Large-scale Packet Classification on FPGA*

Shijie Zhou, Yun R. Qu, Viktor K. Prasanna
Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, CA 90089, USA
{shijiezh, yunqu, prasanna}@usc.edu

***Abstract*—Packet classification is a key network function enabling a variety of network applications, such as network security, Quality of Service (QoS) routing, and other value-added services. Routers perform packet classification based on a predefined rule set. Packet classification faces two challenges: (1) the data rate of the network traffic keeps increasing, and (2) the size of the rule sets are becoming very large. In this paper, we propose an FPGA-based packet classification engine for large rule sets. We present a decomposition-based approach, where each field of the packet header is searched separately. Then we merge the partial search results from all the fields using a merging network. Experimental results show that our design can achieve a throughput of 147 Million Packets Per Second (MPPS), while supporting upto 256 K rules on a state-of-the-art FPGA. Compared to the prior works on FPGA or multi-core processors, our design demonstrates significant performance improvements.**

I. Introduction

The development of the Internet demands routers to support a variety of network applications, such as firewall processing, Quality of Service (QoS) differentiation, policy routing and other value-added services. In order to provide these services, routers need to classify packets [1] into different categories based on a predefined rule set.

As the Internet continues to evolve, many emerging network applications require high throughput to be sustained. Meanwhile, there has been a trend towards matching a large number of rules consisting of a large number of packet header fields (*e.g.*, *OpenFlow table lookup* [2]). Thus, there are two major challenges for packet classification: (1) the increasing demand for high throughput, and (2) the growing size of the rule sets. The current link rate has been pushed beyond 100 Gbps [3], which corresponds to 312 MPPS for minimum packet size (40 bytes). Using existing software-based solutions to achieve such high performance is impractical [4].

Many hardware-based packet classification engines adopt Ternary Content Addressable Memories (TCAMs) [5]. TCAMs can perform parallel searching against all the rules in $O(1)$ time, while they are expensive, power-hungry, and not scalable with respect to clock rate or circuit area [6]. Field-Programmable Gate Array (FPGA) technology has also been widely used to accelerate many applications [7], [8]. Previous works have shown FPGA-based packet classification engines [9], [10] can achieve very high throughput for rule sets of moderate size. When the rule set gets too large, off-chip memory is exploited, incurring long memory access latency. This significantly deteriorates the overall performance. It is still challenging to support large rule sets on FPGA.

In this paper, we present a decomposition-based approach on FPGA. We search all the fields independently; we merge all the partial search results using a merging network. The main contributions of our work are:

- We present a decomposition-based approach for packet classification. We prototype our design on a state-of-the-art FPGA, targeting very large rule sets.
- We search all the fields, independently, using balanced range trees. We merge all the partial search results using a merging network efficiently.
- We achieve 147 MPPS throughput for rule sets consisting of upto 256 K rules. The post-place-and-route results on FPGA demonstrate significant performance improvements, compared to the existing FPGA-based implementations or the prior works on multi-core processors.

The rest of the paper is organized as follows. Section II introduces the background and related works. Section III describes our algorithms; Section IV details the architecture of our design on FPGA. We show the experimental results in Section V. We conclude the paper in Section VI.

II. Background and Related work

A. Multi-field Packet Classification

The classic multi-field packet classification [1] requires packets to be classified based on 5 header fields: source/destination IP addresses (SIP/DIP), source/destination port numbers (SP/DP), and transport layer protocol (Protocol). A *rule set* consists of multiple rules; a rule is a set of matching criteria on all the 5 fields. We denote the total number of rules as N.

Different fields require different types of matches. For example, SIP and DIP require prefix match, SP and DP require range match, and protocol field requires exact match. We denote the total number of fields as M ($M = 5$ for multi-field packet classification). A packet matches a rule only if all the M header fields are matched. If any rule is matched, the associated action is taken. In this paper, we target the *multi-match* problem: when multiple rules are matched, we report all the matches. We show an example of a rule set in Table I.

As a newer version of multi-field packet classification, OpenFlow table lookup [2] involves matching the incoming packet headers against up to 15 fields (*e.g.*, metadata, ingress

* This work is supported by U.S. National Science Foundation under grants CCF-1320211 and ACI-1339756. Equipment grant from Xilinx Inc. is gratefully acknowledged.

TABLE I: An example of a packet classification rule set, $N = 6$, $M = 5$

Rule ID	(32-bit) SIP	(32-bit) DIP	(16-bit) SP	(16-bit) DP	(8-bit) Protocol	Action
0	175.77.88.1/32	192.0.0.0/8	[0, 120)	[0, 500)	0x06	Forward to port 1
1	175.77.88.0/24	192.0.96.12/32	[16, 30)	[0, 655)	0x06	Forward to port 0
2	10.10.10.0/24	125.199.2.72/32	[16, 30)	[200, 300)	0x11	Forward to port 2
3	10.10.10.0/24	12.13.0.0/16	[100, 120)	[0, 655)	0x2F	Broadcast
4	12.1.11.256/32	137.135.0.0/16	[0, 200)	[20, 22)	0x3C	Forward to port 0
5	12.1.11.255/32	12.13.0.0/16	[30, 220)	[20, 22)	0x06	Discard

port, *etc.*). This is much more challenging compared to the classic 5-field packet classification. Also, there has been a trend of matching very large rule sets [1], [11]; this makes packet classification even more challenging.

B. Packet Classification Algorithms

Packet classification algorithms fall into two major categories [6]: decision-tree-based algorithms and decomposition-based algorithms.

The most well-known decision-tree based algorithms are HiCuts [1] and HyperCuts [12]. Such algorithms treat each rule as a hypercube in a multi-dimensional space; each packet is viewed as a point in this space. To construct a decision-tree, several heuristics can be exploited to recursively cut the multi-dimensional space into smaller subspaces. A smaller subspace involves only a small number of rules; the final classification result can be obtained by a linear search in this subspace. The memory consumption of a decision-tree can be $O(N^M)$; this is extremely expensive, especially for OpenFlow table lookup ($M = 15$). Meanwhile, a decision-tree can be very imbalanced, therefore not suitable for hardware implementation.

Decomposition-based algorithms include two phases: searching and merging [6], [11]. In the searching phase, independent searches are performed on each field to obtain the partial search result for each field. A partial search result can be represented either as a Bit Vector (BV) or a Rule ID Set (RIDS). In the merge phase, all intermediate results are combined to produce final result. A BV-based approach requires $O(MN)$ memory and $O(MN)$ merging time. A RIDS-based approach requires $O(MN \log N)$ memory and $O(N)$ merging time. For these merging techniques, it is not easy to overlap the merging time for multiple packets. Hence these algorithms, when deployed on hardware accelerators, usually result in inferior performance.

C. Prior Works on FPGA

A decision-tree can be mapped onto a pipelined architecture on FPGA [4]. The implementation can achieve 80 Gbps (250 MPPS) for a $10\,K$ rule set without using off-chip memory. However, since the memory consumption grows at a rate of $O(N^M)$, the design cannot support larger rule sets.

A class of BV-based approaches have been proposed on FPGA [13], [14]. The FSBV approach [13] examines the header fields bit-by-bit against the rule set and generates a BV (as a partial result) for each bit. In a BV consisting of N bits, a bit is set to "1" only if the packet header matches the corresponding rule. The final classification result is obtained by bitwise-ANDing all the BVs. An enhanced version, StrideBV [14], can examine a stride of several bits instead of a single bit at a time. Since FSBV and StrideBV require $O(MN)$ memory, they cannot support very large rule set without using slow off-chip memory.

BV-TCAM [9] combines TCAMs and the BV algorithm for packet classification. A TCAM is used to perform prefix and exact match, while a multi-bit trie is used to perform source and destination port lookup. Although the entire design on an FPGA device consumed less than 10% of the available logic and fewer than 20% of the available Block RAMs, this approach cannot be easily scaled for very large rule sets since TCAMs are very expensive.

As a decomposition-based approach, hashing can be used to merge all the partial results [15], [16]. After the searching phase, the partial results are used as hash keys to access a huge hash table stored off-chip. Bloom filters [15] allow false positive, where a linear search has to be performed on all the returned rule IDs. Alternatively, the construction of perfect hash functions [16] requires a memory proportional to the total number[1] of pseudo rules [15], [16]. Since the performance of these approaches highly relies on the off-chip memory access speed, it is not easy to scale up the resulting designs.

III. Algorithms

In this section, we first introduce the data structures used in our algorithm in Section III-A. Then we present the motivation of our novel merging technique in Section III-B. We detail our efficient merging network in Section III-C.

A. Range-Tree and Rule ID Set

We exploit range-tree [17] to search each packet header field. For a field requiring range match, the major steps are (1) to flatten all the ranges into *non-overlapping subranges*, and (2) to construct a balanced binary search tree using the subrange boundaries. For instance, Figure 1 shows an example of the range-tree constructed from the SP field of Table I. The non-overlapping subranges are $[0, 16)$, $[16, 30)$, $[30, 100)$, $[100, 120)$, $[120, 200)$, $[200, 220)$; the balanced binary search tree is constructed of the 7 subrange boundaries. We ignore the

[1]This is equivalent to the memory required for direct addressing.

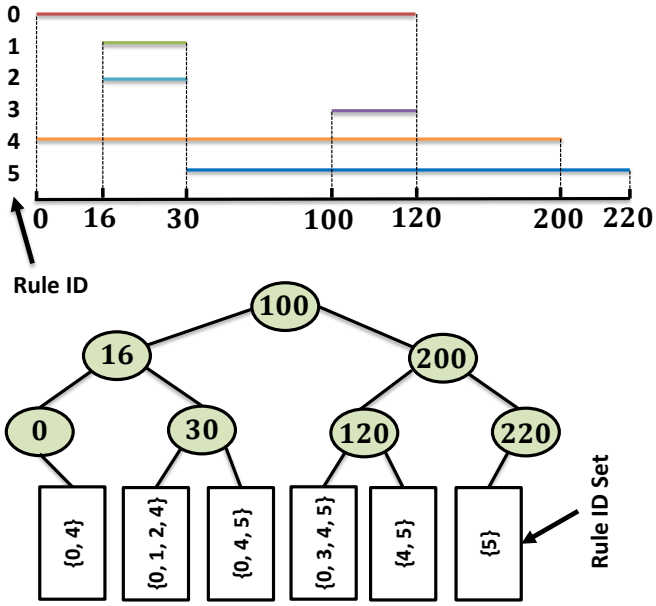

Fig. 1: Construct a range-tree for the SP field

details of this construction since it has been widely studied. Note any prefix or exact value can be translated into a range; hence similar steps can be applied to fields requiring prefix match or exact match.

As discussed in Section II, it is very expensive to store partial search results in BVs. In our approach, each leaf node of range-tree stores a Rule ID Set (RIDS); a RIDS only stores the rule IDs matching the corresponding subrange. For example in Figure 1, the RIDS corresponding to the subrange $[0, 16)$ stores the rule IDs 0 and 4, since any input value falling into this subrange will match Rule 0 and Rule 4 (as can be seen in Table I). Note the RIDSs are prepared when we construct the range-trees. To simplify the merging phase (Section III-C), we enforce the following properties of a RIDS:

1) (*Distinctness*) The rule IDs are all distinct.
2) (*Monotonic*) The rule IDs are in ascending order.

During the searching phase, we search all the M range-trees until we reach the leaf nodes; the RIDS corresponding to a reached leaf node is denoted as *candidate RIDS*. A total number of M candidate RIDSs are extracted as the partial results during the searching phase.

B. Motivation

During the merging phase, M RIDSs have to be combined into the final result. For example, suppose $M = 2$, and we get two RIDSs $\{0, 1\}$ and $\{1, 2, 3\}$; the final result should be $\{1\}$, indicating the packet header matches the rule with ID "1". Note the number of occurrences of "1" in these 2 sets is exactly $M = 2$. In general, *given M RIDSs, we aim to find all the rule IDs with a number of M occurrences.*

The key ideas of our approach is to build a merging network so that the common rule IDs from all the candidate RIDSs can be collected in a streaming fashion. This allows us to pipeline

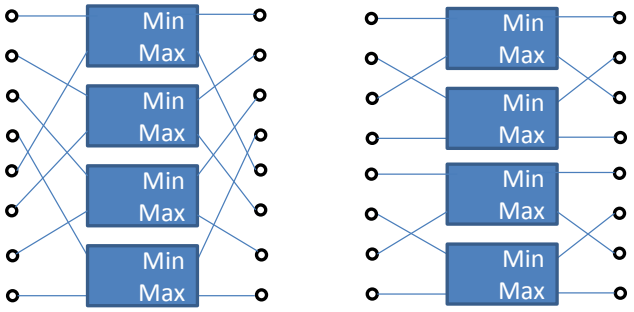

Fig. 2: Stage 0 (left) and Stage 1 (right) of BM(8)

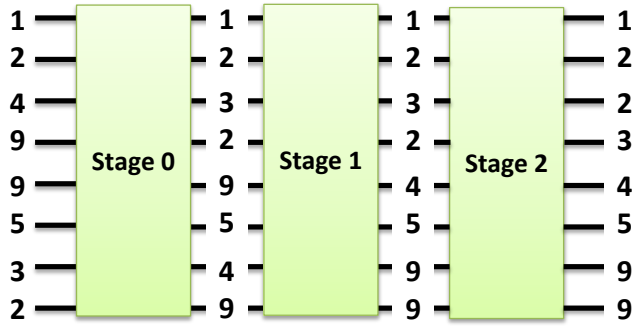

Fig. 3: An example of merging two monotonic sequences $\{1, 2, 4, 9\}$ and $\{9, 5, 3, 2\}$ using BM(8)

the merging phase and improve the overall throughput. To do this, we (1) use a bitonic merging network [18] to merge any 2 RIDSs, and (2) merge all the M RIDSs iteratively in pairs. The rule IDs with M occurrences can be collected easily from a sorted sequence. Note we have arranged the rule IDs in a RIDS in ascending order. A descending order of the rule IDs can be obtained trivially by reversing the order of all the IDs.

C. Merging Network

1) Bitonic Merge: A sequence $\{x_0, x_1, \ldots, x_{k-1}\}$ is *bitonic* if for some cycle shift of $\{x_0, x_1, \ldots, x_{k-1}\}$, the resulting sequence can be split into two subsequences, where the first one is ascending while the second one is descending. Note any of the two subsequences can be empty. For example, "$\{1, 3, 2, 0\}$", "$\{3, 2, 0, 1\}$" (cyclic shifted), and "$\{1, 4, 5\}$" (monotonic) are all considered as bitonic sequences.

A bitonic merge network takes a bitonic sequence of k numbers as input, and produces a monotonic (sorted) sequence. We denote such a bitonic merge network as BM(k), where k denotes the size of this network. A BM(k) has $\log(k)$ stages, indexed by $i = 0, 1, \ldots, \log(k) - 1$. Stage i consists of $\frac{k}{2}$ comparators, while the inputs/outputs are connected to the comparators via 2^i perfect shuffle networks. We show two stages as an example in Figure 2. We show an example of using BM(8) to merge two sorted sequences in Figure 3.

2) Neighborhood Checker: Since the two sequences to be merged each have distinct numbers, the maximum number of

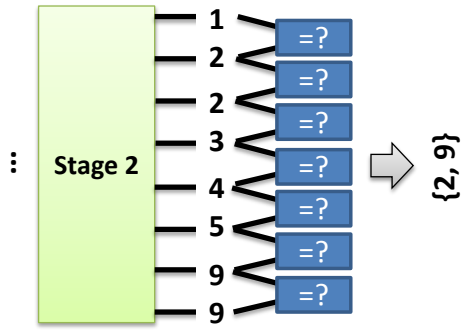

Fig. 4: Neighborhood checker reporting the common numbers

occurrences for any number is 2. In Figure 3, if a number appears twice (*e.g.*, "2" or "9"), this number can be identified by exploring the neighborhood of all the numbers.

Continuing the example of Figure 3, we show the Neighborhood Checker (NC) in Figure 4. An NC for k numbers consists of $(k-1)$ 2-input comparators, each checking whether the 2 input numbers are equal or not. The NC finally reports the common numbers appearing twice.

3) Bitonic-tree: A bitonic merging network along with an NC is only capable of collecting common numbers from 2 sorted sequences efficiently. For a total number of M RIDSs, we exploit a tree-like merging network, and denote this merging network as *bitonic-tree*. The intuition is to collect all the common rule IDs from M RIDSs iteratively. Each node in a bitonic-tree consists of a BM(k) and an NC; we denote such a node as BMNC(k). We will present an example later in Section IV.

IV. Overall Architecture

We show the overall architecture of our design on FPGA in Figure 5. In this example, we show a case where $M = 4$ fields are examined; the bitonic-tree only has 2 levels. In general, the number of levels required in a bitonic-tree is $\lceil \log(M) \rceil$.

Also, in Figure 5, the candidate RIDS in field 0 stores 8 rule IDs; the candidate RIDS in field 1 stores 6 rule IDs. A BMNC(16) is sufficient to collect the common rule IDs from all the $(8+6)$ rule IDs. In general, suppose we have s_0 rule IDs

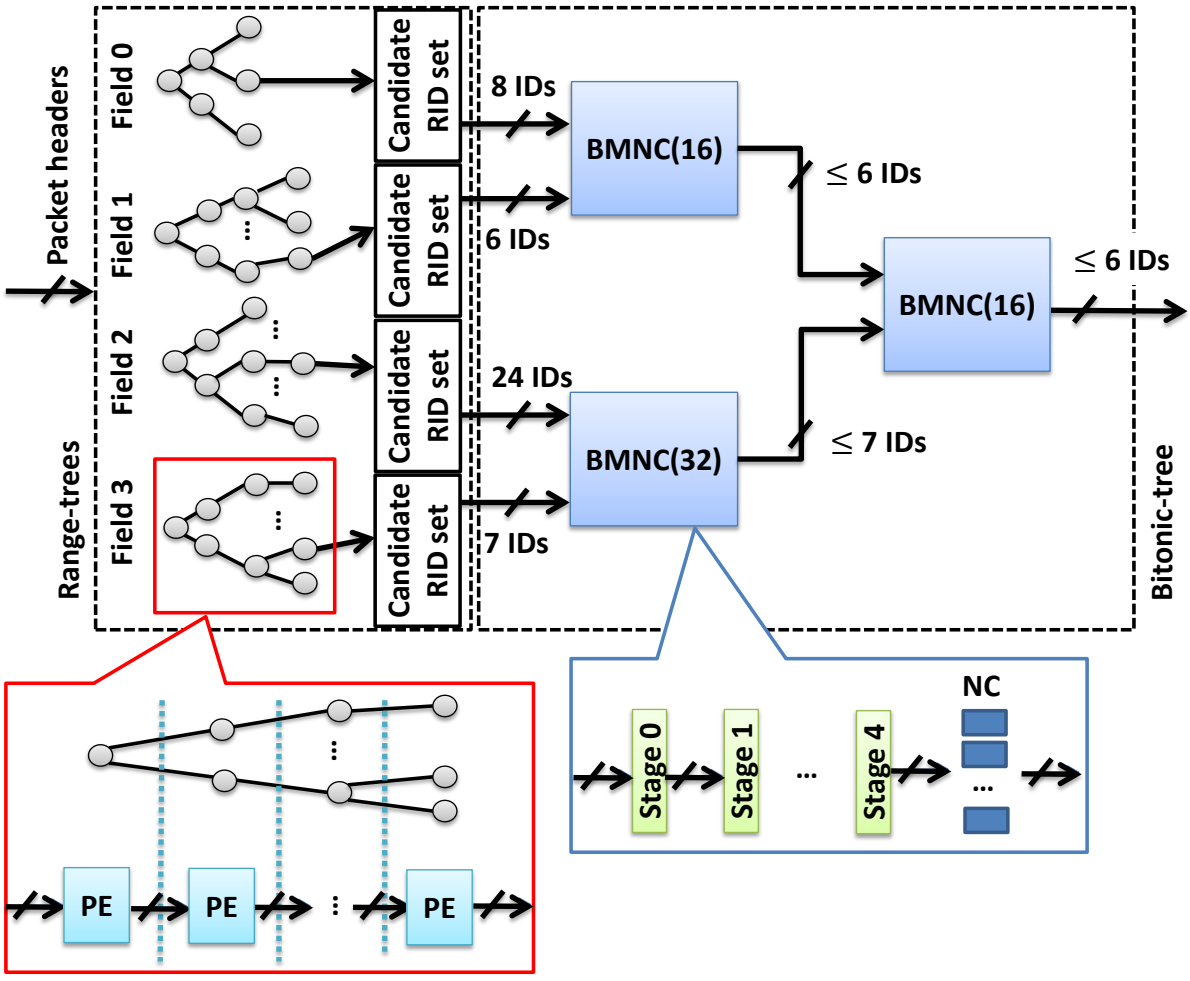

Fig. 5: Example: searching $M = 4$ fields using range-trees, and merging all the RIDSs using a bitonic-tree

to be checked against another s_1 rule IDs, the corresponding BMNC(k) in the bitonic-tree must have $k \geq 2^{\lceil \log(s_0+s_1) \rceil}$.

Theorem 1: The root BMNC of the bitonic-tree reports the common IDs appearing exactly M times in all the RIDSs.

Proof: Suppose the root node reports a common ID x appearing more than M times, according to the pigeonhole principle, there must be a field whose candidate RIDS stores at least two x's. This contradicts the distinctness property of the RIDS (Section III-A).

Besides, since all the nodes (BMNC) in the bitonic-tree only collect common rule IDs, a common ID x reported by the root node must have appeared in all the M fields. Therefore, such a rule ID must have appeared in every single candidate RIDS, in total exactly M times. ■

As shown in Figure 5, to boost the throughput of our classification engine, the entire architecture is deeply pipelined.

A. Mapping Range-tree onto FPGA

To map the range-tree of field m ($m = 0, 1, \ldots, M-1$) onto a pipelined architecture, we construct a group of Processing Elements (PEs). Depending on the lookup result from a previous PE, a PE extracts a subrange boundary from the memory, and matches the m-th field of the input packet header against this subrange boundary. The PEs are concatenated into a linear pipeline, where the last PE stores all the candidate RIDSs in this field.

In a particular field m, the number of PEs depends on the depth of the corresponding range-tree, which in turn depends on the number of non-overlapping subranges in this field. In this paper, we denote the number of non-overlapping subranges in field m as u_m, $m = 0, 1, \ldots, M-1$.

Before the merging phase, note our approach requires all the candidate RIDSs to be ready before any two of the candidate RIDSs can be merged. For those range-trees that are shallow, additional delay stages have to be added. As a result, the number of pipeline stages in all the fields is designed uniformly to be $\lceil \log(\max_m[u_m]) \rceil$. Excluding the memory required for storing all the RIDSs, the pipelines mapped from all the range-trees consume a total memory of size $O(\sum u_m)$.

As a design choice for the pipelines, we choose to use both (LUT-based) distributed RAM and Block RAM (BRAM) available on FPGA to store all the subrange boundaries. This helps us implement very large range-trees by exploiting all the available memory resources on FPGA.

B. Mapping Bitonic-tree onto FPGA

As discussed in Section III-C, each node of the bitonic-tree is a BMNC. For each node:

1) Every single stage in the bitonic merging network is mapped onto a pipeline stage.
2) The NC is mapped onto a pipeline stage.

In a particular field m ($m = 0, 1, \ldots, M-1$), we denote the maximum number of rule IDs in a RIDS as s_m. The number of common rule IDs reported by the BMNC at the root of the bitonic-tree is at most $\min_m[s_m]$. However, in order to reserve sufficient memory for all the rule IDs, in each BMNC node of our bitonic-tree, we allocate a memory of size $O(\max_m[s_m] \cdot \log(N))$. Thus, the total memory required by the bitonic-tree is $O(\max_m[s_m] \cdot M \log(N))$. To handle very large rule sets, the memory required by the bitonic-tree are instantiated using BRAM.

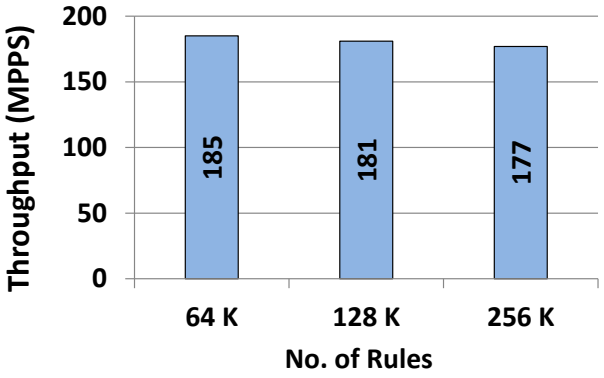

Fig. 6: Throughput with respect to N

V. Performance Evaluation

A. Experimental Setup

We conducted experiments on the state-of-the-art Xilinx Virtex 7 FPGA (XC7VX690T, speed grade -2L). The target platform has 433200 logic slices, 850 I/O pins and 51.6 Mb on-chip BRAMs. The performance was evaluated using Xilinx Vivado 2014.3 development tools. We used the following performance metrics:

- *Throughput*: the number of packets classified per unit time (in MPPS).
- *Resource utilization*: the utilization of basic FPGA resources including slice LUTs, slice registers and BRAMs.

Due to the lack of large real-life rule sets, especially for packet classification involving more than 5 fields [10], we built synthetic rule sets where u_m and s_m ($m = 0, 1, \ldots, M-1$) could be adjusted separately. We used constrained random packet headers as inputs to our packet classification engine; the constraint was to enforce that at least one rule would match the input packet header[2].

In this section, we first evaluated the performance using various combinations of parameters in Section V-B, Section V-C, Section V-D, and Section V-E. We compare our performance results with prior works on FPGA and multi-core processors in Section V-F and Section V-G, respectively.

B. Varying Number of Rules

We first study the impact of N on the performance by fixing $M = 4$, $\max_m[u_m] = 8$ K, and $\max_m[s_m] = 64$. To demonstrate the scalability of our design, we vary N from 64 K to 256 K (the largest to the best of our knowledge). We have observed similar performance results for other combinations

[2]Reasonable because many rule sets contain "wildcard" rules; *i.e.*, a default action will be performed if none of the other rules matches the packet header.

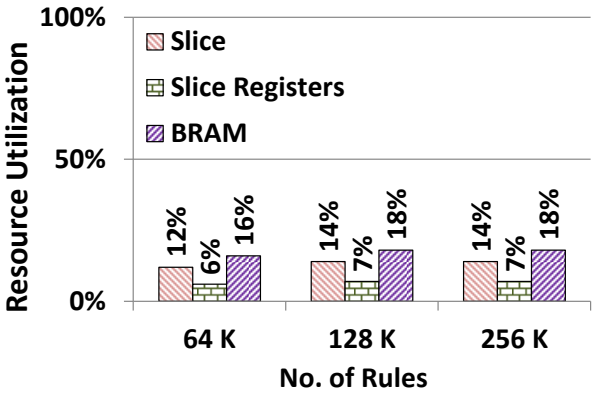

Fig. 7: Resource utilization with respect to N

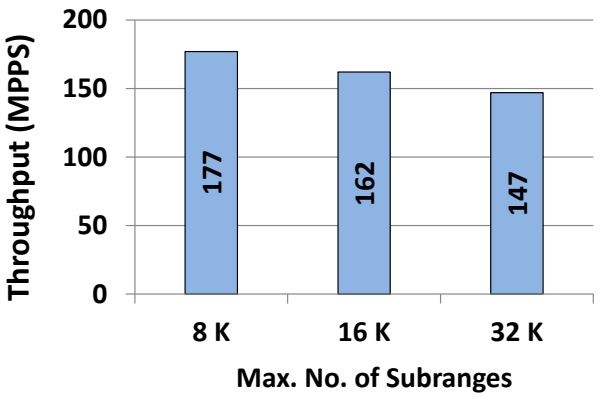

Fig. 8: Throughput with respect to $\max_m[u_m]$

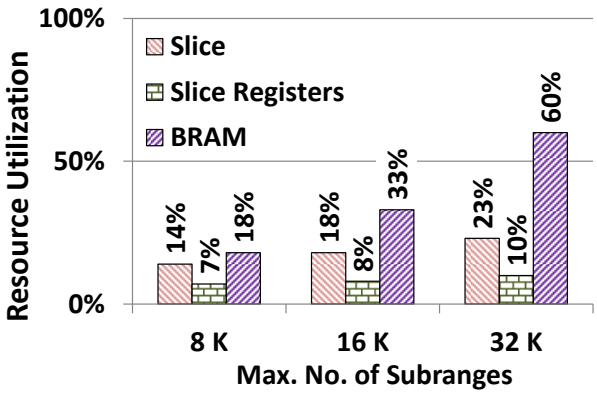

Fig. 9: Resource utilization with respect to $\max_m[u_m]$

of these parameters. Figure 6 and Figure 7 show the throughput and resource utilization with respect to N, respectively.

As can be seen, our architecture maintains a high throughput of over 177 MPPS for various values of N. Since the values of u_m are kept fixed, the depths of the range-trees remain the same. Hence even when N increases, there is only little increase with respect to the resource utilization. Remember the total memory consumption of the bitonic-tree is $O\left(\max_m[s_m] \cdot M \log(N)\right)$; hence a larger rule set usually consumes more memory resources.

C. Varying Number of Subranges

Now we explore the impact of u_m on the performance. We fix $N = 256\,\text{K}$, $M = 4$, and $\max_m[s_m] = 64$. We vary $\max_m[u_m]$ from $8\,\text{K}$ to $32\,\text{K}$ in our synthetic rule sets. We have observed similar performance results for other combinations of these parameters. We show throughput and resource utilization with respect to various values of $\max_m[u_m]$ in Figure 8 and Figure 9, respectively.

We observe that as $\max_m[u_m]$ increases, the performance deteriorates. This is mainly because the depths of the range-trees have been increased. As the range-trees grow deeper, more pipeline stages ($\lceil \log\left(\max_m[u_m]\right) \rceil$ stages) are deployed. The wire lengths of the critical paths also grow linearly with respect to $\max_m[u_m]$. This degrades the clock rate and the throughput of our packet classification engine.

Also, for deeper trees, more resources are consumed. Note that the BRAM utilization increases linearly with respect to $\max_m[u_m]$. This reflects the fact that the number of leaf nodes in a range-tree can require a memory as large as $O(\max_m[u_m])$.

D. Varying Number of Rule IDs in RIDS

We explore the impact of s_m in each set. We fix $N = 256\,\text{K}$, $M = 4$ and $\max_m[u_m] = 8\,\text{K}$; we vary $\max_m[s_m]$ from 64 to 128. We have seen similar performance results for other combinations of these parameters. Figure 10 and Figure 11 show the throughput and resource utilization with respect to various values of $\max_m[s_m]$, respectively. As can be seen:

- The throughput tapers as $\max_m[s_m]$ increases. For larger values of $\max_m[s_m]$, more BRAMs are employed to store all the RIDSs; longer wires are required to connect the BRAMs, leading to a degradation of the clock rate achieved on FPGA.
- The resource utilization increases linearly with respect to $\max_m[s_m]$. This matches our analysis that the total memory consumption of the bitonic-tree is $O\left(\max_m[s_m] \cdot M \log(N)\right)$.

E. Varying Number of Fields

In this subsection, we vary M from 4 to 16 for packet classification with various numbers of fields. We fix $N = 256\,\text{K}$, $\max_m[u_m] = 8\,\text{K}$, and $\max_m[s_m] = 64$. We have seen similar performance results for other combinations of these parameters. We show throughput and resource utilization with respect to various values of M in Figure 12 and Figure 13, respectively. As can be seen:

- The throughput tapers as M increases. When we scale up our design on FGPA by increasing M, the routing gets more complex; also, since our architecture is synchronized to a global clock, the increasing clock skew degrades the achievable clock rate.
- The resource consumption increases (slightly faster than) linearly with respect to M. Note the resource consumed by the range-trees increases linearly with respect to M,

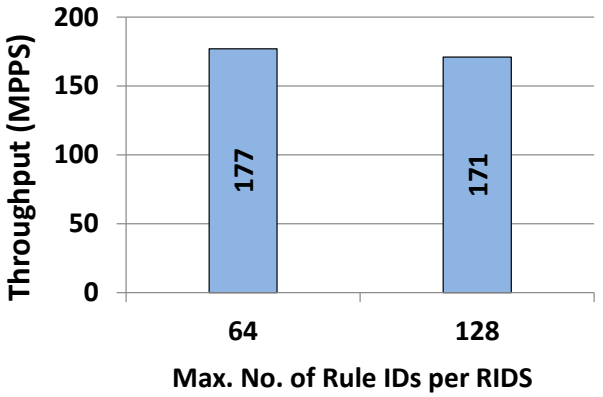

Fig. 10: Throughput with respect to $\max_m[s_m]$

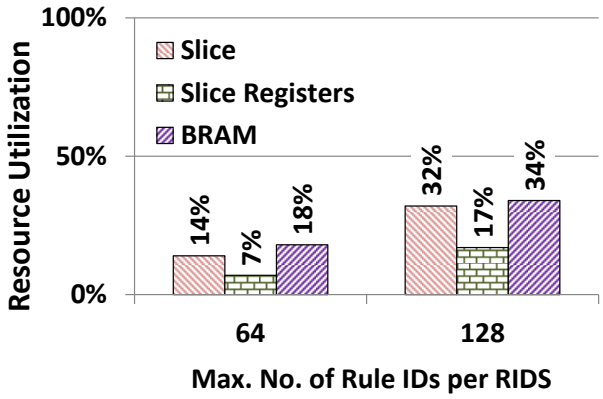

Fig. 11: Resource utilization with respect to $\max_m[s_m]$

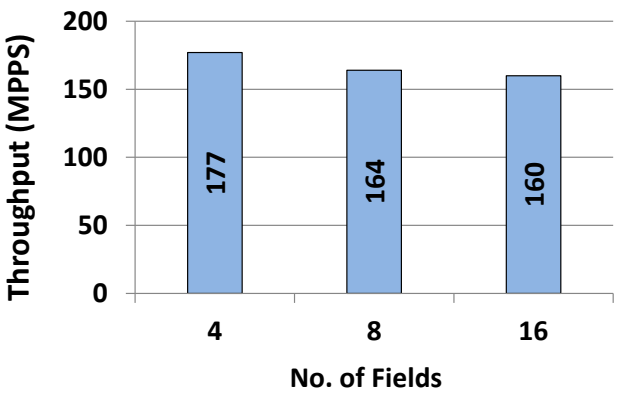

Fig. 12: Throughput with respect to M

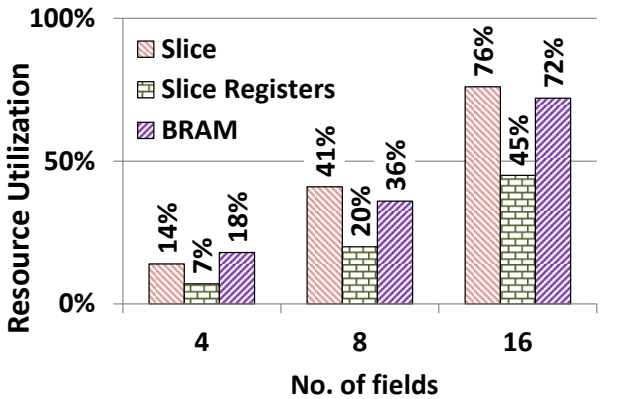

Fig. 13: Resource utilization with respect to M

while the resource consumed by the bitonic-tree increases superlinearly with respect to M.

We compare the performance results for the 5-field packet classification and 15-field OpenFlow Table Lookup [2]. Here, for simplicity, we set again $N = 256\,\text{K}$, $\max_m[u_m] = 8\,\text{K}$ and $\max_m[s_m] = 64$, although similar performance results can be seen for other combinations of these parameters as well. Table II shows the throughput and resource utilization of our classification engines. The performance results agree with our intuition and analysis.

F. Comparison with Existing FPGA-based Approaches

We compare our design with existing FPGA-based approaches in Table III. For fair comparisons, the results of prior works (*e.g.*, on Xilinx Virtex-5 FPGA, *etc.*) are all scaled up to the state-of-the-art Xilinx Virtex-7 platform; this is done by considering the maximum clock frequency supported on these platforms. Besides, considering the maximum memory capacity on these platforms, we also scale up the rule set size for these approaches ($\sim 100\%$ resource utilization). For all the prior works, the estimation on the throughput or the rule set size is quite optimistic, considering many real design constraints such as the limited number of I/O pins on FPGA.

Considering the tradeoff between the throughput and the rule set size, in this subsection, we explore the product of the throughput and the number of rules supported by a specific design as a new performance metric. Note the following two design styles meet the same resource requirement:

1) By replicating a design α times, we can support $\alpha\times$ throughput for the same rule set.
2) Using the same design α times, each supporting a different rule set, we can support in total a rule set upto $\alpha\times$ larger.

In Table III, we use the product of the throughput and the number of rules as a compound performance metric; as can be seen, our design on FPGA demonstrates at least $2.8\times$ improvement compared with prior works. Note that this performance improvement can be sustained even if we assume the maximum number of non-overlapping subranges equal to the number of rules; *i.e.*, $\max_m u_m = N$. This is a pessimistic assumption because many fields in reality only contain a very small number of unique values.

G. Comparison with a Multi-core Implementation

In this subsection, we compare the performance of our implementation on FPGA with our multi-core implementation [11]. This decomposition-based implementation was based on range-trees and RIDSs; a state-of-the-art AMD Opteron 6278 processor was employed in this implementation. The target platform had 16 cores, each running at $2.4\,\text{GHz}$. Each core

TABLE III: Performance comparison

	Approach	No. of Rules ($\times$1 K)	Throughput (MPPS)	Throughput $\times$ No. of Rules (MPPS $\times$ 1 K)
Song's [9]	BV + TCAM	1.2	$\sim$ 20	$\sim$ 24
Kennedy's [19]	Decision-tree	60	$\sim$ 15	$\sim$ 900
Pus's [16]	Decomposition + hashing	0.6	$\sim$ 500	$\sim$ 300
Ganegedara's [14]	pipelined BV	6	$\sim$ 275	$\sim$ 1650
This work	Decomposition + bitonic-tree	32 $\sim$ 256	$\geq$ 147	$\geq$ 4704

TABLE II: 5-field and 15-field Classification

	Throughput	Slice LUT	Slice Reg.	BRAM
5-field	170 MPPS	17%	8%	22%
15-field	161 MPPS	73%	43%	68%

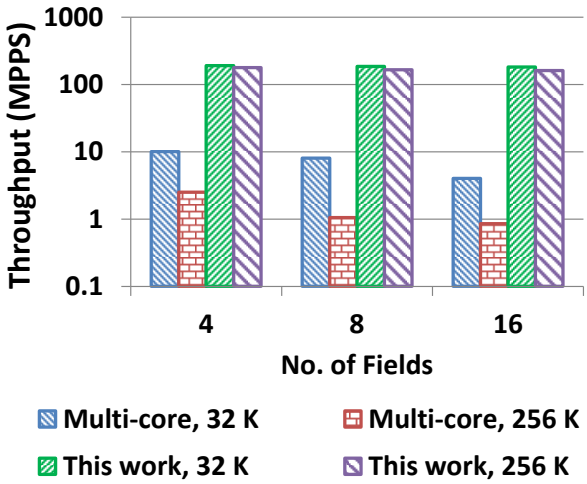

Fig. 14: Comparison with multi-core implementation

had a 16 KB L1 data cache, 64 KB L1 instruction cache and a 2 MB L2 cache. A 60 MB L3 cache is shared among all the 16 cores.

For both implementations, we set $\max_m[u_m] = 8\,\text{K}$ and $\max_m[s_m] = 64$. We vary M from 4 to 16; N is chosen to be 32 K or 256 K as an example. Figure 14 shows the throughput for the two implementations. For the same rule set size, our implementation on FPGA (this work) achieves at least $10\times$ throughput. As the number of fields M or the number of rules N increases, our implementation on FPGA sustains very high throughput while the performance on multi-core processor deteriorates.

VI. Conclusion

In this paper, we presented a decomposition-based approach to classify packets against large rule sets. We deployed range-trees for the searching phase and a merging network for the merging phase. We prototyped our design on a state-of-the-art FPGA. Experimental results showed that our classification engine sustained high throughput for very large rule sets.

Our future work includes supporting dynamic rule updates, and a comprehensive study on the latency and power performance of our classification engines. We plan to investigate more performance tradeoffs among various multi-field packet classification approaches.

References

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

[2] OpenFlow Switch Specification V1.3.1. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf.

[3] Making the move to 100G. http://www.optelian.com/media/153301/Optelian-MoveTo100g-WP.pdf.

[4] Weirong Jiang and Viktor K. Prasanna. Scalable Packet Classification on FPGA. *IEEE Trans. VLSI Syst.*, 20(9):1668–1680, 2012.

[5] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for Advanced Packet Classification with Ternary CAMs. In *Proc. ACM SIGCOMM*, pages 193–204, 2005.

[6] David E. Taylor and Jonathan S. Turner. Scalable Packet Classification using Distributed Crossproducing of Field Labels. In *Proc. IEEE INFOCOM*, pages 269–280, 2005.

[7] P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, and W. Luk. Run-Time Integration of Reconfigurable Video Processing Systems. *IEEE Trans. on VLSI*, 15(9):1003–1016, Sept 2007.

[8] Patrick Cooke, Jeremy Fowers, Greg Brown, and Greg Stitt. A Tradeoff Analysis of FPGAs, GPUs, and Multicores for Sliding-Window Applications. *ACM Trans. Reconfig. Tech. Sys.*, 8(1):2:1–2:24, 2015.

[9] Haoyu Song and John W. Lockwood. Efficient Packet Classification for Network Intrusion Detection using FPGA. In *Proc. ACM/SIGDA FPGA*, pages 238–245, 2005.

[10] Yun R. Qu, Shijie Zhou, and Viktor K. Prasanna. High-performance architecture for dynamically updatable packet classification on FPGA. In *Proc. ACM/IEEE ANCS*, pages 125–136, 2013.

[11] S. Zhou, Y. R. Qu, and V. K. Prasanna. Multi-core Implementation of Decomposition-based Packet Classification Algorithms. In *Proc. PaCT*, pages 105–119, 2013.

[12] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet Classification Using Multidimensional Cutting. In *Proc. ACM SIGCOMM*, pages 213–224, 2003.

[13] Weirong Jiang and Viktor K. Prasanna. Field-split Parallel Architecture for High Performance Multi-match Packet Classification using FPGAs. In *Proc. ACM SPAA*, pages 188–196, 2009.

[14] Thilan Ganegedara and Viktor K. Prasanna. StrideBV: Single Chip 400G+ Packet Classification. In *Proc. IEEE HPSR*, pages 1–6, 2012.

[15] S. Dharmapurikar, Haoyu Song, J. Turner, and J. Lockwood. Fast Packet Classification using Bloom Filters. In *Proc. ACM/IEEE ANCS*, pages 61 –70, 2006.

[16] Viktor Pus, Jan Korenek, and Jan Korenek. Fast and Scalable Packet Classification using Perfect Hash Functions. In *Proc. ACM/SIGDA FPGA*, pages 229–236, 2009.

[17] Pingfeng Zhong. An IPv6 Address Lookup Algorithm based on Recursive Balanced Multi-way Range Trees with Efficient Search and Update. In *Proc. CSSS*, pages 2059–2063, 2011.

[18] K. E. Batcher. Sorting Networks and Their Applications. In *Proc. ACM AFIPS*, pages 307–314, 1968.

[19] Alan Kennedy, Xiaojun Wang, Zhen Liu, and Bin Liu. Low Power Architecture for High Speed Packet Classification. In *Proc. ACM/IEEE ANCS*, pages 131–140, 2008.