Multi-dimensional Packet Classification on FPGA: 100 Gbps and Beyond

Yaxuan Qi [#1], Jeffrey Fong [#2], Weirong Jiang [*3], Bo Xu [#4], Jun Li [#5], Viktor Prasanna [*6]

[#] *Research Institute of Information Technology*
Tsinghua University, Beijing 100084, China
[*] *Ming Hsieh Department of Electrical Engineering*
University of Southern California, Los Angeles, CA 90089, USA
[1,5] {yaxuan, junl}@tsinghua.edu.cn
[2,4] {fangyy09, xb00}@mails.tsinghua.edu.cn
[3,6] {weirongj, prasanna}@usc.edu

***Abstract*—Multi-dimensional packet classification is a key task in network applications, such as firewalls, intrusion prevention and traffic management systems. With the rapid growth of network bandwidth, wire speed multi-dimensional packet classification has become a major challenge for next-generation network processing devices. In this paper, we present an FPGA-based architecture targeting 100 Gbps packet classification. Our solution is based on HyperSplit, a memory-efficient tree search algorithm. First, we present efficient pipeline architecture for mapping HyperSplit tree. Special logics are designed to support dual-packet classification per clock cycle. Second, a node-merging algorithm is proposed to reduce the number of pipeline stages without significantly increasing the memory. Third, a leaf-pushing algorithm is designed to control the memory usage and to support on-the-fly rule update. The implementation results show that our architecture can achieve more than 100 Gbps throughput for the 64-byte minimum Ethernet packets. With a single Virtex-6 chip, our approach can handle over 50K rules. Compared with state-of-the-art multi-core network processors based solutions our FPGA design has at least a 10x speedup.**

***Keywords*—Packet classification, FPGA, Pipeline**

I. Introduction

With the rapid growth of Internet, keeping network operations efficient and secure is crucial. Traffic management, access control, intrusion prevention, and many other network services require a discrimination of network packets based on their multi-field headers. This is achieved by multi-dimensional packet classification.

Although there have been a lot of research on multi-dimensional packet classification over the past decade [1] [2], most existing solutions cannot meet the performance requirement. On the one hand, software solutions commonly used on multi-core network processors for high performance packet classification has good flexibility and programmability, but it inherently lacks high parallelism and abundant on-chip memory. As a result, even the best software classification algorithms on multi-core network processors can only achieve 10 Gbps [3] [4]. This is an order of magnitude less than the widely deployed 100 Gbps links used in large ISP (Internet Service Provider) backbones and DC (Data Center) networks [5]. On the other hand, hardware solutions are mainly based on TCAM (Ternary Contend-Addressable Memory). While TCAM-based solutions can reach wire speed performance, they sacrifice scalability, programmability and power efficiency. TCAM-based solutions also suffer from a range-to-prefix converting problem, making it difficult to support large and complex rule sets [6] [7].

For this reason, the demand for flexible wire speed packet classification is still motivating research today. In this paper, we present high-performance multi-dimensional packet classification architecture based on FPGA technology. The contributions of this paper are:

- ♦ **Architecture design**: The proposed packet classification architecture is carefully designed to map a memory-efficient packet classification algorithm into a linear pipeline. The design also uses the available dual-port RAMs on current FPGA devices to double the classification rate. Compared to existing FPGA-based solutions, this architecture has higher performance while consuming less FPGA resources.
- ♦ **Implementation optimization**: Two optimizations for the packet classification architecture are proposed. First, a node-merging algorithm is used to reduce the number of pipeline stages without significant cost of extra memory. Secondly, a leaf-pushing algorithm is used for limiting the memory usage on each pipeline stage to support on-the-fly update without device reconfiguration.

Table 1. Example rule set

Rule	*Destination IP address*	*Source IP address*	*Destination port*	*Source port*	*Layer-4 protocol*	*Priority*	*Action*
R1	188.111.8.28/32	64.10.8.20/32	80	0~65535	TCP	1	RESET
R2	188.111.8.28/32	0.0.0.0/0	53	0~65535	TCP	2	ACCEPT
R3	188.111.0.0/16	202.110.0.15/32	0~65535	0~65535	UDP	3	DENY
R4	188.111.0.0/16	0.0.0.0/0	0~65535	0~65535	TCP	4	DENY
R5	182.105.3.20/32	0.0.0.0/0	80	6110~6112	UDP	11	ACCEPT
R6	182.105.3.0/24	0.0.0.0/0	0~65535	1024~65535	ANY	12	ACCEPT
R7	0.0.0.0/0	0.0.0.0/0	0~65535	0~65535	ANY	99	DROP

Table 2. Packet classification algorithms

	Parallel search algorithms	*Tree search algorithms*
Typical algorithms	RFC, HSM, DCFL	HiCuts, HyperCuts, AggreCuts
Space decomposition methods	Decompose the search space by end-points on each dimension	Decompose the search space with equal-sized cuttings on selected fields
Packet search	Parallel search on all fields and then lookup cross-producting tables for final match	Traverse a tree and apply linear search the leaf node for final match
Advantages	Fast and deterministic performance	Modest memory usage
Limitations	Large memory usage	Non-deterministic performance

♦ **Performance evaluation**: The proposed architecture is evaluated with publicly available real-life rule sets [8]. When tested with minimum-size (64 bytes) Ethernet packets, over 100 Gbps packet classification rate is achieved. More than 50K rules can be supported by a single Virtex-6 chip.

The rest of the paper is organized as follows. Section II gives the background and related work of the packet classification problem. Section III shows our architecture design for the HyperSplit algorithm. Section IV describes two optimizations for the architecture. Section V evaluates the performance of the proposed architecture. Section VI states our conclusion.

II. Background and Related Work

A. Problem Statement

The purpose of multi-dimensional packet classification is to classify packets according to a given rule set. Each rule $\boldsymbol{R}$ has $\boldsymbol{F}$ components, and the $\boldsymbol{f^{th}}$ component of rule $\boldsymbol{R}$, referred to as $\boldsymbol{R}[\boldsymbol{f}]$, is a *range match expression* on the $\boldsymbol{f^{th}}$ field of the packet header. If $\forall \boldsymbol{f} \in [1, \boldsymbol{F}]$, the $\boldsymbol{f^{th}}$ field of the header of a packet $\boldsymbol{P}$ satisfies the range expression $\boldsymbol{R}[\boldsymbol{f}]$, $\boldsymbol{P}$ matches $\boldsymbol{R}$. Table 1 is a practical example of packet classification rules. From a geometric view, all possible values in the $\boldsymbol{F}$ fields of a packet header form a $\boldsymbol{F}$*-dimensional search space* $\boldsymbol{S}$. Each packet $\boldsymbol{P}$ is then a *point* located in $\boldsymbol{S}$, and each rule $\boldsymbol{R}$ is a $\boldsymbol{F}$-dimensional *hyper-rectangle*. If a packet $\boldsymbol{P}$ matches a particular rule $\boldsymbol{R}$, the point represented by $\boldsymbol{P}$ will fall into the hyper-rectangle specified by $\boldsymbol{R}$. Therefore, according to the point location problem in computational geometry [9], the best bounds for packet classification in $\boldsymbol{N}$ non-overlapping hyper-rectangles are $\boldsymbol{\Theta}(\log \boldsymbol{N})$ time with $\boldsymbol{\Theta}(\boldsymbol{N^F})$ space.

B. Packet Classification Algorithms

Although multi-dimensional packet classification has high theoretical complexity, real-life packet classification rules have redundancies that can be exploited to improve classification rate and reduce memory usage [4]. In general, most packet classification algorithms can be divided in two categories. **Parallel search algorithms** and **Tree search algorithms**. Parallel search algorithms, such as RFC [10], HSM [11] and DFCL [12], apply parallel searches on all $\boldsymbol{F}$ fields of a packet header, and then combine the search results using a set of cross-producting tables. Tree search algorithms, including HiCuts [13], HyperCuts [14] and AggreCuts [15], hierarchically decompose the search space on selected field(s) at each non-leaf node, and apply linear search to find the final match at leaf nodes. Table 2 is a summary of existing packet classification algorithms.

C. Related Work

Most existing FPGA implementations of packet classification engines are based on a parallel search or tree search algorithm. Jedhe et al. [16] implemented the DCFL architecture on a Xilinx Virtex-2 Pro FPGA and achieved a throughput of 16 Gbps (40-byte packet) for 128 rules. They predict the throughput can be 24 Gbps on Virtex-5 FPGAs.

Luo et al. [17] proposed a method called *explicit range search* to allow more cuts per node than the original HyperCuts algorithm. The tree height is reduced at the cost of extra memory consumption. At each internal node, a varying number of memory accesses are required to determine which child node to traverse, making it difficult for pipeline mapping.

For improved power efficiency, Kennedy et al. [18] implemented a simplified HyperCuts algorithm on Altera Cyclone-3 FPGA. Because up to hundreds of rules are stored in each leaf node and matched in parallel, the architecture can only operate at low clock frequencies. Moreover, because the search in the tree is not pipelined, their implementation can

Table 3. A two-field example rule set

Rule	*Priority*	*Field-X*	*Field-Y*
R1	1	00~01	00~00
R2	2	00~01	00~11
R3	3	10~10	00~11
R4	4	11~11	11~11
R5	5	11~11	00~11

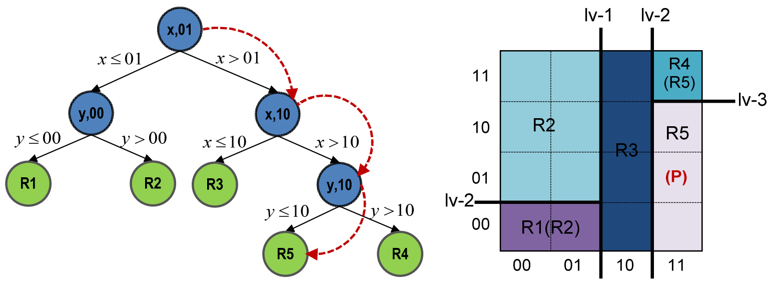

Figure 1. A two-dimensional example of HyperSplit algorithm

only obtain 0.47 Gbps for large rule sets. The design takes as many as 23 clock cycles to classify a packet with 20K firewall rules.

Jiang et al. [19] proposed two optimization methods for the HyperCuts algorithm to reduce memory consumption. By deep pipelining, their FPGA implementation can achieve a throughput of 80 Gbps. However, since the pipeline depth is determined by the decision tree height, their architecture is not practical for decision trees with various heights.

Some researchers use multi-core network processor for high-performance multi-dimensional packet classification. Liu et al. [3] proposed a parallel search algorithm, Bitmap-RFC, which reduces the memory usage of RFC by applying a bitmap compression technique. Qi et al. [4] proposed an optimized tree search algorithm, AggreCuts, using hierarchical bitmap to compress the pointer arrays in each tree node. Although both algorithms support large rule sets and flexible update, their maximum performance is limited to 10 Gbps due to the limitations of the Intel IXP2850 multi-core architecture [20].

III. Architecture Design

A. *Algorithm Motivation*

To reach high-performance and flexible multi-dimensional packet classification, an efficient FPGA-based solution should take the following design considerations:

♦ **Algorithm parallelism:** To achieve wire speed packet classification, the algorithm to be implemented should be able to efficiently exploit the parallelism available on FPGA devices.

♦ **Logic complexity:** To run at high clock rate, the combinational logic in each pipeline stage must be simple and efficient.

♦ **Memory efficiency:** To support large rule sets, the memory usage should be small so as to fit into the on-chip block RAMs.

For this reason, a memory-efficient packet classification algorithm, HyperSplit [4], is selected for the proposed architecture. HyperSplit uses an optimized k-d tree data structure for packet classification. It combines the advantages of both parallel search and tree search algorithms by using a simple but efficient binary search tree for classification. Unlike the basic k-d tree data structure, HyperSplit uses rule-based heuristics to select the most efficient splitting point on a specific field. Figure 1 and Table 3 show a two-dimensional example of the HyperSplit algorithm. From Figure 1, it takes the following steps to classify a packet ***P*** (x=11, y=01). First, at the root node (x, 01), field ***X*** is selected and packet header value x=11 is then compared to the field value stored in the root node, x=01. Because 11 > 01, the search goes to the right child node of the root. Similarly, at the second node (x, 10), the packet header value x=11 is still greater than the stored field value x=10, then it takes the right child. At the third child node (y, 10), the packet header value y=01 is smaller than the field value y=10, so the next node will be the left child which is a leaf node. The search completes at the leaf node, where the best matched rule ***R5*** can be found.

HyperSplit is well suited to FPGA implementation. First, the tree structure of the binary search can be pipelined to achieve high throughput of one packet per clock cycle. Second, the operation at each tree node is simple. Both the value comparison and address fetching can be efficiently implemented with small amount of logic. Additionally, HyperSplit is one of the most efficient algorithms in terms of memory usage compared to best existing algorithms. The memory usage of HyperSplit is at least 10% less [4]. Experimental result shows that even with 50,000 rules HyperSplit consumes less than 6MB of memory. This allows all data structure of the HyperSplit tree to fit into the modern FPGA chips ((e.g. Xilinx Virtex-6).

In the following part of this section, we will present the basic architecture of the HyperSplit algorithm and discuss the design challenges. Then in the next section, we will propose two optimization schemes to further improve the performance and flexibility of the basic architecture.

B. *Basic Architecture*

Because the HyperSplit uses a binary search tree, we map the binary search tree into a pipeline on FPGA. According to Figure 2, all the nodes within a level of the binary search tree are mapped into one stage of the pipeline in the hardware. This architecture allows the incoming packets to be processed in parallel. Because each stage uses one clock cycle to process the packet, the overall classification rate is identical to the clock rate.

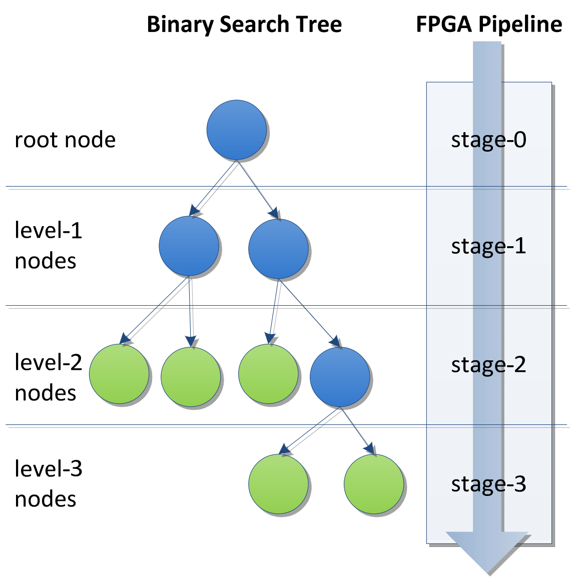

Figure 2. HyperSplit pipeline mapping

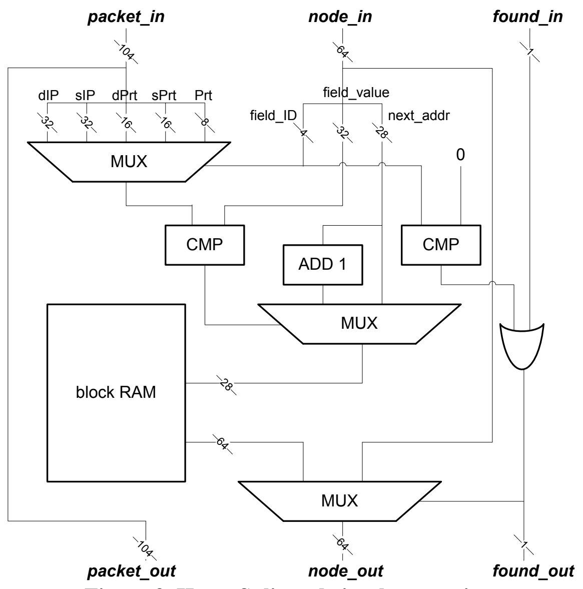

Figure 3. HyperSplit node implementation

In our design, the processing logics for each stage are identical because they perform the same binary search. From Figure 3, each node has 3 inputs, *packet_in*, *node_in* and *found_in*. As the name implies, the *packet_in* is the 104-bit packet header, consisting of the 32-bit *destination_IP_address*, 32-bit *source_IP_address*, 16-bit *destination_port*, 16-bit *source_port*, and 8-bit *L4_protocol* (transport layer protocol). *node_in* is the 64-bit node data loaded from the binary search tree, containing a 32-bit *field_value*, a 4-bit *field_identifier* and a 28-bit *next_node_address*. *found_in* is a 1-bit signal to tell if the best matched rule has been found in previous stages. The outputs of each node are 104-bit *packet_out*, 64-bit *node_out* and 1-bit *found_out*.

Within each stage, the incoming packets are first latched into registers. The output of the register is split up into their corresponding fields and fed into a multiplexer. The multiplexer is controlled by the 4-bit *field_identifier* from *node_in* to choose the corresponding field to compare. Similarly, the 32-bit *field_value* is fed into a comparator along with the output of the multiplexer. The comparison determines whether the output of the multiplexer is less than or equal to the *field_value*. This value in turn drives another multiplexer which selects the left child node at *next_node_address* or the right child node at *next_node_address*+1 (in our design, two child nodes are stored in consecutive entries in a block RAM). This address is immediately fed into the RAM so that at the clock's next rising edge, the data can be read into the next pipeline stage.

To determine whether the node has reached a leaf node, i.e. whether the best matched rule has been found in an earlier stage, we use the following design: First, leaf nodes in this hardware implementation are recognized by checking the *field_identifier*. When this value is 0, it has reached a leaf node, so the signal *found_out* is set to high to tell the next stage that the best matched rule has been found and simply propagate the rule down the pipeline.

The memory in the pipeline is updated by inserting write bubbles [21]. New tree nodes of each pipeline stage are computed offline. When an update is initiated, a write bubble is inserted into the pipeline. Each write bubble is assigned with a *stage_identifier*. If the write enable bit is set, the write bubble will use the new content to update the memory at the stage specified by the *stage_identifier*. This update mechanism supports on-the-fly memory update. Packets preceding the write bubble traverse the old tree while packets following the write bubble look up the new tree.

C. Design Challenges

Although the HyperSplit algorithm is well suited to high performance packet classification architecture on FPGA, there are two major limitations in the current design:

♦ **Number of pipeline stages:** Because the depth of the binary search tree is $\mathbf{\Theta(\log_2 N)}$ [4], the pipeline can be long for large rule sets. For example, with 10K ACL (Access Control List) rules, the pipeline has 28 stages. Deep pipeline will significantly increase the latency of classifying packets resulting in severe problems in packet buffering and system synchronization.

♦ **Block RAM usage:** Because the number of nodes is different at each level of the tree, the memory usage at each stage is not equal. In order to support on-the-fly rule update, the size of block RAMs for each pipeline stage needs to be determined during the implementation of the design. Therefore, an advanced memory allocation scheme is required to control the memory usage of each stage.

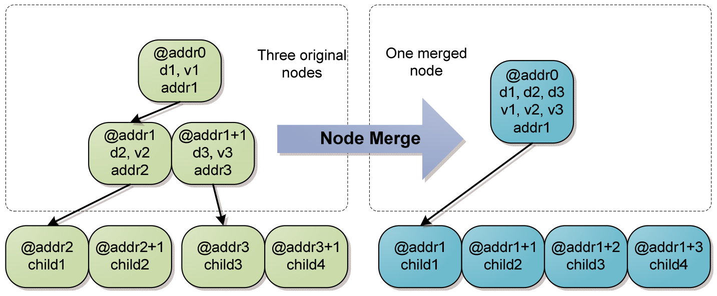

Figure 4. Node merging algorithm

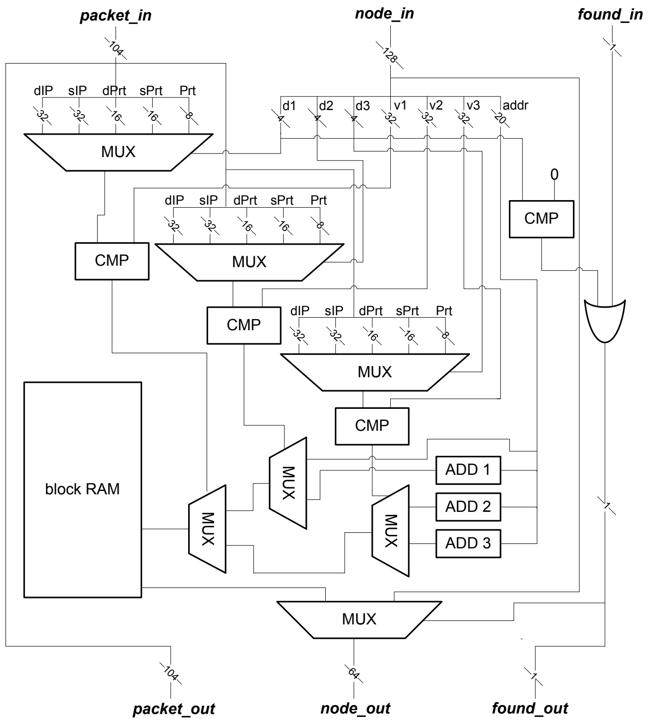

Figure 5. Node merging implementation

Algorithm 1: Node merging

```
// before merging
if pkt[d1] <= v1 then
    node_next = read_ram(addr1+0)
    if pkt[d2] <= v2 then
        node_next = read_ram(addr2+0)
    else
        node_next = read_ram(addr2+1)
    end if
else
    node_next = read_ram(addr1+1)
    if pkt[d3] <= v3 then
        node_next = read_ram(addr3+0)
    else
        node_next = read_ram(addr3+1)
    end if
end if

// After merging
if pkt[d1] <= v1 then
    node_next = (pkt[d2] <= v2) ?
read_ram(addr1+0) : read_ram(addr1+1)
else
    node_next = (pkt[d3] <= v3) ?
read_ram(addr1+2) : read_ram(addr1+3)
end if
```

Figure 6. Node-merging algorithm

Algorithm 2: Leaf pushing

```
Initialize bucket[i], i=0, ..., tree_height-1
node = root
depth = 0
Reshape (node, depth) begin
    if node is leaf then
        while bucket[depth]==0 do depth ++
        end while
        bucket[depth]--
        node->depth = depth
    else
        Reshape (node->child[0], depth+1)
        Reshape (node->child[1], depth+1)
        Reshape (node->child[2], depth+1)
        Reshape (node->child[3], depth+1)
    end if
end
```

Figure 7. Leaf-pushing algorithm

Our solution to the two major problems is described in detail in the next section.

IV. Architecture Optimization

To solve the problems of the basic design in Section III, we propose a node-merging algorithm to reduce the number of pipeline stages, and a leaf-pushing algorithm to control the memory usage. Furthermore, an improved pipeline implementation is used to enable dual-packet classification per clock cycle.

A. Pipeline Depth Reduction

Because the number of pipeline stages is equal to the height of the HyperSplit tree, we implemented a node merging algorithm to reduce the tree height. The node-merging scheme merges a non-leaf node with its two children into a single node. Thus a two-stage search is done in a single stage. The basic idea of the algorithm is shown in Figure 4. From this figure we can see that three non-leaf nodes (left) of the original HyperSplit tree are merged into a single node (right). The merged node stores all field identifiers and fields values of the original nodes. Figure 5 shows the implementation of the merged nodes. Figure 6 is the pseudo code of the node-merging algorithm.

After node merging, the height of the tree is reduced from $\log_2 N$ to $\log_4 N$. Thus the number of pipeline stages is half of the basic design. In addition, because no extra information is stored in the merged node, the size of a merged node is no greater than the total size of the original three nodes. Therefore the node merging scheme will not increase the overall memory usage.

B. Controlled Block RAM Allocation

Because each pipeline stage has its own dedicated block RAMs, if the number of nodes changes we need to reallocate

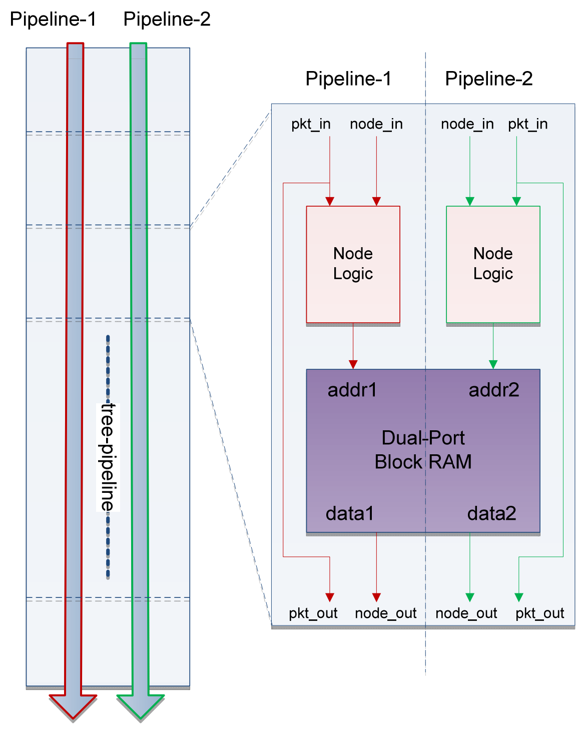

Figure 8. Dual-search pipeline implementation

block RAMs for each stage. However, block RAM reallocation will result in reconfiguring the FPGA device, so the rules cannot be updated on-the-fly with the basic design.

Our optimization maps the HyperSplit tree onto a linear pipeline with controlled memory distribution over each stage. The proposed leaf-pushing algorithm is based on the following two features of the HyperSplit algorithm:

First, because the number tree nodes in the top *l*-level is small (less than 4^l), it is not efficient to instantiate the 1024-entry (or larger [19]) block RAMs for those levels. Instead, we allow those entries in distributed RAMs.

Second, because pushing down the leaf nodes does not change the search semantics of the original tree [19] and the number of leaf nodes is comparable to non-leaf nodes [4], we can reshape the tree by pushing leaf nodes down to lower levels to reduce the memory usage at certain stages.

The pseudo code leaf-pushing algorithm is shown in Figure 7. With this algorithm, the memory usage at the *l*-th stage can be limited by *bucket*[*l*]. The maximum value of *bucket*[*l*] can be determined by the overall available memory of the FPGA chip. If we set *bucket*[*l*] to have the same value, a balanced memory allocation is achieved, i.e. the number of block RAMs pre-allocated for all pipeline stages (l>5) shares the same value.

C. *Dual-packet Search Architecture*

To further improve the performance, we exploit the advanced features of modern FPGA. Because the block RAMs support dual-port reads, i.e. two memory reads can be done in a single clock cycle, a dual-packet search pipeline can be implemented to achieve a 2x speedup. The new pipeline in Figure 8 uses the same dual-port block RAM in each stage with two search logics to classify two packets per clock cycle.

V. Performance Evaluation

A. *Test bed and data set*

We evaluated the effectiveness of our design by conducting experiments with publicly available 5-dimensional packet classification rule sets [8]. The number of rules in our tested rule sets ranges from 100 to 50,000. The rule sets in our tests are denoted by *acl1_num*. For example, *acl1_10K* represents the rule set with 10,000 rules and generated by the Classbench with *acl1* seeds [22].

The FPGA device used in our tests is the Xilinx Virtex-6, (model: XC6VSX475T) containing 7,640 Kb Distributed RAM and 38,304 Kb block RAMs. All the experimental results on FPGA are obtained by post place and route simulation. To compare our design with state-of-the-art multi-core solutions, we have also implemented the HyperSplit algorithm on a 16-MIPS-core Cavium OCTEON3860 processor, which is the same platform used in existing work [4].

B. *Node-merging Optimization*

Figure 9 and Figure 10 show the effectiveness of the node-merging algorithm. From Figure 9, the node-merging algorithm effectively reduces the pipeline stages. Regardless of the size of the rule set, the node-merging algorithm reduces the number of pipeline stages of the basic design (without node merging) by 50%. According Figure 10, the memory usage is increased slightly by the node-merging scheme.

C. *Leaf-pushing Optimization*

Because the number of block RAMs is predetermined, to support on-the-fly rule update, the leaf-pushing algorithm is used to limit the number of nodes in each stage. According to Figure 11, we can see that without leaf-pushing, the memory required by each pipeline stage is uncontrolled. In comparison, Figure 12 shows the memory distribution after leaf-pushing. In this test, we use a balanced memory allocation scheme and set each *bucket*[*l*] to 10,000. Compared to Figure 11, the memory usage of each stage is strictly limited to the bucket size. This allows the new data structure to be adjusted to fit within the original block RAM sizes without reconfiguring the FPGA.

D. *FPGA Performance*

Table 4 shows the performance of our FPGA implementation. The maximum clock rate is obtained from the post place and route report. Even with 10K rules, the clock achieved is 115.4 MHz. This is equivalent to a throughput of 118 Gbps for the minimum-size (64 bytes) Ethernet packets.

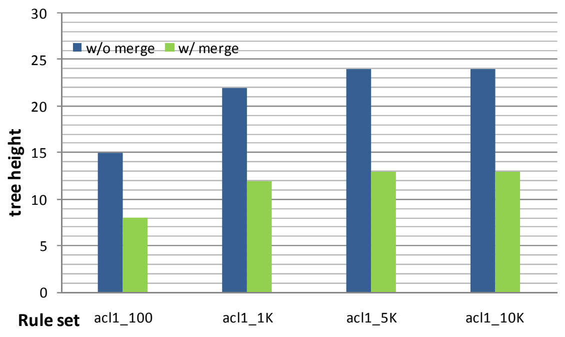

Figure 9. Tree heights with and without node-merging

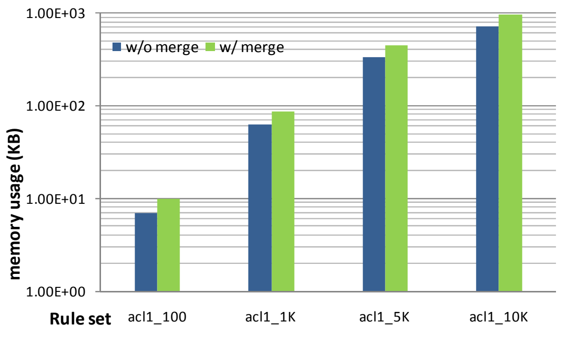

Figure 10. Memory usage with and without node-merging

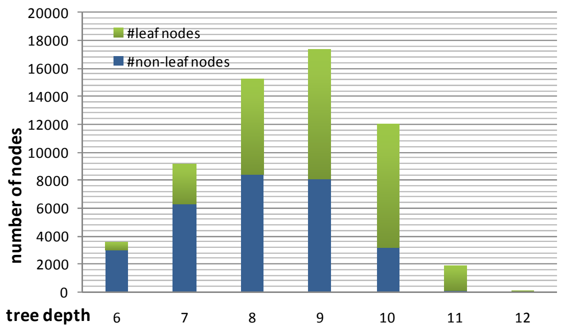

Figure 11. Nodes distribution without leaf pushing

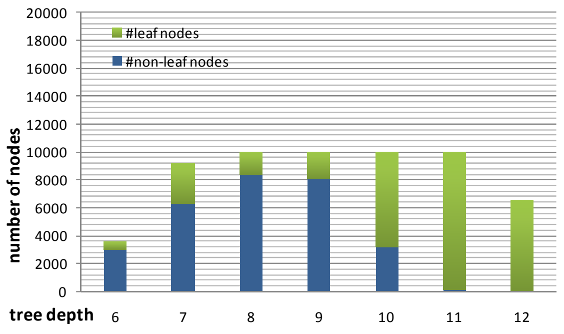

Figure 12. Nodes distribution with leaf pushing

Table 4. FPGA performance and resource utilization

Rules	*Max Clock (MHz)*	*Max Thrupt (Gbps)*	*Tree depth*	*#slices used / available*	*#RAMs used / available*
acl1_100	139.1	142	7	444/37440	10/516
acl1_1K	134.0	137	11	602/37440	18/516
acl1_10K	115.4	118	12	747/37440	103/516

Table 5. Comparison with FPGA-based approaches

Approaches	*Max #rules*	*Max Thrupt (Gbps)*	*for acl1_10K*		
			Pipeline depth	**#slices used**	**#RAMs used**
Our approach	50K	142	12	747	103
HyperCuts on FPGA [19]	10K	128	20	10307	407
HyperCuts Simplified [17]	10K	7.22	--	--	--

Table 6. Comparison with multi-core based approaches

Approaches	*Max Throughput (Gbps)*
Our approach	142
HyperSplit on OCTEON [4]	6.4
RFC on IXP2850 [3]	10

Table 5 compares our approach with start-of-the-art FPGA-based packet classification solutions. From this table, our approach uses significantly less block RAMs (e.g. 103 vs. 407 for *acl1_10K*). This is mainly because our approach does not store the original rules at each pipeline stage for doing the linear search. According to our test, we can support more than 50,000 rules with a single Virtex-6 chip (model: XC6VSX475T). The number of slices used by our approach is also significantly smaller (747 vs. 10307 for *acl1_10K*). One reason is our approach uses a linear pipeline while their approach is a two dimensional pipeline. Another reason is their approach has deeper pipeline due to multiple linear searches at leaf nodes.

Table 6 compares the performance of our approach with state-of-the-art multi-core based solutions. From this table we can see that our approach has at least a 10x speedup compared with based solutions. The main performance bottleneck of I/O bandwidth and latency is solved in our approach by using on-chip block RAMs.

VI. Conclusion

With the rapid growth of network bandwidth, wire speed multi-dimensional packet classification has become a challenge for next-generation network processing devices. In this paper, we propose an FPGA-based architecture to support 100 Gbps packet classification. Two optimization algorithms are proposed to reduce the number of pipeline stages and control the memory usage in each stage. The implementation results show that our architecture can achieve more than 100 Gbps throughput for the minimum-size Ethernet packet. Compared to state-of-the-art FPGA based solutions, our approach uses significantly less FPGA resources and can support over 50K rules with a single FPGA chip. Compared to multi-core based solutions, our approach has at least a 10x speedup.

REFERENCES

[1] P. Gupta and N. McKeown. Algorithms for packet classification. IEEE Network, 15(2):24–32, 2001.

[2] D. E. Taylor. Survey and taxonomy of packet classification techniques. ACM Comput. Surv., 37(3):238–275, 2005.

[3] D. Liu, B. Hua, X. Hu and X. Tang. High-performance packet classification algorithm for many-core and multithreaded network processor. In Proc. CASES, 2006.

[4] Y. Qi, L. Xu, B. Yang, Y. Xue, J. Li. Packet classification algorithms: from theory to practice. In Proc. Infocom'09, 2009.

[5] Ethernet Alliance, 40 Gigabit Ethernet and 100 Gigabit Ethernet Technology Overview White Paper, 2007.

[6] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In Proc. SIGCOMM, 2005.

[7] C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. In Proc. ICNP, 2007.

[8] http://www.arl.wustl.edu/~hs1/PClassEval.html

[9] M. H. Overmars and A. F. van der Stappen. Range searching and point location among fat objects. Journal of Algorithms, 21(3), 1996.

[10] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In Proc. SIGCOMM, 1999.

[11] B. Xu, D. Jiang and J. Li. HSM: A Fast Packet Classification Algorithm. In Proc. AINA, 2005.

[12] D. E. Taylor and J. S. Turner. Scalable packet classification using distributed crossproducing of field labels. In Proc. INFOCOM, 2005.

[13] P. Gupta and N. McKeown. Classifying packets with hierarchical intelligent cuttings. IEEE Micro, 20(1):34–41, 2000.

[14] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In Proc. SIGCOMM, 2003.

[15] Y. Qi, B. Xu, F. He, B. Yang. J. Yu, J. Li. Towards high-performance flow-level packet processing on multi-core network processors. In Proc. ANCS, 2007.

[16] G. S. Jedhe, A. Ramamoorthy, and K. Varghese. A scalable high throughput firewall in FPGA. In Proc. FCCM, 2008.

[17] Y. Luo, K. Xiang, and S. Li. Acceleration of decision tree searching for IP traffic classification. In Proc. ANCS, 2008.

[18] A. Kennedy, X. Wang, Z. Liu, and B. Liu. Low power architecture for high speed packet classification. In Proc. ANCS, 2008.

[19] W. Jiang and V. K. Prasanna. Large-scale wire-speed packet classification on FPGAs. In Proc. FPGA, 2009.

[20] Intel IXP2850 Network Processor Hardware Reference Manual. http://int.xscale-freak.com/XSDoc/IXP2xxx/

[21] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In Proc. INFOCOM, 2003.

[22] D. E. Taylor and J. S. Turner. Classbench: a packet classification benchmark. IEEE/ACM Trans. Netw., 15(3):499–511, 2007.