Large-Scale Wire-Speed Packet Classification on FPGAs *

Weirong Jiang
Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089, USA
weirongj@usc.edu

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

ABSTRACT

Multi-field packet classification is a key enabling function of a variety of network applications, such as firewall processing, Quality of Service differentiation, traffic billing, and other value added services. Although a plethora of research has been done in this area, wire-speed packet classification while supporting large rule sets remains difficult. This paper exploits the features provided by current FPGAs and proposes a decision-tree-based, two-dimensional dual-pipeline architecture for multi-field packet classification. To fit the current largest rule set in the on-chip memory of the FPGA device, we propose several optimization techniques for the state-of-the-art decision-tree-based algorithm, so that the memory requirement is almost linear with the number of rules. Specialized logic is developed to support varying number of branches at each decision tree node. A tree-to-pipeline mapping scheme is carefully designed to maximize the memory utilization. Since our architecture is linear and memory-based, on-the-fly update without disturbing the ongoing operations is feasible. The implementation results show that our architecture can store 10K real-life rules in on-chip memory of a single Xilinx Virtex-5 FPGA, and sustain 80 Gbps (i.e. 2× OC-768 rate) throughput for minimum size (40 bytes) packets. To the best of our knowledge, this work is the first FPGA-based packet classification engine that achieves wire-speed throughput while supporting 10K unique rules.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; C.2.0 [**Computer Communication Networks**]: General—*Security and protection*

General Terms

Algorithms, Design, Performance, Security

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

FPGA'09, February 22–24, 2009, Monterey, California, USA.

Keywords

Packet classification, Pipeline, SRAM, FPGA, Decision tree

1. INTRODUCTION

The development of the Internet demands next-generation routers to support a variety of network applications, such as firewall processing, Quality of Service (QoS) differentiation, virtual private networks, policy routing, traffic billing, and other value added services. In order to provide these services, the router needs to classify the packets into different categories based on a set of predefined rules, which specify the value ranges of the multiple fields in the packet header. Such a function is called *multi-field packet classification*. Due to the rapid growth of the rule set size, as well as the link rate, multi-field packet classification has become one of the fundamental challenges to designing high speed routers. For example, the current link rate has been pushed beyond the OC-768 rate, i.e. 40 Gbps, which requires processing a packet every 8 ns in the worst case (where the packets are of minimum size i.e. 40 bytes). Such throughput is impossible using existing software-based solutions [6].

Recent research in this area seeks to combine algorithmic and architectural approaches, most of which are based on ternary content addressable memories (TCAMs) [12, 19, 24] or a variety of hashing schemes such as Bloom Filters [3, 14, 15]. However, as shown in Table 1, TCAMs are not scalable in terms of clock rate, power consumption, or circuit area, compared to SRAMs. Most of TCAM-based solutions also suffer from range expansion when converting ranges into prefixes [12, 19]. Recently, Bloom Filters have become popular due to their $O(1)$ time performance and high memory efficiency. However, a secondary module is always needed to resolve false positives inherent in Bloom Filters, which may be slow and can limit the overall performance [20]. On the other hand, mapping decision-tree-based packet classification algorithms [5, 18] onto SRAM-based pipeline architecture appears to be a promising alternative [1]. By pipelining the traversal of the decision tree, a high throughput of one packet per clock cycle (PPC) can be sustained.

Table 1: Comparison of TCAM and SRAM technologies (18 Mbits chip)

	TCAM	SRAM
Maximum clock rate (MHz)	266 [16]	450 [17]
Power consumption (Watts)	12 ∼ 15 [8]	1.2 ∼ 1.6 [17]
Number of transistors per bit	16	6

Although some existing decision-tree-based packet classification algorithms claim to be easily pipelined [18], none of them present either hardware implementation details or implementation results. It is also unclear how to implement a decision tree where each node has various numbers of branches. Most of the existing decision-tree-based packet classification algorithms also suffer from high memory consumption and numerous memory accesses, due to rule duplication when building the decision trees. The worst-case memory requirement is $O(N^d)$, where N denotes the number of rules and d the number of fields. The worst-case number of memory accesses is the total number of bits of the packet header to be classified. As a result, the decision tree must be stored in a pipeline consisting of a large number of off-chip SRAMs or even DRAMs, where the memory bandwidth becomes a constraint. Moreover, any static scheme for mapping the decision tree onto the pipeline can result in unbalanced memory distribution across pipeline stages, which implies inefficient memory utilization [1].

FPGA technology has become an attractive option for implementing real-time network processing engines [7, 14, 19], due to its ability to reconfigure and to offer massive parallelism. State-of-the-art SRAM-based FPGA devices such as Xilinx Virtex-5 provide high clock rate and large amounts of on-chip dual-port memory with configurable word width. In this paper, we explore current FPGAs to build a decision-tree-based packet classification engine which aims to achieve the following goals.

1. Sustain wire-speed throughput of more than 40 Gbps, i.e. OC-768 rate. Considering the worst case where the packets are of minimum size, i.e. 40 bytes, the engine must process 125 million packets per second (MPPS).

2. Support 10K unique rules, i.e. twice the number of rules of the current largest rule set in real life [23], on a single FPGA. Interfacing with external SRAMs should be enabled to handle even larger rule sets.

3. Realize on-the-fly update without disturbing the ongoing operations. Both the rules and the decision tree structure should be updatable without service interruption.

There has been some recent work done on FPGA-based packet classification engines [7, 14, 19]. However, most of them are based on another class of packet classification algorithms, namely, decomposition-based algorithms, which operate independent searches on each field in parallel and then combine the results from all fields. They need TCAMs, Bloom Filters or large tables to perform the combination. As a result, none of them achieve all the above performance requirements, even after their results are projected on the state-of-the-art FPGA devices in an optimistic way.

The contributions of this paper are as follows.

- To the best of our knowledge, this work is among the first discussions of implementing decision-tree-based packet classification algorithms onto FPGAs or even hardware. Unlike a simple design which fixes the number of branches at each tree node, our design allows updating on-the-fly the number of branches on multiple packet header fields.

- We exploit the dual-port high-speed Block RAMs provided in Xilinx Virtex FPGAs and present a SRAM-based two-dimensional dual-pipeline architecture to achieve a high throughput of two packets per clock cycle (PPC). On-the-fly rule update without service interruption becomes feasible due to the memory-based linear architecture. All the routing paths are localized to avoid large routing delay so that a high clock frequency is achieved.

- We identify the sources of rule duplication when building the decision tree. Two optimization techniques, called *rule overlap reduction* and *precise range cutting*, are proposed to minimize the rule duplication. As a result, the memory requirement is almost linear with the number of rules so that ten thousand rules can be fit into a single FPGA. The height of the tree is also reduced, limiting the number of pipeline stages.

- To map the tree onto the pipeline architecture, we introduce a fine-grained node-to-stage mapping scheme which allows imposing the bounds on the memory size as well as the number of nodes in each stage. As a result, the memory utilization of the architecture is maximized. The memory allocation scheme also enables using external SRAMs to handle even larger rule sets.

- Implementation results show that our architecture can store 10K 5-field rules in a single Xilinx Virtex-5 FPGA, and sustain 80 Gbps throughput for minimum size (40 bytes) packets. To the best of our knowledge, our design is the first FPGA implementation able to perform multi-field packet classification at wire speed while supporting a large rule set with 10K unique rules.

The rest of the paper is organized as follows. Section 2 states the problem we intend to solve, and summarizes several representative packet classification algorithms. Section 3 reviews the related work on FPGA design of multi-field packet classification engines. Section 4 presents the SRAM-based pipeline architecture and the set of proposed algorithms including decision tree construction and tree-to-pipeline mapping. Section 5 describes the FPGA implementation of our architecture in details. Section 6 evaluates the performance of the algorithms and the architecture. Section 7 concludes the paper.

2. BACKGROUND

2.1 Problem Statement

An IP packet is usually classified based on the five fields in the packet header: the 32-bit source/destination IP addresses (denoted **SA**/**DA**), 16-bit source/destination port numbers (denoted **SP**/**DP**), and 8-bit transport layer protocol. Individual entries for classifying a packet are called *rules*. Each rule can have one or more fields and their associated values, a priority, and an action to be taken if matched. Different fields in a rule require different kinds of matches: prefix match for SA/DA, range match for SP/DP, and exact match for the protocol field. Table 2 shows a simplified example, where each rule contains match conditions for 5 fields: 8-bit source and destination addresses, 4-bit source and destination port numbers, and a 2-bit protocol value.

A packet is considered matching a rule only if it matches all the fields within that rule. A packet can match multiple

Table 2: Example Rule Set. (SA/DA:8-bit; SP/DP: 4-bit; Protocol: 2-bit)

Rule	Source IP (SA)	Destination IP (DA)	Source Port (SP)	Destination Port (DP)	Protocol	Priority	Action
R1	*	*	2 – 9	6 – 11	Any	1	act0
R2	1*	0*	3 – 8	1 – 4	10	2	act0
R3	0*	0110*	9 – 12	10 – 13	11	3	act1
R4	0*	11*	11 – 14	4 – 8	Any	4	act2
R5	011*	11*	1 – 4	9 – 15	10	5	act2
R6	011*	11*	1 – 4	4 – 15	10	5	act1
R7	110*	00*	0 – 15	5 – 6	11	6	act3
R8	110*	0110*	0 – 15	5 – 6	Any	6	act0
R9	111*	0110*	0 – 15	7 – 9	11	7	act2
R10	111*	00*	0 – 15	4 – 9	Any	7	act1

rules, but only the rule with the highest priority is used to take action. Such a problem we intend to solve, called *best-match* packet classification, is slightly different from *multi-match* packet classification [19] which requires reporting all the matching rules for a packet. These two different kinds of problems target different network applications. The context of multi-match packet classification is in network intrusion detection systems (NIDS) [19], while our work focuses on next-generation high-speed routers.

2.2 Packet Classification Algorithms

A vast number of packet classification algorithms have been published in the past decade. Comprehensive surveys can be found in [6,21]. Most of those algorithms fall into two categories: decomposition-based and decision-tree-based approaches.

Decomposition-based algorithms (e.g. Parallel Bit Vector (BV) [11]), perform independent search on each field and finally combine the search results from all fields. Such algorithms are desirable for hardware implementation due to their parallel search on multiple fields. However, substantial storage is usually needed to merge the independent search results to obtain the final result. Thus decomposition-based algorithms have poor scalability, and work well only for small-scale rule sets.

Decision-tree-based algorithms (e.g. HyperCuts [18]), take the geometric view of the packet classification problem. Each rule defines a hypercube in a d-dimensional space where d is the number of header fields considered for packet classification. Each packet defines a point in this d-dimensional space. The decision tree construction algorithm employs several heuristics to cut the space recursively into smaller subspaces. Each subspace ends up with fewer rules, which to a point allows a low-cost linear search to find the best matching rule. After the decision tree is built, the algorithm to look up a packet is very simple. Based on the value of the packet header, the algorithm follows the cutting sequence to locate the target subspace (i.e. a leaf node in the decision tree), and then performs a linear search on the rules in this subspace. Decision-tree-based algorithms allow incremental rule updates and scale better than decomposition-based algorithms. The outstanding representatives of decision-tree-based packet classification algorithms are HiCuts [5] and its enhanced version HyperCuts [18]. At each node of the decision tree, the search space is cut based on the information from one or more fields in the rule. HiCuts builds a decision tree using local optimization decisions at each node to choose the next dimension to cut, and how many cuts to make in the chosen dimension. The HyperCuts algorithm, on the other hand, allows cutting on multiple fields per step, resulting in a fatter and shorter decision tree. Figure 1 shows the example of the HiCuts and the HyperCuts decision trees for a set of 2-field rules which can be represented geometrically. These rules are actually R1∼R5 given in Table 2, when only SP and DP fields are considered.

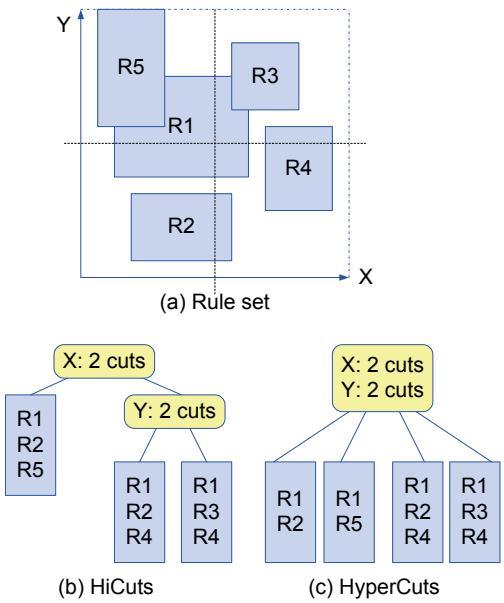

Figure 1: Example of HiCuts and HyperCuts decision trees. ((a) X and Y axes correspond to SP and DP fields for R1∼R5 in Table 2. (b)(c) A rounded rectangle in yellow denotes an internal tree node, and a rectangle in gray a leaf node.)

3. RELATED WORK BASED ON FPGAS

Although multi-field packet classification is a saturated area of research, little work has been done on FPGAs. Most of existing FPGA implementations of packet classification engines are based on decomposition-based packet classification algorithms, such as BV [11] and DCFL [22].

Lakshman et al. [11] propose the Parallel Bit Vector (BV) algorithm, which is a decomposition-based algorithm targeting hardware implementation. It performs the parallel lookups on each individual field first. The lookup on each field returns a bit vector with each bit representing a rule. A bit is set if the corresponding rule is matched on this field; a bit is reset if the corresponding rule is not matched on this field. The result of the bitwise AND operation on these bit vectors indicates the set of rules that matches a given packet. The BV algorithm can provide a high throughput at the cost of low memory efficiency. The memory requirement is $O(N^2)$ where N is the number of rules. By combining TCAMs and the BV algorithm, Song et al. [19] present an architecture called BV-TCAM for multi-match packet classification. A TCAM performs prefix or exact match, while a multi-bit trie implemented in Tree Bitmap [4] is used for source or destination port lookup. The authors never report the actual FPGA implementation results, though they claim that the whole circuit for 222 rules consumes less than 10% of the available logic and fewer than 20% of the available Block RAMs of a Xilinx XCV2000E FPGA. They also predict the design after pipelining can achieve 10 Gbps throughput when implemented on advanced FPGAs.

Taylor et al. [22] introduce Distributed Crossproducting of Field Labels (DCFL), which is also a decomposition-based algorithm leveraging several observations of the structure of real filter sets. They decompose the multi-field searching problem and use independent search engines, which can operate in parallel to find the matching conditions for each filter field. Instead of using bit vectors, DCFL uses a network of efficient aggregation nodes, by employing Bloom Filters and encoding intermediate search results. As a result, the algorithm avoids the exponential increase in the time or space incurred when performing this operation in a single step. The authors predict that an optimized implementation of DCFL can provide over 100 million packets per second (MPPS) and store over 200K rules in the current generation of FPGA or ASIC without the need of external memories. However, their prediction is based on the maximum clock frequency of FPGA devices and a logic intensive approach using Bloom Filters. This approach may not be optimal for FPGA implementation due to long logic paths and large routing delays. Furthermore, the estimated number of rules is based only on the assumption of statistics similar to those of the currently available rule sets. Jedhe et al. [7] realize the DCFL architecture in their complete firewall implementation on a Xilinx Virtex 2 Pro FPGA, using a memory intensive approach, as opposed to the logic intensive one, so that on-the-fly update is feasible. They achieve a throughput of 50 MPPS, for a rule set of 128 entries. They also predict the throughput can be 24 Gbps when the design is implemented on Virtex-5 FPGAs.

Papaefstathiou et al. [15] propose a memory-efficient decomposition-based packet classification algorithm, which uses multi-level Bloom Filters to combine the search results from all fields. Their FPGA implementation, called 2sBFCE [14], shows that the design can support 4K rules in 178 Kbytes memories. However, the design takes 26 clock cycles on average to classify a packet, resulting in low throughput of 1.875 Gbps on average.

Note that both DCFL [22] and 2sBFCE [14] may suffer from false positives due to the use of Bloom Filters, as discussed in Section 1.

Two recent works [9, 13] discuss several issues on implementing decision-tree-based packet classification algorithms on FPGA, with different motivations. Luo et al. [13] propose a method called *explicit range search* to allow more cuts per node than the HyperCuts algorithm. The tree height is dramatically reduced at the cost of increased memory consumption. At each internal node, a varying number of memory accesses may be needed to determine which child node to traverse, which may be infeasible for pipelining. Since the authors do not implement their design on FPGA, the actual performance results are unclear. To achieve power efficiency, Kennedy et al. [9] implement a simplified HyperCuts algorithm on an Altera Cyclone 3 FPGA. They store hundreds of rules in each leaf node and match them in parallel, resulting in low clock frequency (i.e. 32 MHz reported in [9]). Their claim about supporting the OC-768 rate is based on the average cases tested using several specific traffic traces. Since the search in the decision tree is not pipelined, their implementation can sustain only 0.45 Gbps in the worst cases.

4. ARCHITECTURE AND ALGORITHMS

4.1 Algorithmic Motivations

Our work is mainly based on the HyperCuts algorithm which is considered to be the most scalable decision-tree-based algorithm for multi-field packet classification [21]. However, like other decision-tree-based packet classification algorithms, the HyperCuts algorithm suffers from memory explosion due to rule duplication. For example, as shown in Figure 1, rules R1, R2 and R4 are replicated into multiple child nodes, in both HiCuts and HyperCuts trees.

We identify that the rule duplication when building the decision tree comes from two sources: (1) overlapping between different rules, and (2) evenly cutting on all fields. Taking the rule set in Figure 1 as an example, since R1 overlaps with R3 and R5, no matter how to cut the space, R1 will be replicated into the nodes which contain R3 or R5. Since each dimension is alway evenly cut, R2 and R4 are replicated though they do not overlap with any other rule. The second source of rule duplication exists only when cutting the port or the protocol fields of the packet header, since the prefix fields are evenly cut in nature. A prefix is matched from the most significant bit (MSB) to the least significant bit (LSB), which is equal to cutting the value space by half per step.

Accordingly, we propose two optimization techniques, called *rule overlap reduction* and *precise range cutting*, as shown in Figure 2.

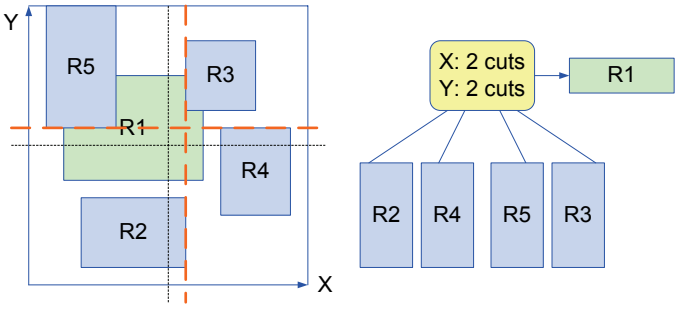

Figure 2: Motivating example

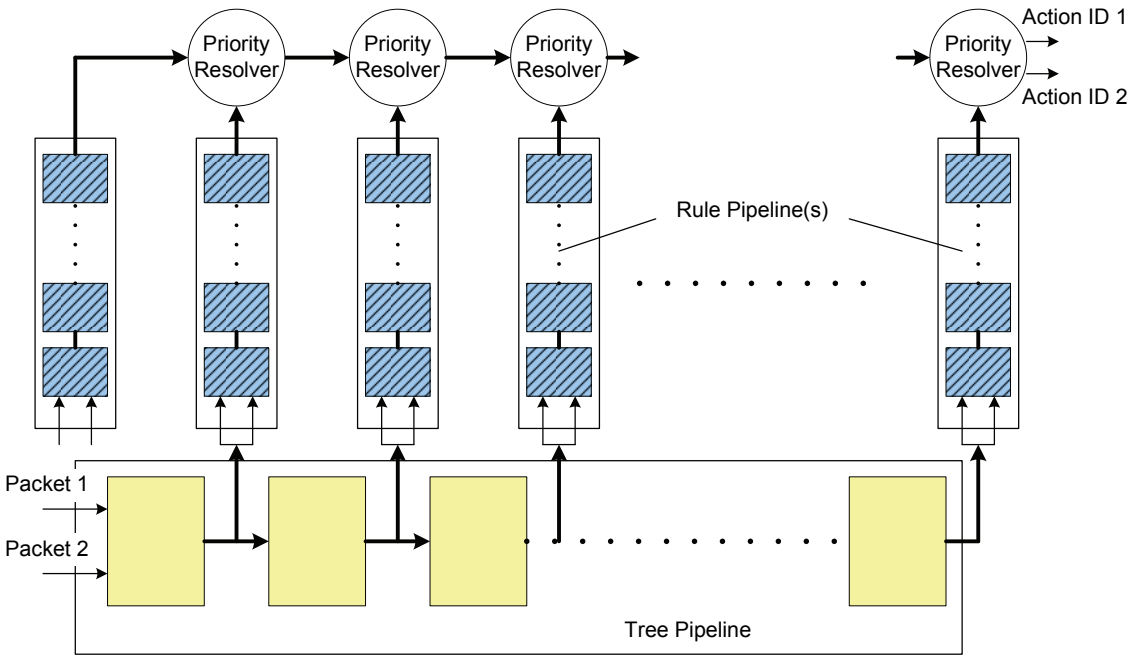

Figure 3: Block diagram of the two-dimensional linear dual-pipeline architecture

- Rule overlap reduction: We store the rules which will be replicated into child nodes, in a list attached to each internal node. These rule lists are called *internal rule lists*, e.g. R1 shown in green in Figure 2.
- Precise range cutting: Assuming both X and Y in Figure 2 are port fields, we seek the cutting points which result in the minimum number of rule duplication, instead of deciding the number of cuts for this field.

As shown in Figure 2, after applying the two optimizations, rule duplication is dramatically reduced and the memory requirement becomes linear with the number of rules. Section 4.3.1 discusses the details about building the decision tree.

The proposed *rule overlap reduction* technique is similar to the *push common rule upwards* heuristic proposed by the authors of HyperCuts [18], where rules common to all descendant leaves are processed at the common parent node instead of being duplicated in all children. However, the *push common rule upwards* heuristic can solve only a fraction of rule duplication that can be solved by our *rule overlap reduction* technique. Taking the HyperCuts tree in Figure 1(c) as an example, only R1 will be pushed upwards while our technique allows storing R2 and R4 in the internal nodes as well. Also, the *push common rule upwards* heuristic is applied after the decision tree is built, while our *rule overlap reduction* technique is integrated with the decision tree construction algorithm.

4.2 Architecture Overview

Like the HyperCuts with the *push common rule upwards* heuristic enabled, our algorithm may reduce the memory consumption at the cost of increased search time, if the process to match the rules in the *internal rule list* of each tree node is placed in the same critical path of decision tree traversal. Any packet traversing the decision tree must perform (1) matching the rules in the *internal rule list* of the current node and (2) branching to the child nodes, in sequence. The number of memory accesses along the critical path can be very large in the worst cases. Although the throughput can be boosted by using a deep pipeline, the large delay passing the packet classification engine requires the router to use a large buffer to store the payload of all packets being classified. Moreover, since the search in the rule list and the traversal in the decision tree have different structures, a heterogeneous pipeline is needed, which complicates the hardware design.

FPGAs provide massive parallelism and high-speed dual-port Block RAMs distributed across the device. We exploit these features and propose a highly parallel architecture with localized routing paths, as shown in Figure 3. The design is based on the following considerations.

1. Regardless of internal rule lists, the traversal of the decision tree can be pipelined. Thus we have a pipeline for traversing the decision tree, shown as light-color blocks in Figure 3. We call this pipeline *Tree Pipeline*.
2. Note that each tree node is attached to a list of rules, no matter whether it is an internal or a leaf node. Analogous to *internal rule list*, the rule list attached to a leaf node is called a *leaf-level rule list*. Search in the rule lists can be pipelined as well.
3. When a packet reaches an internal tree node, the search in the internal rule list can be initiated at the same time the branching decision is made, by placing the rule list in a separate pipeline. We call such a pipeline *Rule Pipeline*, shown in shaded blocks in Figure 3.
4. For the tree nodes mapped onto the same stage of the Tree Pipeline, their rule lists are mapped onto the same Rule Pipeline. Thus, there will be $H+1$ Rule Pipelines if the Tree Pipeline has H stages. One Rule Pipeline is dedicated for the internal rule list associated with the root node.

5. All Rule Pipelines have the same number of stages. The total number of clock cycles for a packet to pass the architecture is $H + listSize$, where $listSize$ is the number of stage in a Rule Pipeline.
6. Consider two neighboring stages of Tree Pipeline, denoted A and B, where Stage B follows Stage A. The Rule Pipeline attached to Stage A outputs the matching results one clock cycle earlier than the Rule Pipeline attached to Stage B. Instead of waiting for all matching results from all Rule Pipelines and directing them to a single priority resolver, we exploit the one clock cycle gap between two neighboring Tree Pipeline stages, to perform the partial priority resolving for the two previous matching results.
7. The Block RAMs in FPGAs are dual-port in nature. Both Tree Pipeline and Rule Pipelines can exploit this feature to process two packets per clock cycle. In other words, by duplicating the pipeline structure (i.e. logic), the throughput is doubled, while the memories are shared by the dual pipelines.

As shown in Figure 3, all routing paths between blocks are localized. This can result in a high clock frequency even when the on-chip resources are heavily utilized. The FPGA implementation of our architecture is detailed in Section 5.

4.3 Algorithms

4.3.1 Decision Tree Construction

To fit tens of thousands of unique rules in the on-chip memory of a single FPGA, we must reduce the memory requirement of the decision tree. In Section 4.1, we presented two optimization techniques, *rule overlap reduction* and *precise range cutting*, for the state-of-the-art decision-tree-based packet classification algorithm, i.e. HyperCuts. This section describes how to integrate the two optimization techniques into the decision tree construction algorithm.

Starting from the root node with the full rule set, we recursively cut the tree nodes until the number of rule in all the leaf nodes is smaller than a parameter named $listSize$. At each node, we need to figure out the set of fields to cut and the number of cuts performed on each field. We restrict the maximum number of cuts at each node to be 64. In other words, an internal node can have 2, 4, 8, 16, 32 or 64 children. For the port fields, we need to determine the precise cut points instead of the number of cuts. Since more bits are needed to store the cut points than to store the number of cuts, we restrict the number of cuts on port fields to be at most 2. For example, we can have 2 cuts on SA, 4 cuts on DA, 2 cuts on SP, and 2 cuts on DP. We do not cut on the protocol field since the first 4 fields are normally enough to distinguish different rules in real life [10].

We use the same criteria as in HiCuts [5] and HyperCuts [18] to determine the set of fields to cut and the number of cuts performed on SA and DA fields. Our algorithm differs from HiCuts and HyperCuts in two aspects. First, when the port fields are selected to cut, we seek the cut point which results in the least rule duplication. Second, after the cutting method is determined, we pick the rules whose duplication counts are the largest among all the rules covered by the current node, and push them into the internal rule list of the current node, until the internal rule list becomes full. Algorithm 1 shows the complete algorithm for building the decision tree, where n denotes a tree node, f a packet header field, and r a rule. Figure 4 shows the decision tree constructed for the rule set given in Table 2.

Algorithm 1 Building the decision tree

```
1: Initialize the root node and push it into nodeList.
2: while nodeList ≠ null do
3:   n ← Pop(nodeList)
4:   if n.numRules < listSize then
5:     n is a leaf node. Continue.
6:   end if
7:   n.numCuts = 1
8:   while n.numCuts < 64 do
9:     f ← ChooseField(n)
10:    if f is SA or DA then
11:      numCuts[f] ← OptNumCuts(n, f)
12:      n.numCuts *= numCuts[f]
13:    else if f is SP or DP then
14:      cutPoint[f] ← OptCutPoint(n, f)
15:      n.numCuts *= 2
16:    end if
17:    Update the duplication counts of all r ∈ n.ruleSet:
       r.dupCount ← # of copies of r after cutting.
18:    while n.internalList.numRules < listSize do
19:      Find r_m which has the largest duplication count
         among the rules in n.ruleSet\n.internalList.
20:      Push r_m into n.internalList.
21:    end while
22:    if All child nodes contain less than listSize rules
       then
23:      Break.
24:    end if
25:  end while
26:  Push the child nodes into nodeList.
27: end while
```

4.3.2 Tree-to-Pipeline Mapping

The size of the memory in each pipeline stage must be determined before FPGA implementation. However, as shown in [1], when simply mapping each level of the decision tree onto a separate stage, the memory distribution across stages can vary widely. Allocating memory with the maximum size for each stage results in large memory wastage. Baboescu et al. [1] propose a Ring pipeline architecture which employs TCAMs to achieve balanced memory distribution at the cost of halving the throughput to one packet per two clock cycles, i.e. 0.5 PPC, due to its non-linear structure.

Our task is to map the decision tree onto a **linear** pipeline (i.e. Tree Pipeline in our architecture) to achieve balanced memory distribution over stages, while sustaining a throughput of one packet per clock cycle (which can be further improved to 2 PPC by employing dual-port RAMs). The memory distribution across stages should be balanced not only for the Tree Pipeline, but also for all the Rule Pipelines. Note that the number of words in each stage of a Rule Pipelines depends on the number of tree nodes rather than the number of words in the corresponding stage of Tree Pipeline, as shown in Figure 5. The challenge comes from the various number of words needed for tree nodes. As a result, the tree-to-pipeline mapping scheme requires not only balanced memory distribution, but also balanced node distribution across stages. Moreover, to maximize the memory

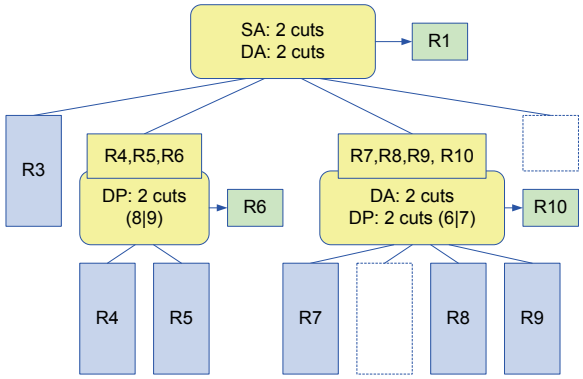

Figure 4: Building the decision tree for the rule set given in Table 2. (The values in parentheses represent the cut points on the port fields.)

utilization in each stage, the sum of the number of words of all nodes in a stage should approach some power of 2. Otherwise, for example, we need to allocate 2048 words for a stage consuming only 1025 words.

The above problem is a variant of bin packing problems, and can be proved to be NP-complete. We use a heuristic similar to our previous study of trie-based IP lookup [8], which allows the nodes on the same level of the tree to be mapped onto different stages. This provides more flexibility to map the tree nodes, and helps achieve a balanced memory and node distribution across the stages in a pipeline, as shown in Figure 5. Only one constraint must be followed:

Constraint 1. If node A is an ancestor of node B in the tree, then A must be mapped to a stage preceding the stage to which B is mapped.

We impose two bounds, namely B_M and B_N for the memory and node distribution, respectively. The values of the bounds are some power of 2. The criteria to set the bounds is to minimize the number of pipeline stages while achieving balanced distribution over stages. The complete tree-to-pipeline mapping algorithm is shown in Algorithm 2, where n denotes a tree node, H the number of stages, S_r the set of remaining nodes to be mapped onto stages, M_i the number of words of the ith stage, and N_i the number of nodes mapped onto the ith stage. We manage two lists, *ReadyList* and *NextReadyList*. The former stores the nodes that are available for filling the current stage, while the latter stores the nodes for filling the next stage. We start with mapping the nodes that are children of the root onto Stage 1. When filling a stage, the nodes in *ReadyList* are popped out and mapped onto the stage, in the decreasing order of their heights [1]. If a node is assigned to a stage, its children are pushed into the *NextReadyList*. When a stage is full or *ReadyList* becomes empty, we move on to the next stage. At that time, the *NextReadyList* is merged into *ReadyList*. By these means, *Constraint* 1 can be met. The complexity of this mapping algorithm is $O(N)$, where N the total number of tree nodes.

External SRAMs are usually needed to handle very large rule sets, while the number of external SRAMs is constrained by the number of IO pins in our architecture. By assigning large values of B_M and B_N for one or two specific stages, our mapping algorithm can be extended to allocate a large number of tree nodes onto few external SRAMs which consume controllable number of IO pins.

[1] *Height* of a tree node is defined as the maximum directed distance from it to a leaf node.

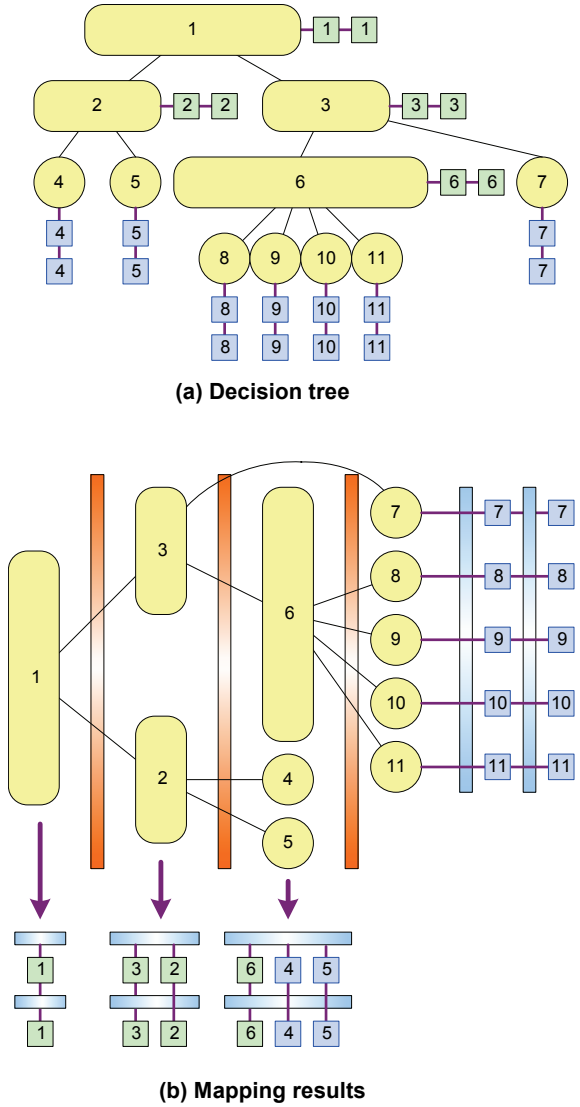

Figure 5: Mapping a decision tree onto pipeline stages ($H = 4$, $listSize = 2$)

5. IMPLEMENTATION

5.1 Pipeline for Decision Tree

As shown in Figure 4, different internal nodes in a decision tree may have different numbers of cuts, which can come from different fields. A simple solution is hard-wiring the connections, which however cannot update the tree structure on-the-fly [13]. We propose a circuit design, shown in Figure 6. We can update the memory content to change the number of bits for SA and DA to cut, and the cut enable bit for SP and DP to indicate whether to cut SP or DP.

Our tree-to-pipeline mapping algorithm allows two nodes on the same tree level to be mapped to different stages. We implement this feature by using a simple method. Each node stored in the local memory of a pipeline stage has one extra

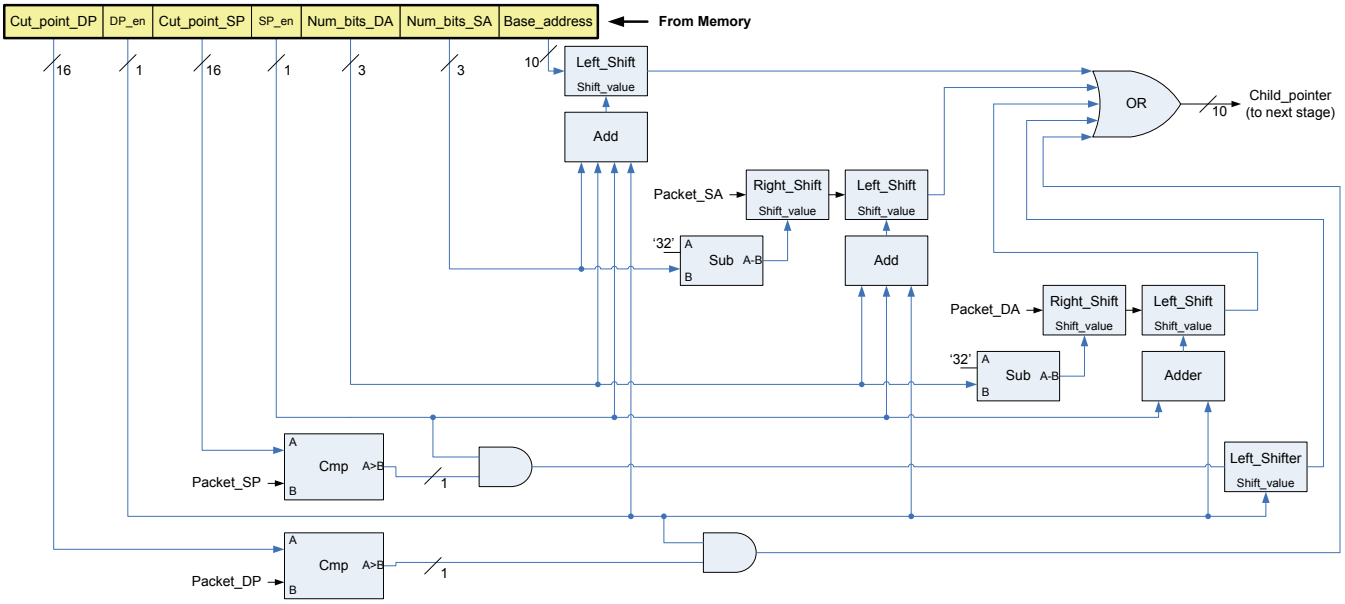

Figure 6: Implementation of updatable, varying number of branches at each node

Algorithm 2 Mapping the decision tree onto a pipeline

Input: The tree T.
Output: H stages with mapped nodes.

```
1: Initialization: ReadyList ← φ, NextReadyList ← φ, S_r ← T, H ← 0.
2: Push the children of the root into ReadyList.
3: while S_r ≠ φ do
4:    Sort the nodes in ReadyList in the decreasing order of their heights.
5:    while M_i < B_M AND N_i < B_N AND Readylist ≠ φ do
6:       Pop node from ReadyList.
7:       Map the popped node n_p onto Stage H.
8:       Push its children into NextReadyList.
9:       M_i ← M_i + size(n_p). Update S_r.
10:   end while
11:   H ← H + 1.
12:   Merge the NextReadyList to the ReadyList.
13: end while
```

field: the distance to the pipeline stage where the child node is stored. When a packet is passed through the pipeline, the distance value is decremented by 1 when it goes through a stage. When the distance value becomes 0, the child node's address is used to access the memory in that stage.

5.2 Pipeline for Rule Lists

When a packet accesses the memory in a Tree Pipeline stage, it will obtain the pointer to the rule list associated with the current tree node being accessed. The packet uses this pointer to access all stages of the Rule Pipeline attached to the current Tree Pipeline stage. Each rule is stored as one word in a Rule Pipeline stage, benefiting from the large word width provided by FPGA. Within a stage of the Rule Pipeline, the packet uses the pointer to retrieve one rule and compare its header fields to find a match. We implement different match types for different fields of a rule, as shown in Figure 7. When a match is found in the current Rule Pipeline stage, the packet will carry the corresponding action information with the rule priority along the Rule Pipeline until it finds another match where the matching rule has higher priority than the one the packet is carrying.

5.3 Rule Update

The dual-port memory in each stage enables only one write port to guarantee the data consistency. We update the memory in the pipeline by inserting *write bubbles* [2]. The new content of the memory is computed offline. When an update is initiated, a write bubble is inserted into the pipeline. Each write bubble is assigned an ID. There is one write bubble table in each stage, storing the update information associated with the write bubble ID. When a write bubble arrives at the stage prior to the stage to be updated, the write bubble uses its ID to look up the write bubble table and retrieves: (1) the memory address to be updated in the next stage, (2) the new content for that memory location, and (3) a write enable bit. If the write enable bit is set, the write bubble will use the new content to update the memory location in the next stage. Since the architecture is linear, all packets preceding or following the write bubble can perform their operations while the write bubble performs an update.

6. EXPERIMENTAL RESULTS

6.1 Algorithm Evaluation

We evaluated the effectiveness of our optimized decision-tree-based packet classification algorithm, by conducting experiments on 4 real-life rule sets of different sizes. Two performance metrics were measured: (1) average memory size per rule, and (2) tree height. The former metric represents the scalability of our algorithm, and the latter dictates the minimum number of stages needed in Tree Pipeline. The results are shown in Table 3. In these experiments, we set $listSize = 8$, which was optimal according to a series of

Table 3: Performance of algorithms for rule sets of various sizes

		Our algorithm		Original HyperCuts	
Rule set	# of rules	Memory (Bytes/rule)	Tree height	Memory (Bytes/rule)	Tree height
ACL_100	98	29.31	8	52.16	23
ACL_1k	916	26.62	11	122.04	20
ACL_5k	4415	29.54	12	314.88	29
ACL_10k	9603	27.46	9	1727.28	29

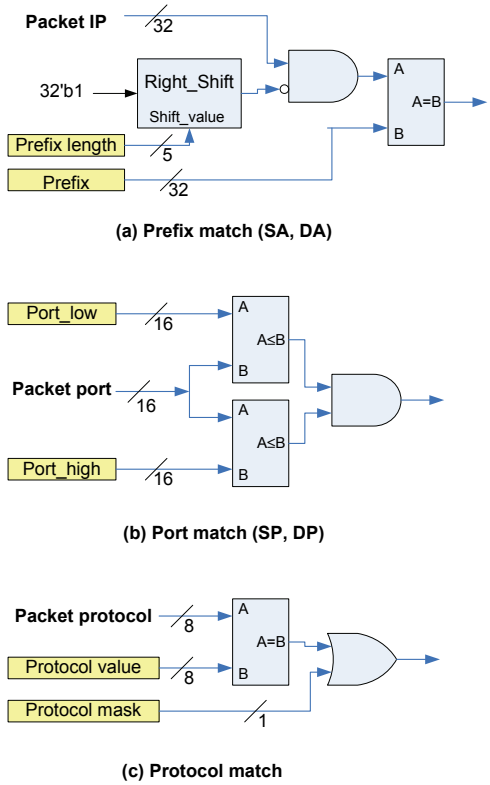

Figure 7: Implementation of rule matching

tests [2]. The detailed results are not presented, due to space limitation. According to Table 3, our algorithm kept the memory requirement to be linear with the number of rules, and thus achieved much better scalability than the original HyperCuts algorithm. Also, the height of the decision tree generated using our algorithm was much smaller than that of HyperCuts, indicating a smaller delay for a packet to pass through the engine.

According to Table 3, the Tree Pipeline needed at least 9 stages to map the decision tree for ACL_10K. We conducted a series of experiments to find the optimal values for the memory and the node distribution bounds, i.e. B_M and B_N. When $B_M \leq 512$ or $B_N \leq 128$, the Tree Pipeline needed more than 20 stages. When $B_M \geq 2048$ or $B_N \geq 512$, the memory and the node distribution over the stages was same as that using *static* mapping scheme which mapped each tree level onto a stage. Only when $B_M = 1024, B_N = 256$, both memory and node distribution were balanced, while the number of stages needed was increased slightly to 11, i.e. $H = 11$. As Figure 8 shows, our mapping scheme outperformed the static mapping scheme with respect to both memory and node distribution.

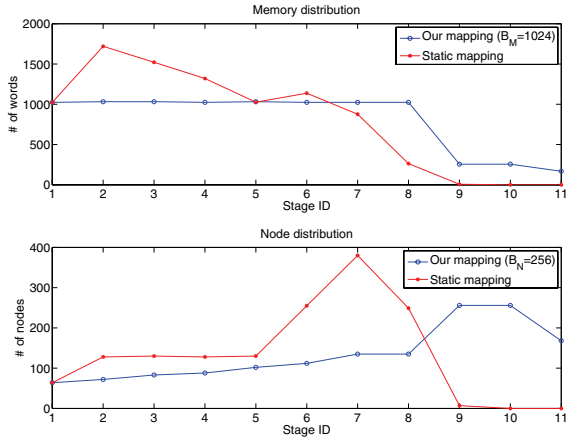

Figure 8: Distribution over Tree Pipeline stages for ACL_10K

6.2 FPGA Implementation Results

Based on the mapping results, we initialized the parameters of the architecture for FPGA implementation. According to the previous section, to include the largest rule set ACL_10K, the architecture needed $H = 11$ stages in Tree Pipeline, and 12 Rule Pipelines each of which had $listSize = 8$ stages. Each stage of Tree Pipeline needed $B_M = 1024$ words, each of which was 72 bits including base address of a node, cutting information, pointer to the rule list, and distance value. Each stage of Rule Pipeline needed $B_N = 256$ words each of which was 171 bits including all fields of a rule, priority and action information.

Table 4: Resource utilization

	Used	Available	Utilization
Number of Slices	10,307	30,720	33%
Number of bonded IOBs	223	960	23%
Number of Block RAMs	407	456	89%

We implemented our design, including write-bubble tables, in Verilog. We used Xilinx ISE 10.1 development tools. The target device was Xilinx Virtex-5 XC5VFX200T with -2 speed grade. Post place and route results showed that our design could achieve a clock frequency of 125.4 MHz. The resource utilization is shown in Table 4. Among the allocated memory, 612 Kbytes was consumed for storing the decision tree and all rule lists.

[2]We found that a larger $listSize$ resulted in lower memory requirement but deeper Rule Pipelines. The memory reduction became unremarkable when $listSize > 8$.

Table 5: Performance comparison

Approaches	# of rules	Total memory (Kbytes)	Throughput (Gbps)	Efficiency (Gbps/KB)
Our approach	9603	612	80.23	1358.9
Simplified HyperCuts [9]	10000	286	7.22 (3.41)	252.5
BV-TCAM [19]	222	16	10 (N/A)	138.8
2sBFCE [14]	4000	178	2.06 (1.88)	46.3
Memory-based DCFL [7]	128	221	24 (16)	13.9

Table 5 compares our design with the state-of-the-art FPGA-based packet classification engines. For fair comparison, the results of the compared work were scaled to Xilinx Virtex-5 platforms based on the maximum clock frequency [3]. The values in parentheses were the original data reported in those papers. Considering the time-space trade-off, we used a new performance metric, named *Efficiency*, which was defined as the throughput divided by the average memory size per rule. Our design outperformed the others with respect to throughput and efficiency. Note that our work is the only design to achieve more than 40 Gbps throughput.

7. CONCLUSIONS

This paper presented a novel decision-tree-based linear pipeline architecture on FPGAs for wire-speed multi-field packet classification. Several optimization techniques were proposed to reduce the memory requirement of the state-of-the-art decision-tree-based packet classification algorithm, so that 10K unique rules could fit in the on-chip memory of a single FPGA. Our architecture provided on-the-fly reconfiguration due to the linear memory-based architecture. To the best of our knowledge, our design was the first FPGA-based packet classification engine that achieved double wire speed while supporting 10K unique rules.

8. REFERENCES

[1] F. Baboescu, D. M. Tullsen, G. Rosu, and S. Singh. A tree based router search engine architecture with single port memories. In *Proc. ISCA*, pages 123–133, 2005.

[2] A. Basu and G. Narlikar. Fast incremental updates for pipelined forwarding engines. In *Proc. INFOCOM*, pages 64–74, 2003.

[3] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood. Fast packet classification using bloom filters. In *Proc. ANCS*, pages 61–70, 2006.

[4] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.

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

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

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

[3]The BV-TCAM paper [19] does not present the implementation result about the throughput. We use the predicted value given in [19].

[8] W. Jiang and V. K. Prasanna. Parallel IP lookup using multiple SRAM-based pipelines. In *Proc. IPDPS*, 2008.

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

[10] M. E. Kounavis, A. Kumar, R. Yavatkar, and H. Vin. Two stage packet classification using most specific filter matching and transport level sharing. *Comput. Netw.*, 51(18):4951–4978, 2007.

[11] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proc. SIGCOMM*, pages 203–214, 1998.

[12] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary. Algorithms for advanced packet classification with ternary CAMs. In *Proc. SIGCOMM*, pages 193–204, 2005.

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

[14] A. Nikitakis and I. Papaefstathiou. A memory-efficient FPGA-based classification engine. In *Proc. FCCM*, 2008.

[15] I. Papaefstathiou and V. Papaefstathiou. Memory-efficient 5D packet classification at 40 Gbps. In *Proc. INFOCOM*, pages 1370–1378, 2007.

[16] Renesas MARIE_Blade 18Mb Full Ternary CAM. http://www.renesas.com. February 2005.

[17] SAMSUNG 18Mb DDRII+ SRAM. http://www.samsung.com. August 2008.

[18] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. In *Proc. SIGCOMM*, pages 213–224, 2003.

[19] H. Song and J. W. Lockwood. Efficient packet classification for network intrusion detection using FPGA. In *Proc. FPGA*, pages 238–245, 2005.

[20] I. Sourdis. *Designs & Algorithms for Packet and Content Inspection*. PhD thesis, Delft University of Technology, 2007.

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

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

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

[24] F. Yu, R. H. Katz, and T. V. Lakshman. Efficient multimatch packet classification and lookup with TCAM. *IEEE Micro*, 25(1):50–59, 2005.