Low Power Decision Tree-Based Flow Search Engine

Eita KOBAYAHSHI, Norio YAMAGAKI,
Takashi TAKENAKA, Satoshi KAMIYA

System IP Core Research Laboratories
NEC Corporation
Shimonumabe 1753, Kawasaki, Kanagawa, JAPAN
{e-kobyaashi@fg, n-yamagaki@cj,
takenaka@aj, kamiya@ak}.jp.nec.com

Masato MOTOMURA

Graduate School of Information Science and Technology
Hokkaido University
Kita 8, Nishi 5, kita-ku, Sapporo, Hokkaido, JAPAN
motomura@ist.hokudai.ac.jp

Abstract— This paper presents a novel architecture for a low-power flow search engine, which is seeking the routing rule on network switch. In general, search engines are implemented by Ternary Content Addressable Memory (TCAM) with huge power consumption. To reduce the power, we propose a combination of the decision tree based pipelines using general-purpose memories, and the linear search pipelines that can compensate for the shortcomings of the tree-based pipelines, instead of TCAM. We also developed a design methodology to configure design parameters while taking into account of the robustness for the fluctuation of the network property. The evaluation results show that our proposed architecture reduces the power consumption by up to 92% with maintenance of throughput as much as TCAM.

I. Introduction

Recently, network routers and switches should process the huge amount of flows in a short time, according to increasing of the network traffic. Then, multi-field packet classification is increasingly becoming more important to enable fine-grained flow control for high performance routers and switches. For this purpose, TCAM, which can process packets-by-packets with low latency and high throughput, is widely used as a flow search engine. TCAM is a device for simultaneous parallel comparison between the entered search key and the inside data strings which consist of "0" , "1" and wildcard ("*"). Thus, this mechanism can promise the fast search and the constant processing time.

However, in general, a TCAM is power-hungry, as the power consumption per bit of TCAM is 150 times as large as the power consumption per bit of static random access memories (SRAMs)[1]. Due to the rapid growth of the size of routing rule set and routing rule width of string to meet the development of the internet demands, it seems to increase the size of TCAM used in a network node, in the near future. Then, TCAM won't be able to apply for network routers and switch because of its huge power consumption for the heavy traffic internet.

To replace TCAM with huge power consumption, the several approaches by the flow search hardware engine using SRAM-based pipelines have been proposed [1][2]. One approach is based on the search engine with the decision tree algorithm, such as HiCuts[3] or HyperCuts[4]. However, these methods have a disadvantage since they require the large amount of SRAM thanks to the rule duplication.

In this paper, we propose a novel hardware architecture of a flow search engine. The proposed search engine is implemented without rule duplication, by the integration with linear search and decision tree-based search. We also develop a design methodology for our proposed architecture to configure parameters such as number of decision trees. It can also adapt the robustness to the fluctuation of the network property. In our experiments, we evaluated the power consumption for the several implementations of our search engine with 128-bits search keys and four rule sets, whose size varies from 64k to 512k. Evaluation results show that our search engine reduces the power consumption up to 92% without degrading performance compared to TCAM.

II. Related Works

A. HyperCuts

As a previous work, HyperCuts[2] was proposed by Singh *et al.* as a decision tree based packet classification algorithm. In this algorithm, a rule composed of d-fields was viewed mapping on to the d-dimensional rule space. The d-dimensional rule space is divided into subspaces in accordance with *dividing criteria*, which defines number of partitions and partition field for space division. Hence, each divided subspace is assigned to a leaf node in a decision tree, rules in the subspace are allocated the corresponding leaf node. In the case that number of rules exceeds threshold, the subspace assigned to the node is re-divided into child subspaces, and all of rules in parent subspace are reallocated into the child leaf nodes. This reallocation is carried out until the number of rules assigned

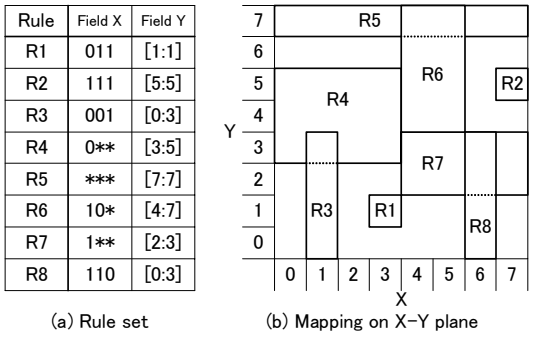

Fig. 1. An example of rule set.

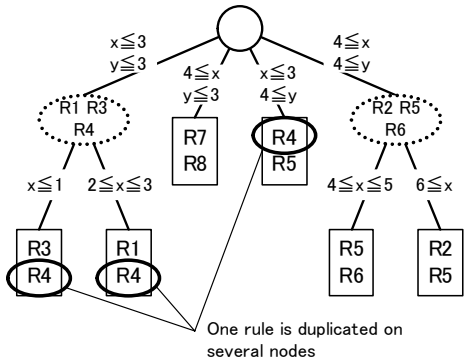

Fig. 2. Allocation to the tree of rules.

to the node fall below the threshold.

Fig.1 shows an example of a rule set mapping onto the rule space. As shown in Fig. 1 (a), a rule composed of two fields. Let each field be Field X and Field Y, respectively. Assume that Field X is defined in the form of prefix match, which was a string consisting of "0", "1" and wildcard. Field Y is also defined in the form of range match, which has minimum and maximum values.

Fig. 1 (b) shows a rule mapping the rule set shown in Fig.1 (a) on to the 2-dimensional rule space. In this example there are four child nodes at tree height 1 because 2-dimensional rule space is initially divided evenly along the lines with x-axis and y-axis, respectively. Suppose that number of rules threshold is 2, there are several nodes exceeds the number of threshold, re-dividing of the subspace and reallocation of the rules to new child nodes are carried out.

In the subspace dividing phase, a rule defined across multiple subspaces is duplicated multiple rules as many as number of covering subspaces. Consequently, as shown in Fig.2, those duplicated rules are allocated in multiple leaf nodes. This fact is called *rule duplication*. It causes a memory shortage because each multiple rule requires own memory space to be allocated to each leaf nodes.

B. Decision Forest

To curb the influence of rule duplication, *Decision Forest*[5] algorithm was proposed by Jiang *et al.* As shown

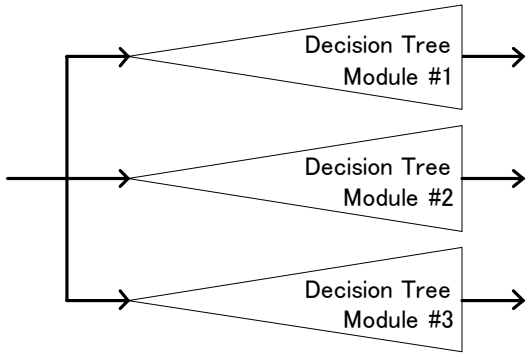

Fig. 3. Decision forest.

in Fig.3, Decision Forest uses multiple decision trees that implement HyperCuts so that rules are flexibly allocated to the decision trees to avoid as much the rule duplication as possible. This algorithm also achieves equal search performance compared to HyperCuts since each of decision trees execute search processing independently and in parallel. However, Decision Forest algorithm can't help tolerating rule duplication in the case that any of decision trees cause the rule duplication for a rule. Therefore, Decision Forest can reduce the rule duplication, but cannot eliminate it completely.

III. Proposed Search Engine Architecture

A. Overview

Fig.4 shows overview of the proposed flow search engine. This flow search engine consists of multiple decision tree modules and (multiple) linear search module(s). All rules are partitioned into some groups and each group is allocated onto one of the decision tree modules or one of the linear search modules. In the decision tree module side, similar to Decision Forest[5], the group of rules are also partitioned into some sub-groups and each sub-group is allocated onto a leaf node. Once a search key is entered into one of the decision tree modules, the key takes a path from a root node of the tree to a leaf node and it is finally compared to the sub-group of rules allocated to the leaf node. On the other hand, in the linear search module side, all rules in the group are allocated without any partition into sub-groups and a search key entered to the linear search module is compared to all rules allocated to the linear search module.

As shown in the Fig.4, once a search key is entered the flow search engine, the key is copied and distributed to all modules. All of decision tree modules and linear search modules process the key in parallel. After all modules generate their results, one of these results is selected as a final result of the search engine based on rule's priority.

The criterion for partitioning of rule space is as follows. Only rules which do not cause the rule duplication can be allocated to the decision tree module. Each decision

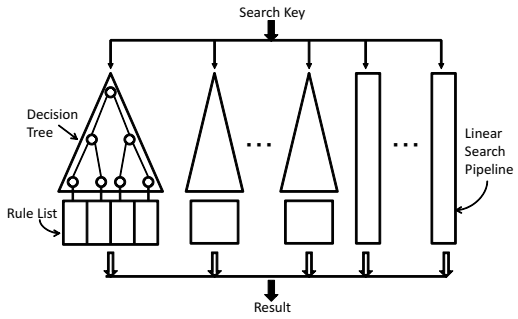

Fig. 4. Overview of our flow search engine.

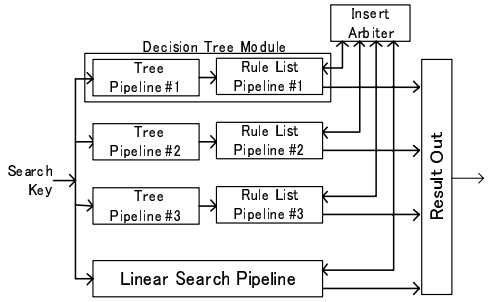

Fig. 5. Schematic diagram of our flow search engine.

tree has different dividing criteria, so most of rules are allocated to the one of these decision trees. However, if a rule which causes the rule duplication in any of decision tree modules, then the rule is allocated to one of the linear search modules. Any rules in the linear search modules are not duplicated since the linear search module does not divide the rule space. According to this criterion, the rule duplication is completely eliminated by using linear search combined with multiple decision trees in our flow search engine. Additionally, we use multiple linear search modules for the sake of preventing increasing in length of a single linear search.

B. *Outline of each modules in our flow search engine.*

Fig.5 shows the outline block diagram of our flow search engine. This section explains each module.

B.1 Decision Tree Module

Decision tree modules in our flow search engine are equipped with additional rule insert/delete faction based on modules in [5]. This module has a pipeline structure which consists of serially-cascaded a *tree pipeline* and a *rule pipeline*. A sort processing is implemented in the tree pipeline and a comparing processing rules with a search key is implemented in the rule pipeline. A search key is compared with rules allocated in a leaf node at the rule pipeline after the search key arrives in a final stage of the tree pipeline. This final stage of a tree pipeline corresponds to leaf node of the decision tree. A decision tree module also has a *entry counter* which manages a number of rules in each leaf node. This counter values are increased/decreased by a rule insert/delete processing.

TABLE I
Rule Profile

♯ of Rules	Wildcard ratio				
	Field1	Field2	Field3	Field4	Field5
64,000	0.11669	0.0634	0.8627	0.4010	0.1964

B.2 Linear Search Pipeline

Linear search module also has a pipeline structure. This module processes sequential search for finding a particular rule in a list without sort processing by tree. The behavior of these modules similar to a rule pipeline in decision tree modules, however each stage of pipeline has at most one rule. Each linear search module also has a counter register which manages a number of rules in this module.

B.3 Insert Arbiter

Our flow search engine is characterized by the having a dynamic rule insert/delete function. When the rule insert request is entered, the *Insert arbiter* determines the rule is inserted in which decision tree or linear search module.

B.4 Result Out

The *Result out* module narrow search results to one in accordance with the priority corresponding with the rule. Those of search results are generated by each decision tree modules or linear search modules in parallel.

IV. Parameter Configuration

A linear search pipeline has large power consumption due to the reason why all of rules in linear search module have to be compared with a search key. To achieve power reduction by our flow search engine, the delicate balance of the number of rules between decision trees modules and linear search modules have to be maintained. In this section, we explain the design method to configure parameters of our search engine. This method also can estimate the robustness to the fluctuation of the network property in terms of the ability of having the number of rules.

A. *Rule Profile*

A *rule profile* consists of total number of rules and wildcard ratio of each field. The wildcard ratio is the ratio of rules which has wildcard. Table I shows an example of the rule profile. This example was obtained from analysis of 5 field seed filter in ClassBench[6]. The table shows that the total number of rules is 64,000 and its 11% of rules have wildcard in Field 1 and so on.

B. *Overview of Design Method*

Fig.6 shows a flow chart of our design method. Once a rule profile is given, our method makes a configuration

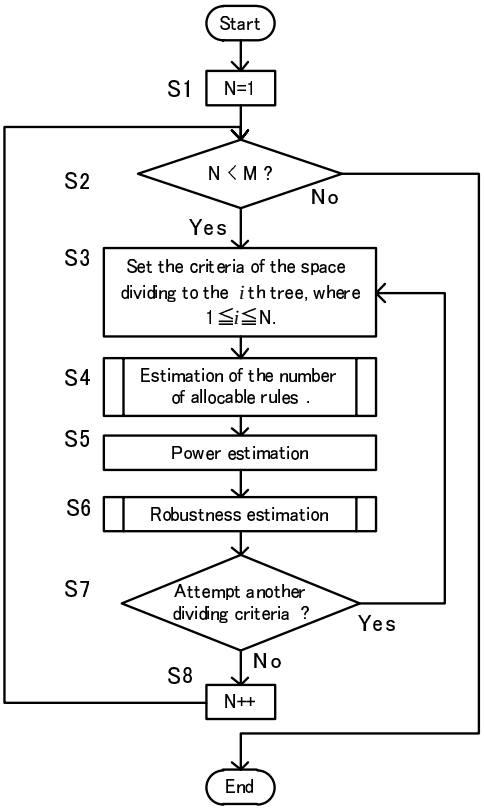

Fig. 6. Flow chart of robustness estimation process.

of design parameters. Next it estimates total amount of hardware resources such as memories in the decision tree module registers in the linear search pipeline. Based on estimation of the hardware resources, it calculates its power consumption. It also evaluates robustness for network fluctuation. These processes are repeated until all configurations are examined.

The detail of this flow chart is described further below.

INPUT Rule profile, Maximum number of decision trees M, Maximum number of allocated rules per leaf node.

OUTPUT Number of Decision trees and leaf nodes, Dividing criteria of each decision tree, Number of rules allocated to linear search module.

Objective Function Power consumption, Robustness.

S1 Let the number of decision trees $N = 1$.

S2 If $N == M$ then exit this flow.

S3 Set the dividing criteria to N decision trees.

S4 Estimate the number of rules allocated to decision trees and linear search modules respectively. Required number of leaf nodes of those trees are also estimated (see section C.).

S5 Estimates the amount of memories and power consumption from the number of allocated rules.

S6 Estimates the robustness of the current configuration.

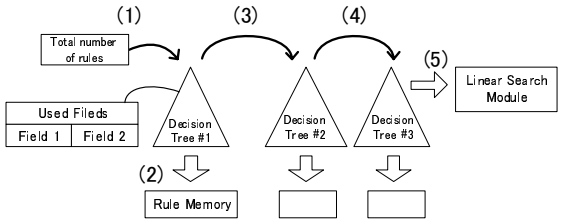

Fig. 7. Estimation of number of rules

S7 If try another dividing criteria , then return to step S3.

S8 Let $N \leftarrow N + 1$, then return to step S2.

In above flow S3 to S7, appropriative dividing criteria are explored to N decision trees. The number of rules allocated to decision trees and linear search modules are estimated by the method in section C. based on the dividing criteria. The power consumption can be estimated from those numbers of rules, register power and memory power. With all these factors, the dividing criteria have a major impact on the amount of power consumption. To reduce the power consumption, the proper dividing criteria have to be explored in this flow. Though there are several exploration methods such as montecarlo or simulated annealing, we use a brute force search method in our evaluation.

C. Number of Rules Estimation

In S4 of Fig.6, the number of rules allocated to each module is estimated from the dividing criteria under the condition that rule duplication is completely eliminated. Fig.7 shows an example of number of rules estimation. Suppose that first decision tree uses field1 and field2 as the dividing criteria and wildcard ratio of field1 and field2 is p_1 and p_2, respectively. This wildcard ratio means that p_1 percent of rules are duplicated from the standpoints of field1, because the rule duplication always occur when try to divide the rule, of which field1 is defined in wildcard, at field1-axis. Suppose total number of rules is R, the number of allocated rule is $R(1-p_1)(1-p_2)$ to the decision tree ♯1 because the decision tree ♯1 uses field1 and field2 to divide the rule space.

As shown in Fig.7, suppose that total number of rules R **(1)**, $R(1-p_1)(1-p_2)$ of rules are allocated rule memory **(2)**, and other rules are considered as a input to next decision tree **(3), (4)**. Rules, which are not allocated to the decision tree ♯3, are considered allocated rules to linear search module. In this way, the numbers of rules allocated to each module are estimated.

D. Robustness Estimation

In S6 in Fig.6, robustness of the configured parameters for network property fluctuation is estimated. Because it was based on the specific rule profile, the search engine

with parameters configured in section B. may not satisfy the target performance from the view of the allocatable number of rules. Since this problem may occur when the situation of network is changed, it is desired that design methodology can configure robust parameters for the network fluctuation. The robustness estimation method is built in our design flow. Suppose that the wildcard ratio of Field1 is p_1 in the rule profile. In this estimation, normal distribution of wildcard ratio $N(p_1, \sigma)$ is modeled where p_1 is the average, σ is given as standard deviation with propriety. This σ is measure of dispersion. This normal distribution of wildcard ratio modeled for all fields like the above. Other wildcard ratios with dispersion from original are obtained by sampling those normal distributions of wildcard ratios. We use those other wildcard ratios to estimate the number of rules likewise section C.. The robustness of configured parameters can be evaluated by comparing newly-required hardware resource to original configuration. If required the number of linear search modules exceeds the original parameters configured in this design flow, then this parameters are evaluated as a low tolerance for the change of network situation. We evaluate the robustness of configured parameters in such a way that percentage of number of times evaluated as a high tolerance in a hundred of wildcard ratio with dispersion, which was obtained from a hundred of normal distribution sampling.

V. Experimental Results

A. Target Performance

We supposed that target search rate is 250M search/sec, throughput is 168Gbps (64 bytes packet), latency is 200ns, respectively. The target frequency is 250MHz because our flow search engine can receive a input in every cycles with the aid of its pipeline architecture. We also supposed that the search key length is 128bits, and total number of rules is varied as 64k, 128k, 256k and 512k.

B. Configured Parameters

TableII shows configured parameters for each target performance. To satisfy the 200ns latency, the tree height is 1 and the number of stages of rule pipeline is 20. We use ClassBench[6] to obtain rule profile for the basis of configuration each parameter. From this rule profile, we decide the configuration which achieves the lowest power consumption by exploring the design space. In this estimation, we use the previously-evaluated power of registers and clock tree.

C. Power Evaluation

We implemented our search engine using Verilog-HDL on the basis of ASIC implementation. We use a 90nm

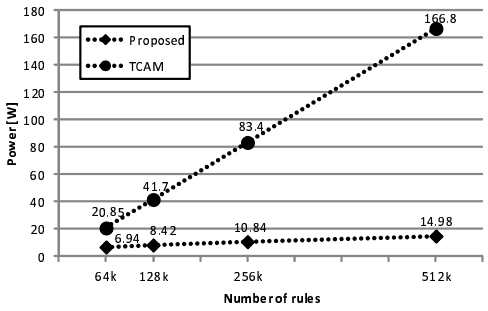

Fig. 8. Power comparison@128bit search key.

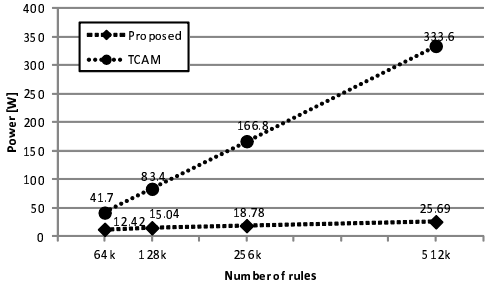

Fig. 9. Power comparison@256bit search key.

standard cell library with logic synthesis done by Synopsis Design Compiler to confirm that maximum operating frequency met the target. Power evaluation was also done by Apache Design Solutions PowerArtist-PT under the situation that search processes were performed frequently at most.

Fig.8 and Fig.9 shows a result of total power evaluation. According to this graph, our flow search engine archives power reduction by up to 92% with TCAM power[7].

To evaluate the allocatable number of rules, we implement the software simulator which simulates the action of rule insert. At first, we configured the parameters using the design flow in section IV. We use a rule profile as shown in tableI, so the target allocatable number of rules is 64,000. Due to the lack of real-life flow rules, we generated 5-tuple like rules. Each rule was composed of 5 header fields that each of field are assumed Source IP (32-bits), Destination IP (32-bits), Source Port (16-bits), Destination Port (16-bits) and Protocol Type (8-bits), respectively. We generated each rule as follows:

1. Each field is stochastically set as a wildcard at a rate of tableI.
2. In the case the field isn't defined as a wildcard, the value is set randomly from its possible exact values.

We simulated the rule allocation to leaf nodes according to the policy as follows: a rule could be allocated to multiple decision trees, we select the one tree so as to allocate to the leaf node which has most fewer rules at that point. To simplification, suppose that heights of all trees are 1 and number of leaf nodes of all trees is the same.

TABLE II
Target performance and configured parameters

		Conf 1	Conf 2	Conf 3	Conf 4	Conf 5	Conf 6	Conf 7	Conf 8
Target Performance	Search Rate	250Msps							
	Search Key Length	128bit				256bit			
	♯ of Rules	64k	128k	256k	512k	64k	128k	256k	512k
Configuration Performance	Tree Height	1							
	Length of Rule List	20							
	♯ of Decision Trees	4	5	6	8	4	5	6	8
	♯ of Leaf Node	4,096	4,096	8,192	8,192	4,096	4,096	8,192	8,192
	Amount of SRAM	10MB	12.5MB	30MB	40MB	20MB	25MB	60MB	80MB
	♯ of Linear Search Pipeline	4	4	3	6	4	4	3	6

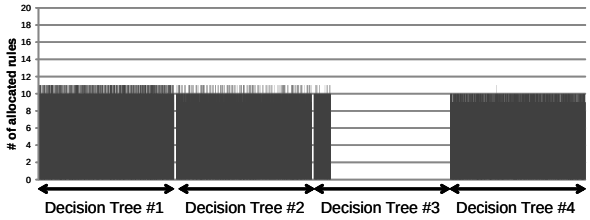

Fig. 10. Number of rules by each leaf node. (64,000 rules)

TABLE III
Assigned Field

Assigned fields			
Tree ♯1	Tree ♯2	Tree ♯3	Tree ♯4
Filed2, Field4	Field1, Field5	Field5	Field2

For robustness evaluation, we supposed normal distribution under the value of standard division is $\sigma = 5\%$. We obtained a hundred of different rule profile with dispersion from the sampling of normal distribution. We finally chose a configuration which can allocate all of rules under the any of a hundred of rule profile.

D. Evaluation of the number of the rules

Fig.10 shows a result of rule allocation simulation which represents numbers of allocated rules per each leaf node. TableIII shows an assigned field using in this simulation. Since the decision tree ♯3 use only a 8bit field5 to divide a rule space, there is a blank leaf which doesn't have a rules. This is because that only $2^8 = 256$ of subspaces are generated by rule space dividing whereas other trees has 4096 of leaf nodes.

As shown in Fig.10, there are rooms for addition rules on the whole because average number of rules in each leaf is approximately 10. For this reason, we use a policy that the numbers of rules in each node have to be balanced preferably. In this simulation, there are no rules overflows from our flow search engine and no rules allocated in the linear search modules. On the contrary, however a wasteful resource is required in this current design flow, our flow search engine achieves a required performance.

VI. Summary and Conclusions

This paper proposed a decision tree-based flow search engine aimed at replacing TCAM. Our flow search engine achieves reduction of the power approximately by 90%. To achieve the target performance, we also proposed a specific design flow for our architecture. This design flow can estimate the robustness of configured parameters with the aim of applying the actual network appliance. Our future work includes the development of our design flow so as to be able to configure more flexibly and effectively.

Acknowledgments

This work was partly supported by Ministry of Internal Affairs and Communications.

References

[1] D.E. Taylor, "Survey and taxonomy of packet classification techniques," ACM Computing Surveys, vol.37, pp.238–275, Sept. 2005.

[2] P. Gupta and N. McKeown, "Algorithms for packet classification," IEEE Networks, vol.15, pp.24–32, 2001.

[3] P. Gupta and N. McKeown, "Classifying packets with hierarchical intelligent cuttings," IEEE Micro, vol.20, pp.34–41, Jan. 2000.

[4] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," In proc. of Conf. on Applications, technologies, architectures, and protocols for computer communications, pp.213–224, 2003.

[5] W. Jiang, V.K. Prasanna, and N. Yamagaki, "Decision forest: A scalable architecture for flexible flow matching on fpga," In proc. of International Conf. on Field programmable Logic and Application, pp.394–399, 2010.

[6] D.E. Taylor and J.S. Turner, "Classbench: a packet classification benchmark," INFOCOM. 24th Annual Joint Conf. of the IEEE Computer and Communications Societies. Proceedings IEEE, vol.3, pp.2068–2079, March 2005.

[7] B. Vamanan, G. Voskuilen, and T.N. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," In proc. of the ACM SIGCOMM 2010, pp.207–218, 2010.