# Scalable Packet Classification: Cutting or Merging?

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

*Abstract*—**Multi-field packet classification is a fundamental function that enables routers to support a variety of network services. Most of the existing multi-field packet classification algorithms can be divided into two classes: cutting-based and merging-based solutions. However, neither of them is scalable with respect to memory requirement for all rule sets with various characteristics. This paper makes several observations on real-life rule sets and proposes a novel hybrid scheme to leverage the desirable features of the two classes of algorithms. We propose a SRAM-based parallel multi-pipeline architecture to achieve high throughput. Several challenges in mapping the hybrid algorithm onto the architecture are addressed. Extensive simulations and FPGA implementation results show that the proposed scheme sustains 80 Gbps throughput for minimum size (40 bytes) packets while consuming a small amount of on-chip resources for large rule sets consisting of up to 10K unique entries.**

## I. Introduction

As the Internet evolves, it becomes necessary for next-generation routers to support a variety of network applications such as firewall processing, Quality of Service differentiation, traffic billing, and other value added services. These applications are enabled by multi-field packet classification where an Internet Protocol (IP) packet is classified based on the multiple fields in the packet header such as the 32-bit source/destination IP addresses (denoted **SA/DA**), 16-bit source/destination port numbers (denoted **SP/DP**), and 8-bit transport layer protocol (denoted **Prtl**). Individual entries for classifying a packet are called *rules* which specify the value ranges of each field in the packet header. Due to the rapid growth of the rule set size, as well as the link rate, multi-field packet classification has become one of the fundamental challenges in 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 [1].

Most of recent research on high-throughput packet classification engines are based on ternary content addressable memories (TCAMs) [2]–[4] or various hashing-based schemes such as Bloom Filters [5]–[7]. However, as shown in [8]–[10], TCAMs are not scalable with respect to clock rate, power consumption, or circuit area, compared to static random access memories (SRAMs). Most of TCAM-based solutions also suffer from range expansion when converting ranges into prefixes [3], [4]. Bloom Filters have become popular due to their $O(1)$ time performance and low memory requirement. However, a secondary module is needed to resolve false positives inherent in Bloom Filters. Such a secondary module may be slow and can limit the overall performance [11].

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). We are aware of the time-storage tradeoff in designing packet classification algorithms for software-based implementation [12]. However in hardware, the time cost in terms of the number of memory accesses can be alleviated using pipeline architectures, while the limited amount of on-chip memories becomes the major constraint. Thus our main concern is to develop scalable packet classification algorithms with low memory requirement so that large rule sets (e.g. 10K entries) can be supported on a single device.

Most existing packet classification algorithms fall into two classes: cutting-based and merging-based approaches. The *cutting*-based algorithms (e.g. HyperCuts [13]), cut the multi-dimensional search space recursively, resulting in a tree with various number of branches at each internal node. Such algorithms usually scale well and are suitable for rule sets in which the rules have little overlap with each other. The *merging*-based algorithms (e.g. BV [14]), perform independent search on each field and finally merge the search results from all the fields. Such algorithms are desirable for hardware implementation due to their parallel search on multiple fields. But substantial storage is usually needed to merge the single-field search results into the final result [9].

We observe that neither of the above two types of algorithms exhibits good storage scalability for all types of rule sets with different characteristics. Cutting-based algorithms suffer from rule overlapping, though they usually outperform merging-based algorithms for large rule sets. Merging-based algorithms have poor storage scalability, but have consistent performance for various rule sets regardless of the overlap among the rules. We further observe that most of the rule overlaps can be reduced by removing few rules from the original rule set. Based on the motivation to combine cutting-based and merging-based algorithms to achieve better storage scalability, this paper makes following contributions.

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

- By conducting experiments on several real-life rule sets, we identify the pros and cons of the representative algorithms of cutting-based and merging-based approaches, respectively. A new metric, named $Overlap_Degree$, is proposed to quantify the rule overlap.
- A hybrid scheme is proposed to combine the desirable features of cutting-based and merging-based algorithms. We extract the rules that have most overlap with others from the original rule set. These rules are placed in a separate subset called the *common set*. The BV algorithm is used to search the common set whose size is usually small. The remaining rules, with much less overlap, are built into a decision tree using the HyperCuts algorithm.
- The search process is parallelized by placing the common set and the decision tree in two separate modules. Both the single-field search in BV and the decision tree search in HyperCuts are regarded as some form of tree traversal, which can be pipelined to achieve high throughput.
- Several challenges in mapping the decision tree onto a pipeline architecture are addressed. We introduce a fine-grained node-to-stage mapping scheme which balances the memory distribution across the stages by allowing two tree nodes on the same level to be mapped onto different stages. We also present the circuit design which enables on-the-fly updating the number of branches on multiple packet header fields in each decision tree node.
- Simulation experiments using various real-life data sets show that the proposed hybrid scheme achieves much better storage scalability than either cutting-based or merging-based algorithms. Hardware implementation results show that our architecture can store 10K 5-field rules in a single state-of-the-art FPGA, and it sustains 80 Gbps throughput for minimum size (40 bytes) packets.

The rest of the paper is organized as follows. Section II reviews the HyperCuts [13] and the BV [14] algorithms, which are representative of cutting-based and merging-based algorithms, respectively. Section III analyzes the impact of rule overlap on the performance of HyperCuts and BV, and proposes a hybrid scheme. Section IV presents the SRAM-based pipeline architecture. Section V evaluates the performance of our solution. Section VI concludes the paper.

## II. Background

A vast number of packet classification algorithms have been published over the past decade (see [1], [9] for a survey). As representatives of cutting-based and merging-based algorithms, respectively, HyperCuts [13] and BV [14] are considered among the most scalable algorithms.

HyperCuts [13] is a cutting-based algorithm. It takes a geometric view of the packet classification problem. Each rule is abstracted as a hypercube in a $D$-dimensional space where $D$ is the number of fields in a rule. Each packet header defines a point in this $D$-dimensional space. Given a set of rules, the HyperCuts algorithm employs several heuristics to cut the space recursively into smaller subspaces, along multiple dimensions in each step. Each subspace ends up with fewer rules, until the number of rules contained by the subspace is small so that it permits a low-cost linear search to find the best matching rule. Such cutting procedure results in a decision tree where different nodes have various number of branches. After the decision tree is built, the algorithm to classify a packet is simple. Based on the value of the packet header, the search 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. Figure 1 shows a geometrical representation of a set of 2-field rules and an example HyperCuts decision tree for the rule set. Like other cutting-based packet classification algorithms, HyperCuts suffers from memory explosion due to rule duplication. For example, as shown in Figure 1, rules R1 and R2 are replicated into multiple children nodes. We identify the overlap between different rules as one major source of rule duplication. Taking the rule set in Figure 1 as an example, since R1 overlaps with R2, R3 and R6, no matter how we cut the space, R1 will be replicated into the nodes which contain R2, R3 or R6. Let $N$ denote the number of rules and $D$ the number of fields in a rule. The worst-case memory requirement of HyperCuts is $O(N^D)$. On the other hand, without rule duplication, HyperCuts requires $O(N)$ memory only.

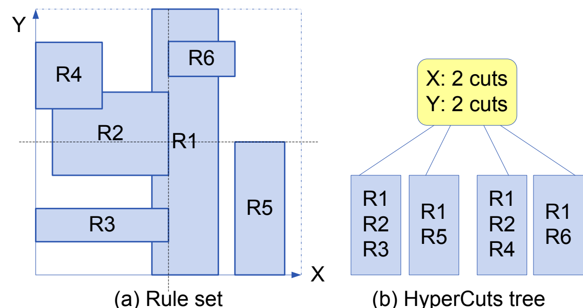


Fig. 1. Example of HyperCuts decision tree. ((a) X and Y axes correspond to the 2 fields of the rules. (b) A rounded rectangle in yellow denotes an internal tree node, and a rectangle in gray a leaf node.)

The BV algorithm [14] is a merging-based algorithm targeted for hardware implementation. First it performs search on each individual field, concurrently for all the fields. This step can be conducted via some standard single field lookup technique, such as binary search and longest prefix match (LPM). The search on each field returns a bit vector where each bit represents a rule. A bit is set to “1” if the corresponding rule is matched on this field; otherwise the bit is reset to “0”. Then the result of the bitwise AND operation on these bit vectors from all fields indicates the set of rules that matches a given packet. BV can provide 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 $U * N * D = O(N^2)$.

Intuitively, effective combination of HyperCuts and BV algorithms should result in memory requirement between $O(N)$ and $O(N^2)$.

## III. Analysis and Algorithms

### A. Analysis and Motivation

As discussed in Section II, overlapping rule can affect the memory requirement of packet classification algorithms. We propose a new metric, called $Overlap_Degree$, to quantify the rule overlap. Equations (1) and (2) define the $Overlap_Degree$ for a rule and for a rule set, respectively, where $r$ denotes a rule, $R$ a rule set, $N_o(r)$ the number of rules overlapping with $r$, and $|R|$ the number of rules in $R$.

$$Overlap_Degree(r) = \frac{N_o(r)}{|R|} \quad (1)$$

$$Overlap_Degree(R) = \frac{\sum_{r \in R} Overlap_Degree(r)}{|R|} \quad (2)$$

We conducted experiments on 6 real-life and synthetic 5-field rule sets from [15] to compare the memory requirement (in terms of bytes per rule) of HyperCuts and BV. Several configurable parameters of the HyperCuts algorithm were set as follows: $bucketSize = 32$, $spfac = 2$. Table I shows the $Overlap_Degree$ of the rule sets as well as the experimental results for both HyperCuts and BV. For the small rule sets, BV achieved lower memory requirement when the $Overlap_Degree$ becomes higher, while HyperCuts suffered from rule overlaps. For the large rule sets, the memory requirement of HyperCuts was affected heavily by the $Overlap_Degree$ of the rule set, while that of BV was more consistent. HyperCuts outperformed BV for rule sets with little rule overlap, while BV required much lower memory for rule sets with high rule overlapping.

TABLE I
Memory requirement of HyperCuts and BV

| Rule set | # of rules | $Overlap_Degree$ | Bytes/rule | |
|---|---|---|---|---|
| | | | HyperCuts | BV |
| $acl1$ | 752 | 0.0101 | 32.58 | 71.80 |
| $ipc1$ | 1550 | 0.0161 | 128.52 | 61.57 |
| $fw1$ | 269 | 0.1327 | 399.18 | 40.72 |
| $acl_10K$ | 9603 | 0.0007 | 54.22 | 789.22 |
| $ipc_10K$ | 9037 | 0.0051 | 2378.35 | 788.69 |
| $fw_10K$ | 9311 | 0.0964 | 12554.18 | 1582.18 |

We also conducted experiments on these rule sets to identify the source of rule overlap. We counted the $Overlap_Degree$ for each rule. Then we obtained the number of rules over different $Overlap_Degree$. Figure 2 shows the rule distribution over $Overlap_Degree$ for all 6 rule sets. Logarithmic (base 10) scale is used for the X axis. The Y axis shows the normalized values obtained by dividing the number of rules with a specific $Overlap_Degree$ by the total number of rules. According to Figure 2, most of rules in a rule set have little overlap with other rules.

Based on the above observations, we propose to partition the original rule set into two subsets. One subset contains a small number of rules with high $Overlap_Degree$. We call this subset the *common set*. The rules in the common set, called *common rules*, can be matched using merging-based algorithms such as BV. The other subset contains the

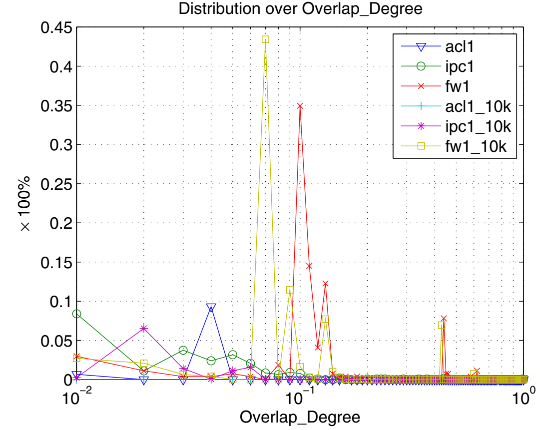


Fig. 2. Rule distribution over $Overlap_Degree$

remaining rules with much lower $Overlap_Degree$, which are suitable for using cutting-based algorithms such as HyperCuts. Figure 3 illustrates the application of our idea for the example rule set shown in Figure 1.

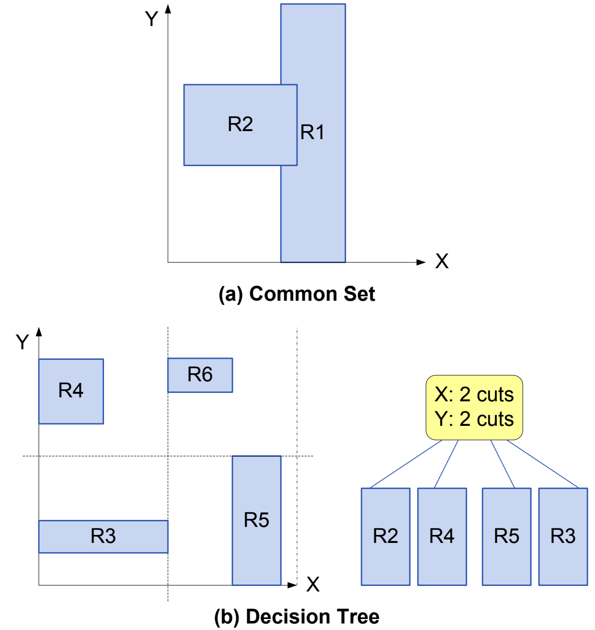


Fig. 3. Motivating example

### B. Building the Decision Tree and the Common Set

Motivated by the above idea, we integrate the common set construction into the HyperCuts tree building algorithm, as shown in Figure 4, where $n$ denotes a node, $r$ a rule and $NB$ the number of bits selected. The algorithm cuts the multi-dimensional search space until the resulting node contains no more than $bucketSize$ rules. Then this node becomes a leaf node. At each step, we use a similar method as HyperCuts to choose the optimal number of cutting bits from each field. After the bits from each field is selected, we identify the common rules and push them into the common set. We set a $Threshold$, whose value is between 0 and 1. A rule $r$ is

identified as a *common rule* if it satisfies condition (3), where $K$ denotes the total number of children nodes of the current node and $K_C(r)$ the number of children nodes containing $r$. A parameter called $commonSize$ is used to configure the maximum number of rules in the common set.

$$\frac{K_C(r)}{K} >= Threshold \tag{3}$$

1: Initialize the root node and push it into $nodeList$.
2: **while** $nodeList \neq null$ **do**
3: $n \leftarrow Pop(nodeList)$
4: **if** $n.numrule < bucketSize$ **then**
5: Push $n$ into the list of leaf nodes.
6: Continue.
7: **end if**
8: $NB_{SA} \leftarrow OptNumBit(n.SA)$
9: $NB_{DA} \leftarrow OptNumBit(n.DA)$
10: $NB_{SP} \leftarrow OptNumBit(n.SP)$
11: $NB_{DP} \leftarrow OptNumBit(n.DP)$
12: $NB_{Prtl} \leftarrow OptNumBit(n.Prtl)$
13: $NB = NB_{SA} + NB_{DA} + NB_{SP} + NB_{DP} + NB_{Prtl}$
14: **while** $commonSet.size < commonSize$ **do**
15: $r \leftarrow CommonRule(n, NB_{SA}, NB_{DA}, NB_{SP}, NB_{DP}, NB_{Prtl})$
16: Push $r$ into $commonSet$.
17: **end while**
18: **for** $i \leftarrow 0$ to $2^{NB} - 1$ **do**
19: $n_i \leftarrow CreateNode(n, i)$
20: Push $n_i$ into $nodeList$.
21: **end for**
22: **end while**

Fig. 4. Algorithm: Building the decision tree and the common set

The BV algorithm is used to match the rules in the common set. For each field, a quad search tree is built using the endpoints of the elementary ranges, which are the projections of the rules on the field. Range matching is performed by traversing the quad tree. Each elementary range corresponds to a bit vector indicating the possible matching rules.

# IV. HARDWARE ARCHITECTURE

Field-programmable gate array (FPGA) technology has become an attractive option for implementing real-time network processing engines [4], [7], [16], due to its ability to reconfigure and massive parallelism. State-of-the-art SRAM-based FPGA devices such as Xilinx Virtex-5 [17] provide high clock rate and large amount of on-chip dual-port memory with configurable word width.

## A. Architecture Overview

We propose a SRAM-based multi-pipeline architecture, shown in Figure 5. The common set and the decision tree are implemented as two separate modules. For each input packet, we perform the search on the two modules in parallel. The search results from common set and the decision tree are directed to the priority resolver, which finally outputs the rule with the higher priority. We also exploit the dual-port RAMs provided by state-of-the-art FPGAs to achieve high throughput of two packets per clock cycle.

## B. Search in Common Set

We employ quad trees to perform the range search on each field of the packet header. The search process is pipelined by assigning each tree level to a separate memory block. Instead of explicitly storing the pointers of children nodes at each tree node, we obtain the address of the children node by cascading the address of the current node with the comparison results at the current level. The search process ends at leaf nodes and returns the matching bit vector for the field.

After retrieving the bit vectors from all the fields, bitwise AND operations are performed to merge the bit vectors. Then the final matching bit vector is directed to the priority encoder to retrieve the ID of the highest-priority rule.

## C. Mapping the Decision Tree onto Pipeline

Although the HyperCuts algorithm has been claimed to be easily pipelined [13], we are not aware of its hardware implementation. It is unclear how to implement a decision tree where each node has various number of branches. A simple solution is hard-wiring the connections, which however cannot update the tree structure on-the-fly. We propose a circuit design using left and right shifters. Given an input packet field $f$, its left shift offset is the number of cutting bits of $f$, while its right shift offset is the sum of the number of cutting bits of preceding fields. The shifted-out value of $f$ is then ORed with those from other fields to get the memory address offset for accessing the children nodes. The number of cutting bits for each field is stored in memory. Thus, we can update the memory to change the decision tree structure, as the rules are updated.

Moreover, the size of the memory in each pipeline stage must be determined before hardware implementation. The static mapping scheme is to simply map each level of the decision tree onto a separate stage. However, the memory distribution across the stages can vary widely. Allocating memory with the maximum size for each stage can result in large memory wastage [18]. The problem is to map the decision tree onto a linear pipeline to achieve balanced memory distribution over the stages, while sustaining a throughput of two packets per clock cycle. The challenge comes from the various number of words needed for different tree nodes.

The above problem is a variant of packing problems, and can be proved to be NP-complete. We use a fine-grained mapping scheme similar as [19] to allow the nodes on the same level of the tree to be mapped onto different pipeline stages. This provides more flexibility in mapping the tree nodes, and helps achieve a balanced memory distribution across the stages in a pipeline. There is only one constraint to 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.

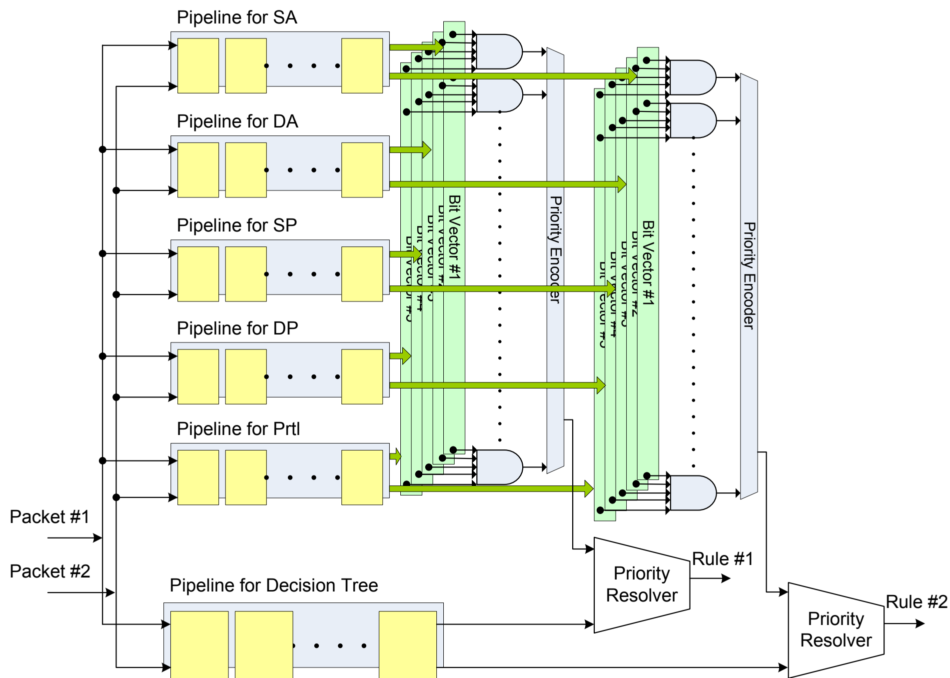


Fig. 5. Pipeline architecture for the hybrid scheme

We use a simple method to enable such fine-grained node-to-stage mapping. Each node stored in 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 passes 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.

### D. Rule Update

Since our architecture is linear and memory-based, on-the-fly update without disturbing the ongoing operations is feasible. We update the memory in the pipeline by inserting *write bubbles* [20]. The new content of the memory is computed offline. When an update is initiated, a write bubble with an assigned ID is inserted into the pipeline. Each stage has a write bubble table. The table stores 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. The bubble then 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 new content will be used to update the specified memory location of the next stage. All packets preceding or following the write bubble can perform their operations without interruption.

## V. Performance Evaluation

To evaluate the effectiveness of our schemes, we conducted simulation experiments using rule sets from [15]. These synthetic rule sets are generated using ClassBench [21], with parameter files extracted from real-life rules. The size of the rule sets varies from hundreds of to tens of thousands of rules.

### A. Algorithm Evaluation

The key performance metric is the average memory size per rule which represents the scalability of the algorithm. We set the same values for these parameters shared by HyperCuts: $bucketSize = 32$, $spfac = 2$, which were the typical settings for HyperCuts. The other parameters for our scheme were $commonSize = 128$, $Threshold = 1$. A series of our experiments showed that such settings were near optimal. Due to space limitation, the detailed results are omitted here. As Table II shows, our scheme outperformed both HyperCuts and BV with respect to storage scalability.

TABLE II
Memory requirement of algorithms for various rule sets

| Rule sets | # of rules | Bytes/rule | | |
|---|---|---|---|---|
| | | Our scheme | HyperCuts | BV |
| $acl_100$ | 98 | 24.44 | 27.78 | 47.35 |
| $acl_1K$ | 916 | 22.98 | 38.15 | 91.63 |
| $acl_5K$ | 4415 | 24.83 | 59.64 | 257.23 |
| $acl_10K$ | 9603 | 25.51 | 54.22 | 789.22 |
| $ipc_100$ | 99 | 23.65 | 24.57 | 69.16 |
| $ipc_1K$ | 938 | 25.63 | 61.34 | 176.03 |
| $ipc_5K$ | 4460 | 49.46 | 406.80 | 358.61 |
| $ipc_10K$ | 9037 | 43.30 | 2378.35 | 788.69 |
| $fw_100$ | 92 | 56.63 | 113.37 | 27.46 |
| $fw_1K$ | 791 | 215.06 | 6110.58 | 67.08 |
| $fw_5K$ | 4653 | 255.13 | 16132.65 | 691.69 |
| $fw_10K$ | 9311 | 248.54 | 12554.18 | 1582.18 |

### B. Architecture Implementation

We mapped the largest rule set, $acl_10K$, onto the proposed pipeline architecture with 16 stages. As Figure 6 shows, our mapping scheme achieved much more balanced memory distribution across the stages than the static mapping scheme.

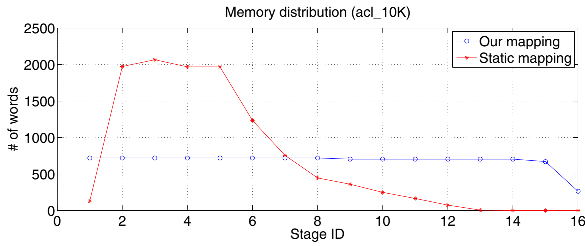


Fig. 6. Memory distribution over 16 stages for $acl_10K$

TABLE III
RESOURCE UTILIZATION

| | Used | Available | Utilization |
|---|---|---|---|
| Number of Slices | 6,557 | 30,720 | 21% |
| Number of bonded IOBs | 236 | 960 | 24% |
| Number of Block RAMs | 321 | 456 | 70% |

Based on the mapping results for $acl_10K$, we implemented our design on FPGA using 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 achieved a clock frequency of 125 MHz while consuming a small amount of on-chip resources as shown in Table III.

Table IV compares our design with the state-of-the-art hardware-based packet classification engines. For fair comparison, the results of the FPGA-based work were scaled to Xilinx Virtex-5 based on the maximum clock frequency [1]. The values in parentheses are the original data reported in the corresponding papers. Considering the time-storage tradeoff in various designs, we used a new performance metric, called *Efficiency*. *Efficiency* was defined as the ratio of the throughput to the average amount of memory per rule. Our design outperformed the other designs including the ASIC-based solutions with respect to both throughput and *Efficiency*.

## VI. CONCLUSION

This paper analyzed the impact of rule overlap on the scalability of cutting-based and merging-based packet classification algorithms. A hybrid scheme was proposed to leverage the desirable features of both algorithms so that the low memory requirement was achieved for rule sets with various characteristics. We mapped the hybrid scheme onto a SRAM-based multi-pipeline architecture and addressed several implementation challenges. Extensive simulation and FPGA implementation results show that our solution can store large rule sets on a single device. It sustains 80 Gbps throughput, which is twice the current backbone network link rate. To the best of our knowledge, this design is among few packet classification engines achieving over 40 Gbps throughput while supporting 10K unique rules. Our future work includes developing adaptive schemes to automatically find the optimal $commonSize$ for different rule sets.

[1]The BV-TCAM paper [4] does not present implementation results regarding the throughput. We used the predicted value given in [4].

## REFERENCES


[1] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.
[2] 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, 2005.
[3] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in *Proc. SIGCOMM*, 2005, pp. 193–204.
[4] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *Proc. FPGA*, 2005, pp. 238–245.
[5] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using bloom filters," in *Proc. ANCS*, 2006.
[6] I. Papaefstathiou and V. Papaefstathiou, "Memory-efficient 5D packet classification at 40 Gbps," in *Proc. INFOCOM*, 2007, pp. 1370–1378.
[7] A. Nikitakis and I. Papaefstathiou, "A memory-efficient FPGA-based classification engine," in *Proc. FCCM*, 2008, pp. 53–62.
[8] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?" in *Proc. INFOCOM*, 2003.
[9] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, 2005.
[10] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup," in *Proc. INFOCOM*, 2008, pp. 1786–1794.
[11] I. Sourdis, "Designs & algorithms for packet and content inspection," Ph.D. dissertation, Delft University of Technology, 2007. [Online]. Available: http://ce.et.tudelft.nl/publicationfiles/1464_564_sourdis_phdthesis.pdf
[12] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in *Proc. INFOCOM*, 2000, pp. 1193–1202.
[13] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. SIGCOMM*, 2003.
[14] 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.
[15] Packet Classification Filter Sets, "http://www.arl.wustl.edu/~hs1/pclasseval.html."
[16] G. S. Jedhe, A. Ramamoorthy, and K. Varghese, "A scalable high throughput firewall in FPGA," in *Proc. FCCM*, 2008, pp. 43–52.
[17] Xilinx Virtex-5 FPGAs, "http://www.xilinx.com."
[18] 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.
[19] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *Proc. FPGA*, 2009.
[20] A. Basu and G. Narlikar, "Fast incremental updates for pipelined forwarding engines," in *Proc. INFOCOM*, 2003, pp. 64–74.
[21] D. E. Taylor and J. S. Turner, "Classbench: a packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.
[22] A. Kennedy, X. Wang, Z. Liu, and B. Liu, "Low power architecture for high speed packet classification," in *Proc. ANCS*, 2008, pp. 131–140.


TABLE IV
PERFORMANCE COMPARISON

| Packet classification engines | Platform | # of rules | Total memory used (Kbytes) | Throughput (Gbps) | Efficiency (Gbps/KB) |
|---|---|---|---|---|---|
| Our approach | FPGA | 9603 | 245 | 80 | 3135.67 |
| Optimized HyperCuts [19] | FPGA | 9603 | 612 | 80.23 | 1358.9 |
| Simplified HyperCuts [22] | FPGA | 10000 | 286 | 10.84 (5.12) | 379.0 |
| BV-TCAM [4] | FPGA | 222 | 16 | 10 (N/A) | 138.8 |
| 2sBFCE [7] | FPGA | 4000 | 178 | 2.06 (1.88) | 46.3 |
| Memory-based DCFL [16] | FPGA | 128 | 221 | 24 (16) | 13.9 |
| B2PC [6] | ASIC | 3300 | 540 | 13.6 | 83.1 |