Scalable and Dynamically Updatable Lookup Engine for Decision-trees on FPGA*

Yun R. Qu
Ming Hsieh Dept. of Electrical Engineering,
University of Southern California
Los Angeles, CA 90089
yunqu@usc.edu

Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Engineering,
University of Southern California
Los Angeles, CA 90089
prasanna@usc.edu

***Abstract*—Architectures for tree structures on FPGAs as well as ASICs have been proposed over the years. The exponential growth in the memory size with respect to the number of tree levels restricts the scalability of these architectures. In this paper, we propose a scalable lookup engine on FPGA for large decision-trees; this engine sustains high throughput even if the tree is scaled up with respect to (1) the number of fields and (2) the number of leaf nodes. The proposed engine is a 2-dimensional pipelined architecture; this architecture also supports dynamic updates of the decision-tree. Each leaf node of the tree is mapped onto a horizontal pipeline; each field of the tree corresponds to a vertical pipeline. We use dual-port distributed RAM (distRAM) in each individual Processing Element (PE); the resulting architecture for a generic decision-tree accepts two search requests per clock cycle. Post place-and-route results show that, for a typical decision-tree consisting of 512 leaf nodes, with each node storing 320-bit data, our lookup engine can perform 536 Million Lookups Per Second (MLPS). Compared to the state-of-the-art implementation of a binary decision-tree on FPGA, we achieve $2\times$ speed-up; the throughput is sustained even if frequent dynamic updates are performed.**

I. Introduction

Tree structures [1] have been widely used for high-performance computation; a tree can be used as the underlying data structure for search or lookup operations [2]. In particular, binary decision-tree has been widely studied due to its simple structure; elegant algorithms based on balanced BST have been proposed [3] for efficient searching of input data.

Hardware architecture for balanced tree structures have been proposed to enhance the performance. Most hardware-based architectures for tree structures are optimized with respect to search latency [4]–[6] instead of throughput. Thus, these architectures cannot sustain high throughput for large trees. Meanwhile, a new trend in high-performance computing is to support dynamic updates for the underlying data structures [7]; realizing dynamic updates without sacrificing the performance is even more challenging.

State-of-the-art Field Programmable Gate Arrays (FPGAs), with their flexibility and reconfigurability, are especially suitable for implementing a large tree structure [8]. To support large tree structures on FPGA, off-chip memory is usually used. State-of-the-art implementations of large trees on FPGA often utilize a large number of I/O pins for off-chip memory access. The number of pins used is usually proportional to the number of pipeline stages; this is expensive. Also, most of the existing FPGA-based architectures make simple assumptions that off-chip memory access rate can be easily scaled up with respect to the number of used I/O pins [9]. As a result, achieving scalability and high throughput has been a dominant issue in the existing implementations for large tree structures.

In this paper, we present a scalable architecture for decision-tree-based lookup engine on FPGA. This architecture is a 2-dimensional pipelined architecture. It sustains high throughput and supports dynamic updates. Specifically, our contributions include the following:

- We convert a generic decision-tree into a rule table. The conversion technique only depends on the number of fields and the number of leaves in a decision-tree.
- We partition the table and store the table in a distributed manner. We propose an efficient algorithm to update the rule table dynamically.
- We implement our design on a state-of-the-art FPGA device with careful timing constraints. We use an individual PE to locally access each partition of the rule table. We parameterize our design and optimize the performance with respect to various design parameters.
- We achieve 536 MLPS throughput for a decision-tree consisting of 512 leaves, with each node storing 320-bit data. The sustained throughput (with dynamic updates) is $2\times$ the peak throughput (without dynamic updates) of the state-of-the-art implementation.

The rest of the paper is organized as follows: Section II covers the related work. We introduce the relevant data structures and algorithms in Section III. We construct modular PE and present the overall architecture in Section IV. Section V evaluates the performance. Section VI concludes this paper.

II. Background

A. Generic Decision-tree

A generic decision-tree (not necessarily balanced) consisting of N leaf nodes can be described as follows: We first define *field* as a distinct set of data (*e.g.*, Internet packet size,

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

978-1-4799-6233-4/14/$31.00

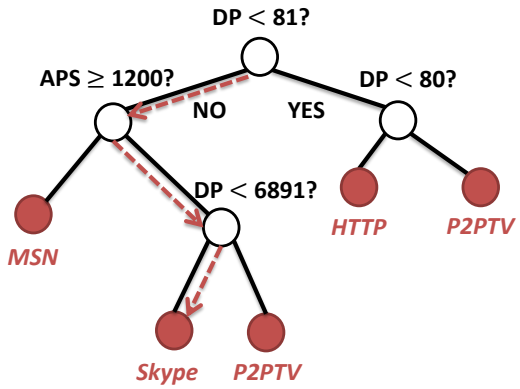


Fig. 1: An example of a decision-tree T

date, time, *etc.*). A decision-tree is then constructed on a set of fields. Each decision-tree node keeps a criterion specified in a particular field. The corresponding field of the input is checked against the criterion of a tree node; the comparison result then determines the next-hop of the lookup process. The lookup process continues until a leaf node is reached, where a final decision can be made.

We show an example of a binary decision-tree for traffic classification [2] in Figure 1. In this example, the decision-tree are built on two fields: 16-bit Destination Port number (DP) field and 12-bit Average Packet Size (APS) field[1]. Note a field can be used more than once along a path from the root to a leaf. We use M to denote the number of fields examined by a decision-tree.

B. Tree Structures on FPGA

A lot of the existing architectures have focused on optimizing the search delay using tree structures on FPGA. For example, [10] employs the carry chains to build compressor trees; its performance is only measured with respect to the delay and area. A comparative study on parallel prefix trees is done in [11], where the main performance differences among 14 architecture configurations are presented. However, the comparison work still uses delay rather than throughput as a performance metric for these parallel trees.

In [6], a pipelined binary tree is implemented for pattern counting on FPGA. Each tree level is mapped into an individual pipeline stage; each stage consists of a logic component and a memory module. Inside a pipeline stage, a memory module larger than 512 entries is further divided into multiple sub-modules to reduce the wire length and enhance the throughput. A pipeline architecture based on dynamic search tree is proposed in [7]; it offers high throughput for lookup, insert and delete operations. These architectures employ a straightforward mapping from a tree to a pipeline on FPGA: Each tree level is mapped into a pipeline stage. We denote this straightforward mapping as the state-of-the-art implementation of a decision-tree in this paper (See Section V).

[1]Without loss of generality, we assume that if the corresponding field of the input is less than the value stored in a node, the left child of that node is to be searched; otherwise the right child is to be searched.

TABLE I: Rule table $R_{N,W}$ ($N = 5$, $M = 2$, $W = 28$)

16-bit Dst. Port (DP)	12-bit Avg. Pkt. Size (APS)	Class
[81, 65536)	[0, 1200)	MSN
[6891, 65536)	[1200, 4096)	Skype
[81, 6891)	[1200, 4096)	P2PTV
[80, 81)	[0, 4096)	HTTP
[0, 80)	[0, 4096)	P2PTV

Denoting Ω as the total number of nodes in a tree, the state-of-the-art implementation of a decision-tree is expensive: (1) An unbalanced tree may require $O(\Omega)$ pipeline stages, while a lookup may take $O(\Omega)$ time. (2) For balanced trees, the lookup time is bounded by $O(\log(\Omega))$; however, the memory consumption and the wire length in each stage increases linearly with respect to Ω. For large balanced decision-trees (large Ω), the long wires deteriorate the clock rate. (3) It is not easy to perform dynamic updates, including deleting, modifying, and inserting a decision-tree node. The data stored in decision-tree nodes are correlated; an update may result in too many changes to be performed globally in the tree structure. For example, in Figure 1, deleting the node "$DP < 81$" requires reorganizing the entire tree. This usually leads to long processing latency and complex update mechanism [7].

C. Problem Definition

We define the problem as: *Given an arbitrary decision-tree consisting of N leaf nodes (indexed from left to right as $k = 0, 1, \ldots, N-1$) and M fields (with total width of W bits), design a high-throughput lookup engine on FPGA which also supports dynamic updates of the tree.* We define dynamic updates as the following operations performed during run-time: (1) *Modification*: to change the values stored in a node. (2) *Deletion*: to remove the criterion stored in a node. (3) *Insertion*: to add a node between any pair of a parent node and a child node, or add a child to a leaf node.

III. Data Structures & Algorithms

Our work is based on the following observation: (1) Each root-to-leaf path can be fully characterized by only a "rule" consisting of a few number of comparisons. The rules can be mapped onto FPGA more efficiently than the state-of-the-art implementation. (2) By modifying the data stored locally, we refrain from migrating large amounts of data; instead, dynamic updates are performed in a distributed manner.

We introduce our novel data structure to represent any given decision-tree in Section III-A. Then we present an efficient algorithm to search this data structure in Section III-B. Later we will propose an efficient mapping from this algorithm to the hardware architecture in Section IV.

A. Rule Table

Let us revisit the decision-tree in Figure 1. Suppose the input field values are x_{DP} in the DP field, and x_{APS} in the APS field, respectively. Let W denote the total width

(in bits) of all the fields. If we want to reach the leaf node "Skype" starting from the root, the following criteria have to be satisfied: $6891 \leq x_{DP} < 65536$, and $1200 \leq x_{APS} < 4096$. We denote such a set of criteria as a *rule*; each rule defines M criteria on M fields, respectively.

We denote the decision-tree as T; note each leaf node of T corresponds to a rule. We store in a table all the rules generated for T; each rule is stored as a row in this table. We denote such a table as a *rule table*. Suppose a rule table consists of N rows, and the total width of all the fields is W bits; we use $R_{N,W}$ to denote this rule table. In Table I, we show the rule table $R_{5,28}$ generated from the decision-tree in Figure 1.

Note the conversion from a decision-tree to a rule table does not depend on the size, depth, or degree of the decision-tree. The conversion technique can be easily extended for any arbitrary decision-tree.

B. Splitting and Partitioning

After we have converted a given decision-tree into a rule table, we focus on how to search the inputs in this table. Intuitively, we can search the rule table field-by-field in parallel, and finally merge all the partial results from all the fields. For example, in Table I, we can split the table into two fields (DP and APS); each field can be searched independently.

However, it is not easy to achieve high throughput by only splitting a rule table into multiple fields. This is because we still have to use a tree-based structure to do range search in each field [12]. The performance may still degrade due to the reasons discussed in Section II.

1) *Splitting*: We split a field into multiple W_m-bit sub-fields. For example in Table I, the 12-bit APS field can be split into 3 sub-fields, each of $W_m = 4$ bits. If we split all the fields, we have in total $\frac{W}{W_m}$ sub-fields. We use *sub-rule* to denote the projection of a rule onto a sub-field; hence each sub-rule has W_m bits.
2) *Partitioning*: We further partition the rule table consisting of N rules into multiple *sub-tables* (denoted as $S_{i,j}$, $i = 0, 1, ..., \frac{N}{N_n} - 1$, $j = 0, 1, ..., \frac{W}{W_m} - 1$), each sub-table consisting of N_n sub-rules. For example in Table I, after we split the DP and APS fields into a total number of 7 sub-fields, the rule table can be partitioned into 5×7 sub-tables, each sub-table corresponding to 1 sub-rule.

The splitting and partitioning techniques allow us to utilize massive parallelism on state-of-the-art hardware. The parameters W_m and N_n are determined in Section V.

C. Dynamic Updates

For dynamic decision-trees, the number of leaf nodes N varies during run-time; hence we use N_{max} to denote the maximum value of N during run-time. Upon initialization, we convert a decision-tree of N leaf nodes into a rule table $R_{N,W}$. We add a number of $(N_{max} - N)$ *invalid rules* as placeholders to $R_{N,W}$; these rules do not produce any valid match result. The resulting rule table is $R_{N_{max},W}$. For example, we can add $x_{DP} < 0$ and $x_{APS} < 0$ as an invalid rule into the rule table of Table I; the resulting rule table $R_{6,28}$ has 6 rows ($N_{max} = 6$).

Algorithm 1 Modifying/deleting a node

Input Sub-tables $S_{i,j}$,
$i = 0, 1, ..., \frac{N_{max}}{N_n} - 1$, $j = 0, 1, ..., \frac{W}{W_m} - 1$.
Decision-tree T and the node z to be modified into z'.
Output Updated sub-tables $S'_{i,j}$,
$i = 0, 1, ..., \frac{N_{max}}{N_n} - 1$, $j = 0, 1, ..., \frac{W}{W_m} - 1$.

1: **for** $j = 0$ to $(\frac{W}{W_m} - 1)$ **do**
2: **if** z does not check sub-field j **then**
3: break {ignore j}
4: **end if**
5: **for** $i = 0$ to $(\frac{N_{max}}{N_n} - 1)$ **do**
6: **for** $ii = 0$ to $(N_n - 1)$ **do**
7: **if** leaf node $(i * N_n + ii)$ is a descendant of z **then**
8: Construct a new sub-rule based on z'
9: Replace the ii-th sub-rule by the new sub-rule {in sub-table $S_{i,j}$}
10: **end if**
11: **end for**
12: **end for**
13: **end for**

Algorithm 2 Inserting a node

Input Sub-tables $S_{i,j}$,
$i = 0, 1, ..., \frac{N_{max}}{N_n} - 1$, $j = 0, 1, ..., \frac{W}{W_m} - 1$.
Decision-tree T, the node z' to be inserted, and the node $\widehat{z}$ that should be the parent of z' after insertion.
Output Updated sub-tables $S'_{i,j}$,
$i = 0, 1, ..., \frac{N_{max}}{N_n} - 1$, $j = 0, 1, ..., \frac{W}{W_m} - 1$.

1: **if** z' is not a leaf node **then**
2: Use Algorithm 1 to modify all of the leaf nodes that are descendants of z'
3: **else**
4: Locate an invalid rule as a leaf node z
5: Use Algorithm 1 to modify z into z'
6: **end if**

Dynamic updates of the decision-tree can be viewed as: modify rule(s), delete rule(s), or insert rule(s) on $R_{N_{max},W}$. Note the splitting and partitioning techniques for our rule-table lead to multiple small sub-tables $S_{i,j}$; these sub-tables can be stored on hardware in a fully distributed manner. Therefore, update operations can be performed distributedly.

To modify a tree node z, we show a distributed algorithm in Algorithm 1. The memory write access required to overwrite a data can be performed in different sub-tables independently. Note sometimes it is necessary to flip the polarity of the comparator in Step 9, especially when the comparison type is to be changed from "$<$" to "$\geq$", or vice versa.

Deleting a node z can be viewed as to remove the constraint put by z in the tree. This can be done by modifying the range upperbound (lowerbound) specified by z to the maximum (minimum) value in this subfield; hence we also use Algorithm 1 to delete a node. For example, in Figure 1, to delete the node "$APS \geq 1200$", we modify this node to be

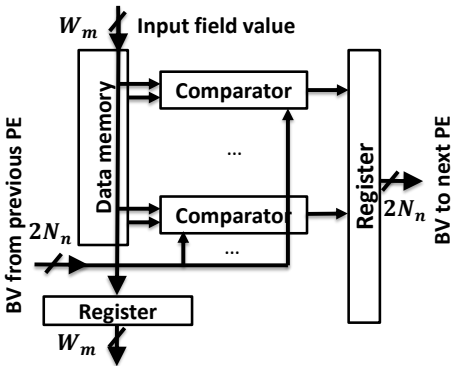


Fig. 2: Modular PE

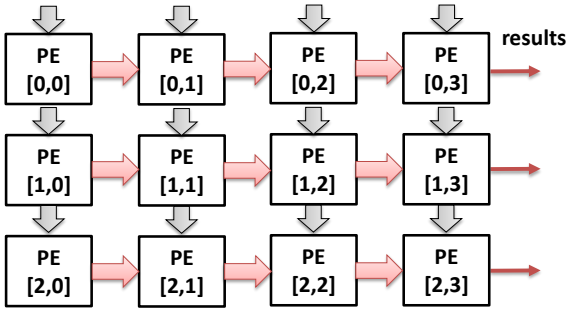


Fig. 3: 2-dimensional pipelined architecture

"$APS \geq 0$" while keeping all of its descendants unchanged.

Algorithm 2 shows the pseudo-code used to insert a node. $\widehat{z}$ is provided as an input to indicate the location where z' is to be inserted. If z' is inserted as an internal node, all the leaf nodes that are descendants of z' have to be changed. If z' is inserted as a new leaf node, z' corresponds to a new rule in $R_{N_{max},W}$; hence we modify an invalid rule into this new rule.

IV. Architecture

In this section, we map our data structures discussed in Section III onto FPGA. To handle each sub-table, we present the modular PE in Section IV-A. We show the overall architecture in Section IV-B. We introduce the dynamic update mechanism in Section IV-C.

A. Modular PE

We show the organization of a modular PE in Figure 2. Each sub-table $S_{i,j}$ is associated with a modular PE$[i,j]$, $i = 0, 1, ..., \frac{N_{max}}{N_n} - 1$, $j = 0, 1, ..., \frac{W}{W_m} - 1$. The modular PE compares the input field value of W_m bits with N_n sub-rules in parallel. The input needs to be compared with the upperbound and the lowerbound of each sub-rule, hence there are $2 \cdot N_n$ parallel comparators. Each comparison result depends on the input field value, the range boundary of a sub-rule stored in the data memory, and the corresponding comparison result from the previous PE. In each modular PE, the comparison results are represented by a Bit Vector (BV) of $(2 \cdot N_n)$ bits [12], [13].

In each PE, a register is used to propagate the BV horizontally to the next PE on its right side; another register is used to propagate the W_m-bit input field value vertically to the next PE below this PE. The data memory is implemented using dual-port distRAM; this allows localized memory access and dual-threaded lookup in each PE [2], [13].

B. 2-dimensional Pipelined Architecture

The modular PEs can be concatenated into a 2-dimensional pipelined architecture. We show an example for $\frac{N_{max}}{N_n} = 3$ and $\frac{W}{W_m} = 4$ in Figure 3. The horizontal arrows indicate the data flow of the BVs and the final lookup results; the vertical arrows denote the data flow of the input field values. Compared to the state-of-the-art implementation, this architecture is scalable with respect to:

1) Field width W. As W increases, more PEs can be used in each horizontal pipeline; the length of the longest wire is not greatly affected by the value of W. We can sustain high throughput even when we scale up the value of W.
2) Maximum number of rules N_{max}. As N_{max} increases, we can use more PEs in each vertical pipeline. We can sustain high throughput for large values of N_{max}.

Note for the state-of-the-art implementation of a decision-tree, the length of the longest wire(s) grows much faster (can be linear) with respect to the number of leaves.

C. Dynamic Updates for Decision-tree

The 2-dimensional pipelined architecture also supports dynamic updates. For each PE, the pins used for input field values can be reused as input pins for overwriting data in the data memory. Each memory write access takes 1 clock cycle; this blocks the lookup process for 1 clock cycle per PE. The memory write accesses propagate through the 2-dimensional array in a diagonal waveform manner, since the control signals have to be propagated from PE to PE in a pipelined fashion. We illustrate the dynamic update mechanism using the example (considering worst-case scenario) in Figure 3:

1: In the first clock cycle, the range boundary used for the first comparator in PE$[0,0]$ is being overwritten.
2: In the second clock cycle, the range boundary for the second comparator stored in PE$[0,0]$ is being overwritten; the range boundaries for the first comparators in PE$[0,1]$ and PE$[1,0]$ are also being overwritten.
3: After $2 \cdot N_n$ clock cycles, all the range boundaries in PE$[0,0]$ have been overwritten; PE$[0,0]$ resumes its lookup process.
3: This process continues until all the PEs overwrite their data. The memory accesses to $\frac{N_{max} \times W}{N_n \times W_m}$ PEs complete after $(2 \cdot N_n + \frac{N_{max}}{N_n} + \frac{W}{W_m} - 1)$ clock cycles.

V. Performance Evaluation

We conducted experiments using Verilog on Xilinx Vivado Design Suite 2013.4 [14], targeting the Virtex-7 XC7VX1140t FLG1930 -2L FPGA [15]. This device has 1100 I/O pins, 218800 logic slices, 67 Mb BRAM, and can be configured to realize large amount of distRAM (up to 17 Mb). A conservative timing constraint of 250 MHz is used for place-and-route process. Maximum achievable clock rate and resource

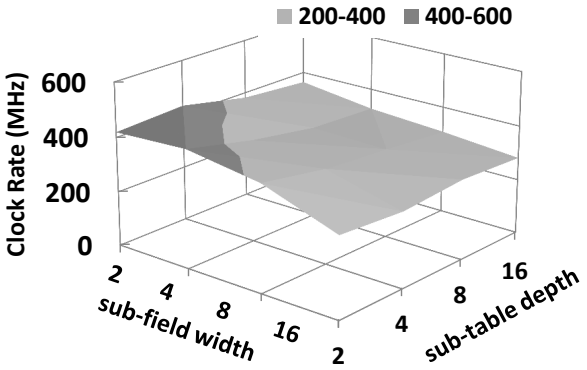


Fig. 4: Choosing W_m and N_n

consumption are shown using post-place-and-route timing reports and resource utilization reports, respectively.

The rest of this section is organized as follows: Section V-A defines the performance metrics. We identify the values of W_m and N_n through experiments in Section V-B. In Section V-C, we compare the performance of our architecture with the state-of-the-art implementation of the same decision-tree. Section V-D demonstrates the scalability of our architecture with respect to the total width of all the fields (W) and the maximum number of leaf nodes in a decision-tree (N_{max}). In Section V-E, we discuss the relationship between the update rate and the sustained throughput.

A. Performance Metrics

The following metrics are widely-used in most of the hardware-based classification engines [2], [7], [16], [17]:

1) Peak/Sustained Throughput: The number of lookups performed per unit time (MLPS) without/with dynamic updates.

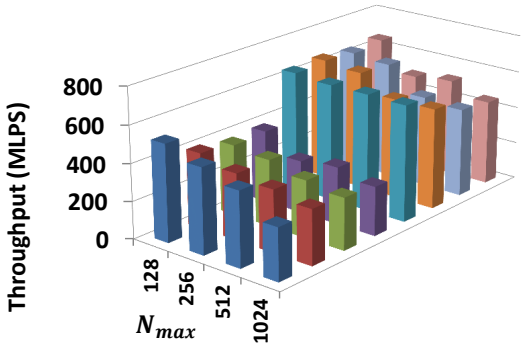
2) Resource Consumption: Total amount of hardware resources consumed by the lookup engine, including logic slices and I/O pins utilized on FPGA.

B. Empirical Optimization of Parameters

We have to properly choose the values of W_m (the width of each sub-field) and N_n (the depth of each sub-table) in order to achieve the best clock rate (and also the best throughput) on FPGA. In this section, we use relatively small values of W ($W = 80$, the total width of all the fields) and N_{max} ($N_{max} = 128$, the maximum number of leaves), and vary the values of W_m and N_n. As can be seen later, the best combination of W_m and N_n can also sustain high performance even for larger values of W_m and N_n.

The values of W_m and N_n both range from 2 to 16; for larger values, the clock rate drops dramatically. We show the achievable clock rate in Figure 4. As can be seen:

- The highest clock rate achievable is above 400 MHz, which corresponds to a lookup rate of over 800 MLPS.
- Smaller values of W_m and N_n perform better, since individual logic cells on FPGA can be used more efficiently. Large values of W_m and N_n lead to more resource consumption per PE, as well as more routing resources between logic cells; the slow interconnections between large amounts of logic cells degrade the clock rate.
- We can either choose $W_m = 2$ and $N_n = 4$, or $W_m = 4$ and $N_n = 2$, since there is no significant difference with respect to clock rate. We choose $W_m = 4$ and $N_n = 2$ in our work.



	128	256	512	1024
BL, W=32	522.33	459.66	401.93	280.35
BL, W=64	404.45	356.95	333.78	295.90
BL, W=96	379.51	358.17	308.98	279.25
BL, W=128	388.12	281.49	306.75	263.26
Prop., W=32	646.20	630.32	630.72	625.20
Prop., W=64	658.33	637.76	540.54	541.27
Prop., W=96	642.88	625.59	491.16	474.72
Prop., W=128	660.72	499.25	522.60	454.96

Fig. 5: Comparison of peak throughput

C. Comparison with State-of-the-art

We show the comparison of the proposed design (Prop.) with the state-of-the-art implementation in Figure 5. As a baseline (BL), we denote the following implementation as the state-of-the-art implementation of a decision-tree [2]: (1) Each tree level is mapped directly into a pipeline stage; all the field values in a tree level is stored in the same memory module of a pipeline stage. (2) Both distRAM and BRAM can be used in each pipeline stage; the synthesis tool [14] chooses which one to use in each pipeline stage with high optimization effort. The proposed design and the state-of-the-art implementation both employ dual-port memory on the same FPGA.

To make a fair comparison, we randomly generated decision-trees (not necessarily complete or balanced) for each combination of W and N_{max}. For each pair of (W, N_{max}) and a specific implementation, the throughput do not differ very much between different decision-trees; therefore we only generated 10 decision-trees for each pair of (W, N_{max}). The decision-trees are chosen to be binary, although our architecture also applies directly to trees of higher degrees. In Figure 5, the average performance of 10 decision-trees is shown for each pair of (W, N_{max}). We can observe that our implementations achieve better (2×) peak throughput compared to the state-of-the-art implementation.

TABLE II: Resource consumption

W	N_{max}	Logic slices (%)	I/O (%)	Peak Throughput (MLPS)
80	128	4.95	19.45	597.01
	256	9.04	21.63	626.37
	384	13.30	23.81	614.06
	512	18.45	26.00	569.80
160	128	9.11	35.81	700.28
	256	17.74	38.00	580.04
	384	25.95	40.18	566.09
	512	35.17	42.36	564.97
240	128	13.36	52.18	591.89
	256	26.58	54.36	565.61
	384	39.41	56.54	564.17
	512	53.61	58.72	557.56
320	128	17.64	68.54	609.57
	256	34.95	70.72	585.30
	384	55.00	72.90	538.78
	512	70.72	75.09	536.19

D. Scalability

We show the throughput and resource consumption of the proposed design with respect to various values of W and N_{max} in Table II. We choose W to range from 80 to 320; this range covers most of values of W in many decision-tree-based network classification problems [2], [12], [16]. We have the following observations:

- As we scaled up the values of W and N_{max}, we observe little throughput degradation; we sustain over 530 MLPS throughput for large values of W and N_{max}.
- The resource consumption scales linearly with respect to both W and N_{max}. However, even when over 70% resources are utilized, we can still sustain high throughput.

In Table II, the largest values of W and N_{max} supported by the targeted FPGA are 320 and 512, respectively. Note in Section V-C, we have also used $W = 128$ and $N_{max} = 1024$. This means the values of W and N_{max} can be tuned to target a specific application.

E. Sustained Throughput and Dynamic Updates

As can be seen, Algorithm 2 has higher complexity than Algorithm 1, since Algorithm 2 requires a chain of modifications to be performed in the corresponding PEs. Hence we pessimistically assume all updates are node insertions.

Note although a single update of the tree node may trigger many updates on the rule table, the distributed update algorithms in Algorithm 1 and Algorithm 2 allow the update time to be overlapped among multiple PEs. As a result, the time overhead to update a single PE is $2 \cdot N_n$ clock cycles in the worst case.

Let U denote the update rate, f the maximum clock rate achieved on FPGA, T the sustained throughput. Using the above assumptions and notations, the sustained throughput with the worst-case[2] dynamic updates, is given by:

$$T = 2(f - 2 \cdot N_n \cdot U) \quad (1)$$

[2]Assuming all the update operations are node insertions, and all the $2 \cdot N_n$ range boundaries are modified in each PE.

The typical dynamic update rate required by network applications, is much less than the lookup rate to be sustained [13]. For example, the update rate for Internet IP address lookup is usually less than 10 K updates per second [18]. This indicates our sustained throughput is almost the same as the peak throughput without any update operation ($T \approx 2f$). Note that most of the existing dynamically updatable architecture introduce much higher update overhead [7], [19].

VI. Conclusion

In this paper, we presented a scalable and dynamically updatable lookup engine for generic decision-trees. The proposed approach can be generalized for other decision-tree-based applications. In the future, we plan to investigate self-learning mechanism on this architecture.

References

[1] R. Sedgewick, "Left-Leaning Red-Black Tree," http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf, September 2008.

[2] D. Tong, L. Sun, K. Matam, and V. Prasanna, "High Throughput and Programmable Online Trafficclassifier on FPGA," in *Proc. of FPGA*, 2013, pp. 255–264.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[4] M. Fujita and R. Murgai, "Delay Estimation and Optimization of Logic Circuits: a Survey," in *Proc. of ASP-DAC*, 1997, pp. 25 –30.

[5] T. Kowsalya, "Tree Structured Arithmetic Circuit by using Different CMOS Logic Styles," *ICGST Intl. Journal on Prog. Dev., Cir. and Sys., PDCS*, vol. 8, pp. 11–18, 2008.

[6] D. Sheldon and F. Vahid, "Don't Forget Memories: A Case Study Redesigning a Pattern Counting ASIC Circuit for FPGAs," in *Proc. of CODES+ISSS*, 2008, pp. 155–160.

[7] Y.-H. E. Yang and V. K. Prasanna, "High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA," in *Proc. of FPGA*, 2010, pp. 83–92.

[8] H. Le and V. Prasanna, "Scalable High Throughput and Power Efficient IP-Lookup on FPGA," in *Proc. of FCCM*, 2009.

[9] M. Bando and J. Chao, "FlashTrie: Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps," in *Proc. of INFOCOM*, 2010, pp. 1–9.

[10] H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Exploiting Fast Carry-chains of FPGAs for Designing Compressor Trees," in *Proc. of FPL*, 2009, pp. 242–249.

[11] F. Liu, F. Forouzandeh, O. Mohamed, G. Chen, X. Song, and Q. Tan, "A Comparative Study of Parallel Prefix Adders in FPGA Implementation of EAC," in *Proc. of DSD*, 2009, pp. 281 –286.

[12] Y. R. Qu, S. Zhou, and V. Prasanna, "Scalable Many-Field Packet Classification on Multi-core Processors," in *Proc. of SBAC-PAD*, 2013, pp. 33–40.

[13] Y. R. Qu, S. Zhou, and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA," in *Proc. of ANCS*, 2013, pp. 125–136.

[14] "Vivado Design Suite," http://www.xilinx.com/products/design-tools/vivado/.

[15] "Virtex-7 FPGA Family," http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm.

[16] H. Le and V. Prasanna, "High-Throughput IP Lookup Supporting Dynamic Routing Tables using FPGA," in *Proc. of FPT*, 2010, pp. 287 –290.

[17] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: Fast and Efficient IP Lookup Architecture," in *Proc. of ANCS*, 2006, pp. 51–60.

[18] B. Vamanan and T. N. Vijaykumar, "TreeCAM: Decoupling Updates and Lookups in Packet Classification," in *Proc. of CoNEXT*, 2011, pp. 27:1–27:12.

[19] W. Jiang and M. Gokhale, "Real-Time Classification of Multimedia Traffic Using FPGA," in *Proc. of FPL*, 2010, pp. 56–63.