these *listSize* 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 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 $H + \text{listSize}$ clock cycles for each packet to go through the architecture, where H denotes the number of tree stages.

B. Pipeline

Like the HyperCuts with the *push common rule upwards* heuristic enabled, our algorithm may reduce the memory consumption at the cost of increased search time, if the process to match the rules in the *internal rule list* of each tree node is placed in the same critical path of decision tree traversal. Any packet traversing the decision tree must perform: 1) matching the rules in the *internal rule list* of the current node and 2) branching to the child nodes, in sequence. The number of memory accesses along the critical path can be very large in the worst cases. Although the throughput can be boosted by using a deep pipeline, the large delay passing the packet classification engine requires the router to use a large buffer to store the payload of all packets being classified. Moreover, since the search in the rule list and the traversal in the decision tree have different structures, a heterogeneous pipeline is needed, which complicates the hardware design.

FPGAs provide massive parallelism and high-speed dual-port Block RAMs distributed across the device. We exploit these features and propose a highly parallel architecture with localized routing paths, as shown in Fig. 7. The design is based on the following considerations.

1) The traversal of the decision tree can be pipelined. Thus we have a pipeline for traversing the decision tree, shown as light-color blocks in Fig. 7. We call this pipeline *Tree Pipeline*.
2) Whether it is an internal or a leaf node, each tree node is attached to a list of rules. Analogous to *internal rule list*, the rule list attached to a leaf node is called a *leaf-level rule list*. Search in the rule lists is pipelined as well. We call such a pipeline *Rule Pipeline*, shown in shaded blocks in Fig. 7.
3) When a packet reaches an internal tree node, the search in the internal rule list can be initiated at the same time the branching decision is made, by placing the rule list in a separate pipeline.
4) For the tree nodes mapped onto the same stage of the Tree Pipeline, their rule lists are mapped onto the same Rule Pipeline. Thus, there will be $H + 1$ Rule Pipelines if the Tree Pipeline has H stages.
5) All Rule Pipelines have the same number of stages. The total number of clock cycles for a packet to pass the architecture is $H + \text{listSize}$, where *listSize* is the number of stage in a Rule Pipeline.
6) Consider two neighboring stages of Tree Pipeline, denoted A and B, where Stage B follows Stage A. The Rule Pipeline attached to Stage A outputs the matching results one clock cycle earlier than the Rule Pipeline attached to Stage B. Instead of waiting for all matching results from all Rule Pipelines and directing them to a single priority resolver, we exploit the one clock cycle gap between two neighboring Tree Pipeline stages, to perform the partial priority resolving for the two previous matching results.
7) The Block RAMs in FPGAs are dual-port in nature. Both Tree Pipeline and Rule Pipelines can exploit this feature to process two packets per clock cycle. In other words, by duplicating the pipeline structure (i.e., logic), the throughput is doubled, while the memories are shared by the dual pipelines.

As shown in Fig. 7, all routing paths between blocks are localized. This can result in a high clock frequency even when the on-chip resources are heavily utilized. The FPGA implementation of our architecture is detailed in Section VI.

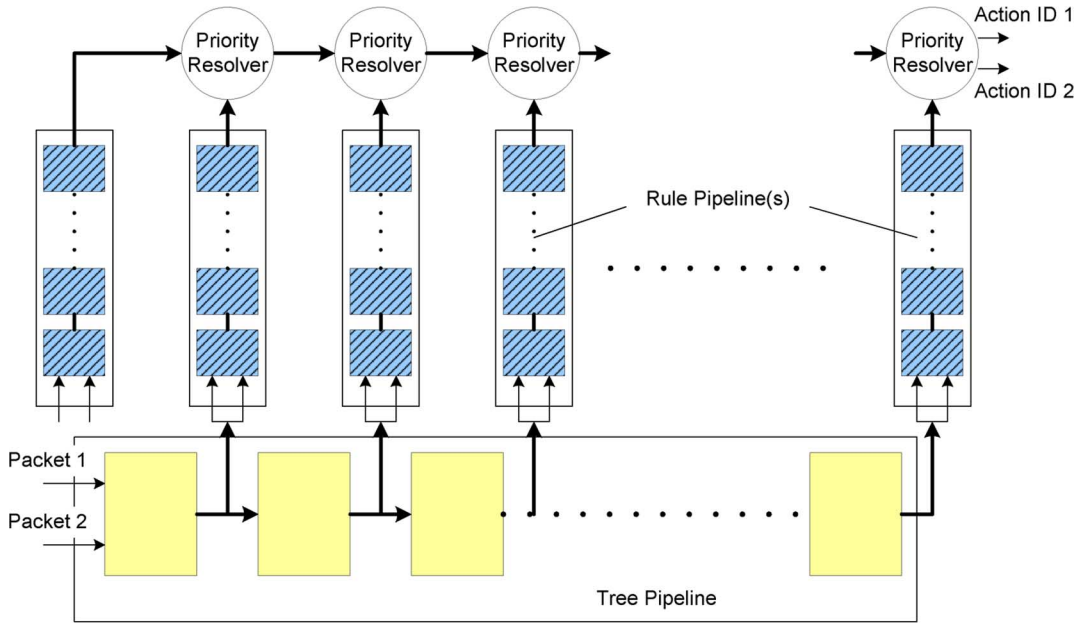


Fig. 7. Block diagram of the 2-D linear dual-pipeline architecture. H stages in the Tree Pipeline. *listSize* stages in each Rule Pipeline.

C. Tree-to-Pipeline Mapping

The size of the memory in each pipeline stage must be determined before FPGA implementation. However, as shown in [29], when simply mapping each level of the decision tree onto a separate stage, the memory distribution across stages can vary widely. Allocating memory with the maximum size for each stage results in large memory wastage. Baboescu *et al.* [29] propose a Ring pipeline architecture which employs TCAMs to achieve balanced memory distribution at the cost of halving the throughput to one packet per two clock cycles, i.e., 0.5 PPC, due to its non-linear structure.

Our task is to map the decision tree onto a **linear** pipeline (i.e., Tree Pipeline in our architecture) to achieve balanced memory distribution over stages, while sustaining a throughput of one packet per clock cycle (which can be further improved to 2 PPC by employing dual-port RAMs). The memory distribution across stages should be balanced not only for the Tree Pipeline, but also for all the Rule Pipelines. Note that the number of words in each stage of a Rule Pipelines depends on the number of tree nodes rather than the number of words in the corresponding stage of Tree Pipeline, as shown in Fig. 8. The challenge comes from the various number of words needed for tree nodes. As a result, the tree-to-pipeline mapping scheme requires not only balanced memory distribution, but also balanced node distribution across stages. Moreover, to maximize the memory utilization in each stage, the sum of the number of words of all nodes in a stage should approach some power of 2. Otherwise, for example, we need to allocate 2048 words for a stage consuming only 1025 words.

The above problem is a variant of bin packing problems, and can be proved to be NP-complete. We use a heuristic similar to our previous study of trie-based IP lookup [30], which allows the nodes on the same level of the tree to be mapped onto different stages. This provides more flexibility to map the tree nodes, and helps achieve a balanced memory and node distribution across the stages in a pipeline, as shown in Fig. 8. Only one constraint must be followed.

Constraint 1: If node A is an ancestor of node B in the tree, then A must be mapped to a stage preceding the stage to which B is mapped.

We impose two bounds, namely B_M and B_N for the memory and node distribution, respectively. The values of the bounds are some power of 2. The criteria to set the bounds is to minimize the number of pipeline stages while achieving balanced distribution over stages. The complete tree-to-pipeline mapping algorithm is shown in Fig. 9, where n denotes a tree node, H the number of stages, S_r the set of remaining nodes to be mapped onto stages, M_i the number of words of the ith stage, and N_i the number of nodes mapped onto the ith stage. We manage two lists, *ReadyList* and *NextReadyList*. The former stores the nodes that are available for filling the current stage, while the latter stores the nodes for filling the next stage. We start with mapping the nodes that are children of the root onto Stage 1. When filling a stage, the nodes in *ReadyList* are popped out and mapped onto the stage, in the decreasing order of their heights.[2] After a node is assigned to a stage, its children are pushed into *NextReadyList*. When a stage is full or *ReadyList* becomes empty, we move on to the next stage. At that time, *NextReadyList* is merged into *ReadyList*. By these means, *Constraint* 1 is met. The complexity of this mapping algorithm is $O(N)$, where N denotes the total number of tree nodes.

Our tree-to-pipeline mapping algorithm allows two nodes on the same tree level to be mapped to different stages. We implement this feature by using a simple method. Each node stored in

[2]*Height* of a tree node is defined as the maximum directed distance from it to a leaf node.

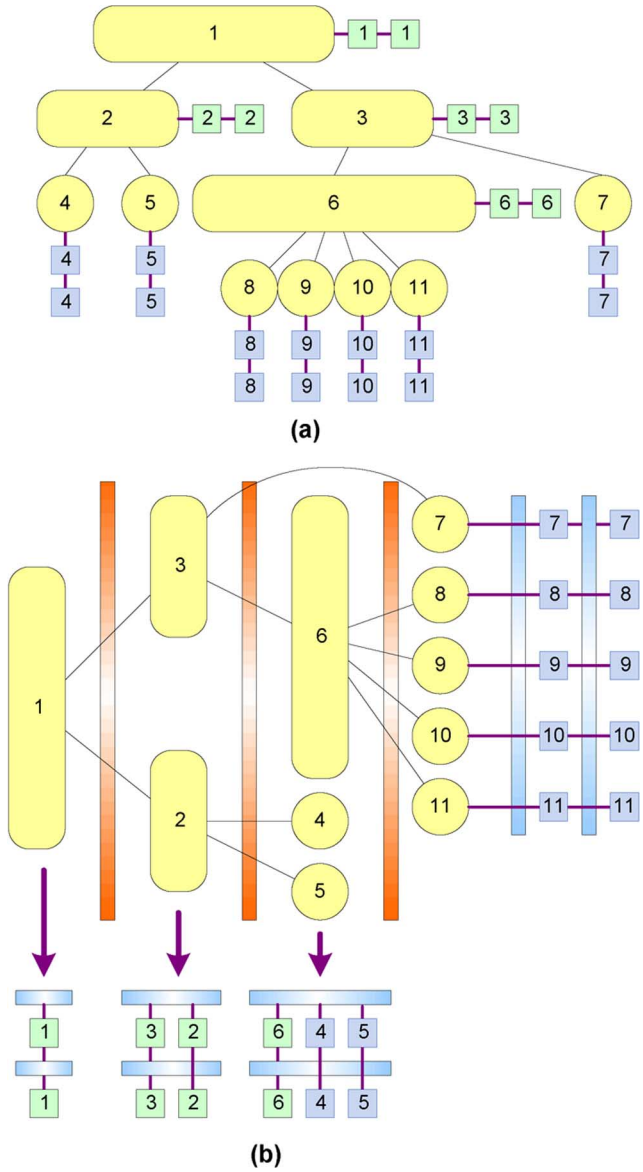


Fig. 8. Mapping a decision tree onto pipeline stages ($H = 4, listSize = 2$). The rounded nodes denote the tree nodes. The square nodes denote the rule lists. The rectangle bars denote the registers.

Input: The tree T.
Output: H stages with mapped nodes.
1: Initialization: $ReadyList \leftarrow \phi$, $NextReadyList \leftarrow \phi$, $S_r \leftarrow T$, $H \leftarrow 0$.
2: Push the children of the root into $ReadyList$.
3: **while** $S_r \neq \phi$ **do**
4: Sort the nodes in $ReadyList$ in the decreasing order of their heights.
5: **while** $M_i < B_M$ AND $N_i < B_N$ AND $Readylist \neq \phi$ **do**
6: Pop node from $ReadyList$.
7: Map the popped node n_p onto Stage H.
8: Push its children into $NextReadyList$.
9: $M_i \leftarrow M_i + size(n_p)$. Update S_r.
10: $H \leftarrow H + 1$.
11: Merge the $NextReadyList$ to the $ReadyList$.

Fig. 9. Mapping the decision tree onto a pipeline.

the local memory of a pipeline stage has one extra field: the distance to the pipeline stage where the child node is stored. When a packet is passed through the pipeline, the distance value is decremented by 1 when it goes through a stage. When the distance value becomes 0, the child node's address is used to access the memory in that stage.

External SRAMs are usually needed to handle very large rule sets, while the number of external SRAMs is constrained by the number of I/O pins in our architecture. By assigning large values of B_M and B_N for one or two specific stages, our mapping algorithm can be extended to allocate a large number of tree nodes onto few external SRAMs which consume controllable number of I/O pins.

VI. Implementation

A. Pipeline for Decision Tree

As shown in Fig. 5, different internal nodes in a decision tree may have different numbers of cuts, which can come from different fields. A simple solution is hard-wiring the connections, which however cannot update the tree structure on-the-fly [24]. We propose a circuit design, where the child node pointer is computed based on the cut information read from the memory. Recall that we perform precise range cutting on port fields. We store the cut points for SP/DP fields and the number of cut bits for SA/DA fields. We can update the memory content to change the number of bits for SA and DA to cut, and the cut enable bit for SP and DP to indicate whether to cut SP or DP.

B. Pipeline for Rule Lists

When a packet accesses the memory in a Tree Pipeline stage, it will obtain the pointer to the rule list associated with the current tree node being accessed. The packet uses this pointer to access all stages of the Rule Pipeline attached to the current Tree Pipeline stage. Each rule is stored as one word in a Rule Pipeline stage, benefiting from the large word width provided by FPGA. Within a stage of the Rule Pipeline, the packet uses the pointer to retrieve one rule and compare its header fields to find a match. When a match is found in the current Rule Pipeline stage, the packet will carry the corresponding action information with the rule priority along the Rule Pipeline until it finds another match where the matching rule has higher priority than the one the packet is carrying.

C. Rule Update

The dual-port memory in each stage enables only one write port to guarantee the data consistency. We update the memory in the pipeline by inserting *write bubbles* [31]. The new content of the memory is computed offline. When an update is initiated, a write bubble is inserted into the pipeline. Each write bubble is assigned an ID. There is one write bubble table in each stage, storing the update information associated with the write bubble ID. When a write bubble arrives at the stage prior to the stage to be updated, the write bubble uses its ID to look up the write bubble table and retrieves: 1) the memory address to be updated in the next stage; 2) the new content for that memory location; and 3) a write enable bit. If the write enable bit is set, the write bubble will use the new content to update the memory location in the next stage. Since the architecture is linear, all packets preceding or following the write bubble can perform their operations while the write bubble performs an update. In the worst

case, a rule update may result in the change of all nodes in the decision trees. Then we need insert $O(N)$ write bubbles, where N denotes the total number of tree nodes.

VII. Experimental Results

We conducted extensive experiments to evaluate the performance of our schemes including the algorithms and FPGA prototype of the architecture for both traditional 5-tuple and next-generation 12-tuple packet classification problems.

A. Experimental Setup

For traditional 5-tuple packet classification, we used the rule sets from [32]. These synthetic rule sets are generated using ClassBench [33], with parameter files extracted from real-life rules. The size of the rule sets varies from hundreds of to tens of thousands of rules.

Due to the lack of large-scale real-life OpenFlow rule sets, we generated synthetic 12-tuple OpenFlow-like rules to examine the effectiveness of our decision forest -based schemes for next generation packet classification. 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.

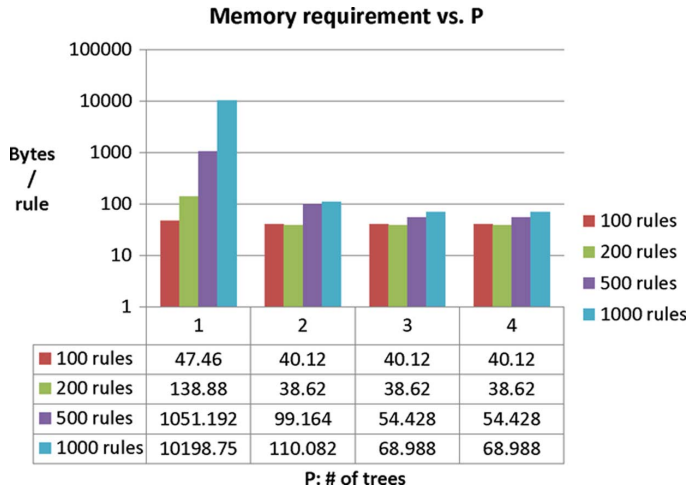
In this way, we generated four OpenFlow-like 12-tuple rule sets with 100, 200, 500, and 1K rules, each of which is independent of the others. Note that our generated rule sets include many impractical rules because each field value is set at random.

B. Algorithm Evaluation

1) 12-Tuple Rule Sets: 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. It represents the scalability of our algorithms.
- *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 $\mathrm{listSize} = 64$, $\mathrm{depthBound} = 16$, and varied the number of trees $P = 1, 2, 3, 4$. Fig. 10 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

Fig. 10. Average memory requirement with increasing P. ($\mathrm{listSize} = 64$).

As shown in Fig. 11(a) and (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.

2) 5-Tuple Rule Sets: We set $P = 1$ and evaluated the effectiveness of our optimized decision-tree-based packet classification algorithm, by conducting experiments on 4 real-life 5-tuple rule sets of different sizes from [32]. The results are shown in Table IV. In these experiments, we set $\mathrm{listSize} = 8$, which was optimal according to a series of tests where we found that a larger *listSize* resulted in lower memory requirement but deeper Rule Pipelines. The memory reduction became unremarkable when $\mathrm{listSize} > 8$. According to Table IV, our algorithm kept the memory requirement to be linear with the number of rules, and thus achieved much better scalability than the original HyperCuts algorithm. Also, the depth of the decision tree generated using our algorithm was much smaller than that of HyperCuts, indicating a smaller delay for a packet to pass through the engine. Note that the tree depth is determined by not only the size of the rule set but also the characteristics of the rule set. All the decision tree-based algorithms share this problem that the tree depth is indeterministic.

According to Table IV, the Tree Pipeline needed at least 9 stages to map the decision tree for ACL_10K. We conducted a series of experiments to find the optimal values for the memory and the node distribution bounds, i.e., B_M and B_N. When $B_M \leq 512$ or $B_N \leq 128$, the Tree Pipeline needed more than 20 stages. When $B_M \geq 2048$ or $B_N \geq 512$, the memory and the node distribution over the stages was same as that using *static* mapping scheme which mapped each tree level onto a stage. Only when $B_M = 1024, B_N = 256$, both memory and node distribution were balanced, while the number of stages

TABLE IV
PERFORMANCE OF ALGORITHMS FOR RULE SETS OF VARIOUS SIZES

		Our algorithm		Original HyperCuts	
Rule set	# of rules	Memory (Bytes/rule)	Tree depth	Memory (Bytes/rule)	Tree depth
ACL_100	98	29.31	8	52.16	23
ACL_1k	916	26.62	11	122.04	20
ACL_5k	4415	29.54	12	314.88	29
ACL_10k	9603	27.46	9	1727.28	29

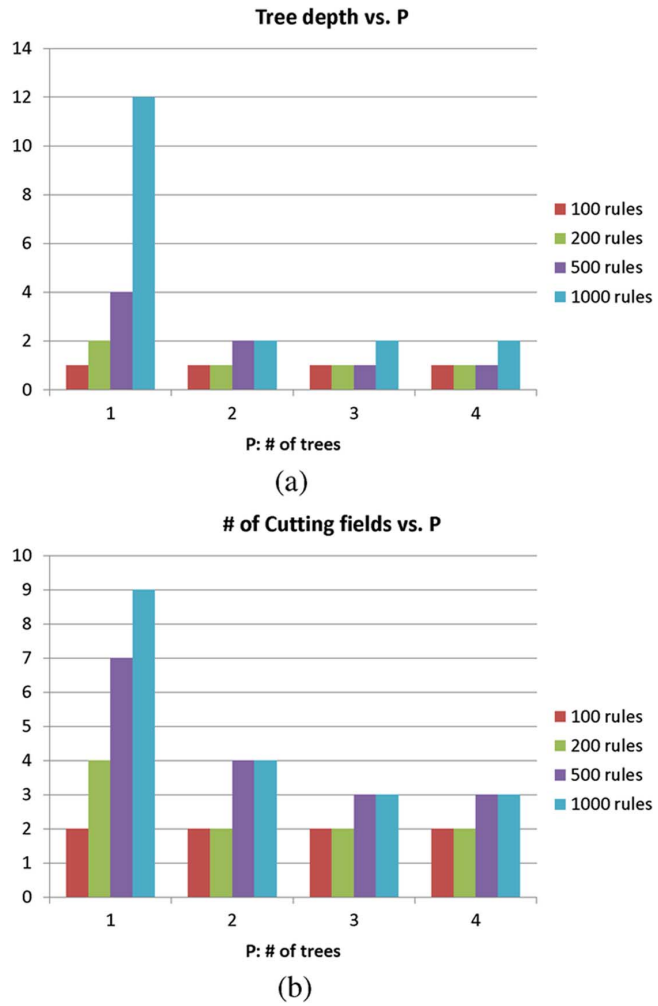


Fig. 11. (a) Tree depth and (b) # of cutting fields, with increasing P. (listSize $= 64$).

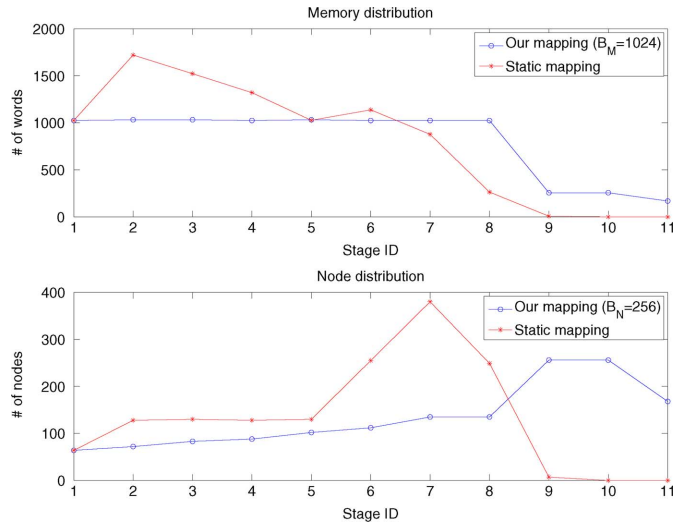


Fig. 12. Distribution over Tree Pipeline stages for ACL_10K.

TABLE V
BREAKDOWN OF A $P = 4$-TREE DECISION FOREST (listSize $= 32$)

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

TABLE VI
RESOURCE UTILIZATION (listSize $= 32$)

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

TABLE VII
RESOURCE UTILIZATION

	Used	Available	Utilization
Number of Slices	10,307	30,720	33%
Number of bonded IOBs	223	960	23%
Number of Block RAMs	407	456	89%

needed was increased slightly to 11, i.e., $H = 11$. As Fig. 12 shows, our mapping scheme outperformed the static mapping scheme with respect to both memory and node distribution.

C. Implementation Results

1) 12-Tuple Rule Sets: To implement the decision forest for 1K 12-tuple rules in FPGA, we examined the performance results of each tree in a forest. Table V shows the breakdown with $P = 4$, listSize $= 32$, depthBound $= 4$.

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

TABLE VIII
PERFORMANCE COMPARISON

Packet classification engines	Platform	# of rules	Total memory used (Kbytes)	Throughput (Gbps)	Efficiency (Gbps/KB)
Our approach	FPGA	9603	612	80.23	1358.9
Simplified HyperCuts [25]	FPGA	10000	286	10.84 (5.12)	379.0
BV-TCAM [7]	FPGA	222	16	10 (N/A)	138.8
2sBFCE [10]	FPGA	4000	178	2.06 (1.88)	46.3
Memory-based DCFL [15]	FPGA	128	221	24 (16)	13.9
B2PC [9]	ASIC	3300	540	13.6	83.1

for minimum size (40 bytes) packets. The resource utilization of the design is summarized in Table VI.

2) 5-Tuple Rule Sets: Based on the mapping results, we initialized the parameters of the architecture for FPGA implementation. According to the previous section, to include the largest 5-tuple rule set ACL_10K, the architecture needed $H = 11$ stages in Tree Pipeline, and 12 Rule Pipelines each of which had $\text{listSize} = 8$ stages. Each stage of Tree Pipeline needed $B_M = 1024$ words, each of which was 72 bits including base address of a node, cutting information, pointer to the rule list, and distance value. Each stage of Rule Pipeline needed $B_N = 256$ words each of which was 171 bits including all fields of a rule, priority and action information.

We implemented our design in Verilog. We used Xilinx ISE 10.1 development tools. The target device was Xilinx Virtex-5 XC5VFX200T with -2 speed grade. Post place and route results showed that our design could achieve a clock frequency of 125.4 MHz. The resource utilization is shown in Table VII. Among the allocated memory, 612 Kbytes was consumed for storing the decision tree and all rule lists.

Table VIII compares our design with the state-of-the-art FPGA-based packet classification engines. For fair comparison, the results of the compared work were scaled to Xilinx Virtex-5 platforms based on the maximum clock frequency.[3] The values in parentheses were the original data reported in those papers. Considering the time-space trade-off, we used a new performance metric, named *Efficiency*, which was defined as the throughput divided by the average memory size per rule. Our design outperformed the others with respect to throughput and efficiency. Note that our work is the only design to achieve more than 40 Gbps throughput. Such high throughput is due to our deeply pipelined architecture.

VIII. CONCLUSION

This paper presented a novel decision-tree-based linear multi-pipeline architecture on FPGAs for wire-speed multi-field packet classification. We considered the next-generation packet classification problems where more than 5-tuple packet header fields would be classified. Several optimization techniques were proposed to reduce the memory requirement of the state-of-the-art decision-tree-based packet classification algorithm, so that 10K 5-tuple rules or 1K 12-tuple rules could fit in the on-chip memory of a single FPGA. Our architecture provided on-the-fly reconfiguration due to the linear memory-based architecture. Extensive simulation and FPGA implementation results demonstrate the effectiveness of our solution. The FPGA design supports 10K 5-tuple rules or 1K OpenFlow-like complex rules and sustains over 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.

[3]The BV-TCAM paper [7] does not present the implementation result about the throughput. We use the predicted value given in [7].

REFERENCES

[1] M. Casado, T. Koponen, D. Moon, and S. Shenker, "Rethinking packet forwarding hardware," in *Proc. HotNets—VII*, 2008, pp. 1–6.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[3] OpenFlow Foundation, "OpenFlow Switch Specification, Version 1.0.0," 2009. [Online]. Available: http://www.openflowswitch.org/documents/openflow-spec-v1.0.0.pdf

[4] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[5] F. Yu, R. H. Katz, and T. V. Lakshman, "Efficient multimatch packet classification and lookup with TCAM," *IEEE Micro*, vol. 25, no. 1, pp. 50–59, Jan. 2005.

[6] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in *Proc. SIGCOMM*, 2005, pp. 193–204.

[7] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proc. FPGA*, 2005, pp. 238–245.

[8] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," in *Proc. ANCS*, 2006, pp. 61–70.

[9] I. Papaefstathiou and V. Papaefstathiou, "Memory-efficient 5D packet classification at 40 Gbps," in *Proc. INFOCOM*, 2007, pp. 1370–1378.

[10] A. Nikitakis and I. Papaefstathiou, "A memory-efficient FPGA-based classification engine," in *Proc. FCCM*, 2008, pp. 53–62.

[11] W. Jiang and V. K. Prasanna, "Sequence-preserving parallel IP lookup using multiple SRAM-based pipelines," *J. Parallel Distrib. Comput.*, vol. 69, no. 9, pp. 778–789, 2009.

[12] H. Yu and R. Mahapatra, "A power- and throughput-efficient packet classifier with n bloom filters," *IEEE Trans. Comput.*, vol. 60, no. 8, pp. 1182–1193, Aug. 2011.

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

[14] I. Sourdis, "Designs & algorithms for packet and content inspection" Ph.D. dissertation, Comput. Eng. Div., Delft Univ. Technol., Delft, The Netherlands, 2007. [Online]. Available: http://ce.et.tudelft.nl/publicationfiles/1464_564 sourdis phdthesis.pdf

[15] G. S. Jedhe, A. Ramamoorthy, and K. Varghese, "A scalable high throughput firewall in FPGA," in *Proc. FCCM*, 2008, pp. 43–52.

[16] Xilinx, Inc., San Jose, CA, "Xilinx Virtex-6 FPGA family," 2009. [Online]. Available: www.xilinx.com/products/virtex6/

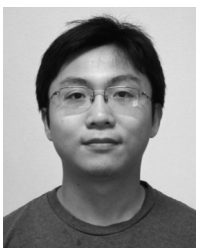
[17] Altera Corp., San Jose, CA, "Altera Stratix IV FPGA," 2009. [Online]. Available: http://www.altera.com/products/devices/stratix-fpgas/stratix-iv/

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

[19] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. SIGCOMM*, 1998, pp. 203–214.

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

[21] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, 2000.
[22] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducing of field labels," in *Proc. INFOCOM*, 2005, pp. 269–280.
[23] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: Hardware/software IP lookups with incremental updates," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.
[24] Y. Luo, K. Xiang, and S. Li, "Acceleration of decision tree searching for IP traffic classification," in *Proc. ANCS*, 2008, pp. 40–49.
[25] A. Kennedy, X. Wang, Z. Liu, and B. Liu, "Low power architecture for high speed packet classification," in *Proc. ANCS*, 2008, pp. 131–140.
[26] Y. Luo, P. Cascon, E. Murray, and J. Ortega, "Accelerating OpenFlow switching with network processors," in *Proc. ANCS*, 2009, pp. 70–71.
[27] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," in *Proc. ANCS*, 2008, pp. 1–9.
[28] M. E. Kounavis, A. Kumar, R. Yavatkar, and H. Vin, "Two stage packet classification using most specific filter matching and transport level sharing," *Comput. Netw.*, vol. 51, no. 18, pp. 4951–4978, 2007.
[29] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh, "A tree based router search engine architecture with single port memories," in *Proc. ISCA*, 2005, pp. 123–133.
[30] W. Jiang and V. K. Prasanna, "Parallel IP lookup using multiple SRAMbased pipelines," in *Proc. IPDPS*, 2008, pp. 1–14.
[31] A. Basu and G. Narlikar, "Fast incremental updates for pipelined forwarding engines," in *Proc. INFOCOM*, 2003, pp. 64–74.
[32] Applied Research Lab, Washington Univ., St. Louis, "Packet classification filter sets," 2009. [Online]. Available: http://www.arl.wustl.edu/~hs1/pclasseval.html
[33] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Mar. 2007.



Weirong Jiang (M'06) received the B.S. degree in automation and the M.S. degree in control science and engineering from Tsinghua University, Beijing, China, in 2004 and 2006, respectively, and the Ph.D. degree in computer engineering from University of Southern California, Los Angeles, in 2010.

He is a Member of Technical Staff with Juniper Networks Inc., Sunnyvale, CA. His research is on parallel algorithm and architecture design for high-performance low-power packet processing in Internet infrastructure. His research interests also include network security, reconfigurable computing, data mining, and wireless networking. He has published 2 book chapters and over 30 papers on top conferences and journals.

Dr. Jiang was a recipient of five Best Paper Awards. He is a member of the ACM.

Viktor K. Prasanna (F'96) received the B.S. degree in electronics engineering from the Bangalore University, Bangalore, India, the M.S. degree from the School of Automation, Indian Institute of Science, Bangalore, India, and the Ph.D. degree in computer science from the Pennsylvania State University, Philadelphia.

He is a Charles Lee Powell Chair in Engineering with the Ming Hsieh Department of Electrical Engineering and Professor of Computer Science, University of Southern California (USC), Los Angeles. His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He is the Executive Director of the USC-Infosys Center for Advanced Software Technologies (CAST) and is an Associate Director of the USC-Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). He also serves as the Director of the Center for Energy Informatics at USC.

Dr. Prasanna served as the Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTERS during 2003–2006. Currently, he is the Editor-in-Chief of the *Journal of Parallel and Distributed Computing*. He was the founding Chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the Steering Co-Chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the Steering Chair of the IEEE International Conference on High Performance Computing (HiPC). He is a Fellow of the ACM and the American Association for Advancement of Science (AAAS). He was a recipient of the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University.