Ultra-High Throughput Low-Power Packet Classification

Alan Kennedy and Xiaojun Wang

***Abstract*— Packet classification is used by networking equipment to sort packets into flows by comparing their headers to a list of rules, with packets placed in the flow determined by the matched rule. A flow is used to decide a packet's priority and the manner in which it is processed. Packet classification is a difficult task due to the fact that all packets must be processed at wire speed and rulesets can contain tens of thousands of rules. The contribution of this paper is a hardware accelerator that can classify up to 433 million packets per second when using rulesets containing tens of thousands of rules with a peak power consumption of only 9.03 W when using a Stratix III field-programmable gate array (FPGA). The hardware accelerator uses a modified version of the HyperCuts packet classification algorithm, with a new pre-cutting process used to reduce the amount of memory needed to save the search structure for large rulesets so that it is small enough to fit in the on-chip memory of an FPGA. The modified algorithm also removes the need for floating point division to be performed when classifying a packet, allowing higher clock speeds and thus obtaining higher throughputs.**

***Index Terms*— Hardware accelerator, high throughput, low power, packet classification, parallel processing.**

I. Introduction

THE increasing growth in Internet usage has been aided by its ease of access through a wide range of devices such as desktops, notebooks, netbooks, tablets, and smartphones, putting a real strain on the networking equipment needed to inspect and process the resultant traffic. A survey carried out [1] showed how this ease of access has allowed Internet penetration to reach 32.7% of the world's population by December 2011, with the number of Internet users growing by 528% between 2000 and 2011. This survey also showed that the U.S. had over 108 million internet users in 2000 and that the number of worldwide users in 2011 was over 2.2 billion, which is important when considering that the total amount of energy used in the year 2000 by various networking devices in the U.S. equated to the yearly output of a typical nuclear reactor unit [2]. This means that the current amount of energy used by networking devices worldwide could exceed the yearly output of 21 typical nuclear reactor units. Power consumption should, therefore, be a key concern when designing any new networking equipment for the purpose of processing the ever-increasing amount of network traffic.

Network processors are key components used to process packets as they pass through a network, carrying out tasks such as packet fragmentation and reassembly, encryption, forwarding, and classification. The growing number of tasks that need to be carried out, along with the increase in line rates, have placed the network processor under increased pressure. Relieving this pressure through the addition of extra processing capacity is not easy due to factors such as silicon limitations and tight power budgets. Ramping up clock speeds to gain extra performance is difficult due to physical limitations in the silicon used to create these devices, while increasing the number of processing cores can cause difficulty when it comes to writing the software needed to control the network processors. Both these approaches also lead to large increases in power consumption due to the extra heat generated by increasing the clock speed and the extra transistors needed to increase the number of processing cores.

The use of hardware accelerators dedicated to the most computationally heavy tasks of a network processor can help to reduce power consumption while increasing processing capacity. This is because a hardware accelerator can be designed to have fewer transistors than that of the general-purpose processors used in multi-core network processors. Hardware accelerators can also process more data than a general-purpose processor while running at slower clock speeds as they are optimized to carry out specific tasks. A reduction in clock speed and number of transistors leads to large savings in power consumption.

The information presented in this paper centers around the design and implementation of an energy-efficient packet classification hardware accelerator (classifier) that can relieve a network processor's processing engines of the difficult and power hungry networking task of packet classification. Packet classification is difficult due to the fact that all packets entering a router must be processed at wire speed. The large number of services being provided by network providers makes this problem even more difficult as rulesets containing thousands of rules are needed.

Software approaches to packet classification use algorithms [3]–[11] that run on the processing engines of multi-core network processors. A network processor's flexibility reduces throughput, limiting packet classification to edge routers where line speeds are typically only a few gigabits per second. Analysis of popular packet classification algorithms in [12] showed that even the best performing algorithm in terms of

Manuscript received June 11, 2012; revised December 22, 2012; accepted January 8, 2013. Date of publication February 11, 2013; date of current version January 17, 2014. This work was supported in part by FP7 ECONET and the Irish Research Council for Science, Engineering and Technology (IRCSET).

A. Kennedy is with the School of Electronic Engineering, Dundalk Institute of Technology, Dundalk, Ireland (e-mail: alan.kennedy@dkit.ie).

X. Wang is with the School of Electronic Engineering, Dublin City University, Dublin 9, Ireland (e-mail: xiaojun.wang@dcu.ie).

Digital Object Identifier 10.1109/TVLSI.2013.2241798

throughput RFC [5] can only classify around 400 000 packets per second. This is when it is implemented in software and run on an reduced instruction set computing (RISC) processor similar to the type used as the processing cores in many of today's programmable network processors.

Most hardware approaches that can classify packets at core network line speeds, which can exceed 40 Gb/s, use power hungry ternary content addressable memory (TCAM). The Cypress Ayama 10 000 Network Search Engine [13], for example, uses up to 19.14 W when classifying 133 million packets per second.

The classifier presented here allows packet classification to be moved to the core of a network, thus improving security. It uses multiple packet classification engines working in parallel with a shared memory, allowing it to classify packets at speeds of up to 138.56 Gb/s, while using rulesets containing tens of thousands of rules. It implements a modified version of the HyperCuts [3] packet classification algorithm, which breaks a ruleset into groups, with each group containing a small number of rules that can be searched linearly. A decision tree is used to guide a packet based on its header values to the correct group to be searched.

The rest of this paper is organized as follows. Section II explains decision tree-based packet classification and gives a detailed explanation of the HyperCuts algorithm. This is done so that the changes made here to make the algorithm more suited to hardware acceleration can be better understood. These changes are explained in Section III. Section IV explains the architecture of the classifier. The performance results including memory usage, throughput, and power consumption are given in Section V. This section also compares the performance of the classifier against prior art. Section VI concludes this paper.

II. Decision Tree-Based Packet Classification

The fields of a packet's header most commonly used to perform packet classification are the 32 b source and destination IP addresses, the 16 b source and destination port numbers, and the 8 b protocol number. The simplest way to match these five fields to a rule is to linearly search through the rules one at a time, starting with the highest priority rule and ending with the lowest priority rule, until a match is found. This will result in an unacceptably large worst case processing time, making it difficult to classify packets at the speeds required for the core or even edge of a network. This worst case amount of processing time can be reduced by using the HyperCuts [3] packet classification algorithm. It is a decision tree-based algorithm that builds a search structure that allows incremental updates to a ruleset. Search structures that allow incremental updates do not have to be rebuilt each time a ruleset has a rule added or deleted. HyperCuts works by breaking a ruleset into groups, with each group containing a small number of rules suitable for a linear search. Each group of rules is stored in a leaf node of a decision tree, with a packet finding the leaf node that contains the matching rule by traversing the decision tree using values from its header to guide it.

TABLE I
Sample Ruleset Containing Nine Rules

RuleID	S. IP	D. IP	S. Port	D. Port	Protocol	Action
R_1	0000	0101	30-80	0-65535	UDP	ACT_1
R_2	111*	1***	0-2000	10-10	UDP	ACT_2
R_3	1***	101*	60-80	0-65535	TCP	ACT_3
R_4	101*	0***	0-65535	960-990	TCP	ACT_4
R_5	00**	101*	0-65535	800-811	TCP	ACT_5
R_6	000*	0111	30-80	0-65535	UDP	ACT_6
R_7	000*	0110	30-80	0-65535	UDP	ACT_7

HyperCuts creates this decision tree by taking a geometric view of a ruleset, with each rule considered to be a hypercube in hyperspace. The boundaries of each hypercube are defined by the ranges of the rule it represents. The algorithm cuts into this hyperspace by performing cuts to the fields used to define it. Each cut will create subregions, with each subregion containing the rules whose hypercube overlap. The information regarding the first set of cuts used to divide the hyperspace is stored in the root node of a decision tree. This information includes the number of cuts that are to be performed to each field and the memory location of each of the resulting subregions. These subregions are known as the root's child nodes, with subregions that contain no rules known as empty nodes. Subregions whose number of rules do not exceed a user-defined limit are known as leaf nodes. This user-defined limit is known as the $binth$ value. Each leaf node stores one rule group that can be searched linearly. A subregion that contains more rules than is allowed by the $binth$ value is known as an internal node and the space it occupies must be further cut up into smaller subregions. An internal node records the number of cuts that must be performed to the fields used to split the space it occupies into smaller subregions. It also stores the memory locations of the resulting subregions that are the internal node's child nodes. An internal node can also have empty, leaf, and internal nodes. The division of the hyperspace into ever-smaller subregions ends when the number of rules in all subregions do not exceed the $binth$ value.

A. Building a Decision Tree

This section describes, step by step, how to build a decision tree from the ruleset shown in Table I. The source and destination IP addresses have been reduced from 32 to 4 b to aid the explanation. The first step in building the decision tree is to decide a value for $spfac$ and $binth$. The $spfac$ value is used to control how many cuts can be made to each root or internal node and is used to control memory usage. In this example, $spfac$ will be three and $binth$ will be two. The next step involves deciding which dimensions should be used by the root node to cut the hyperspace. This is done by first calculating the number of distinct range specifications for each field, with the source and destination IP addresses both having six, the source and destination ports both having four, and the protocol number having two, giving a mean of 4.4. The fields whose distinct number of range specifications is greater than or equal to the mean number of distinct range specifications are then considered for cutting. The source and destination IP addresses shall, therefore, be considered for cutting as they

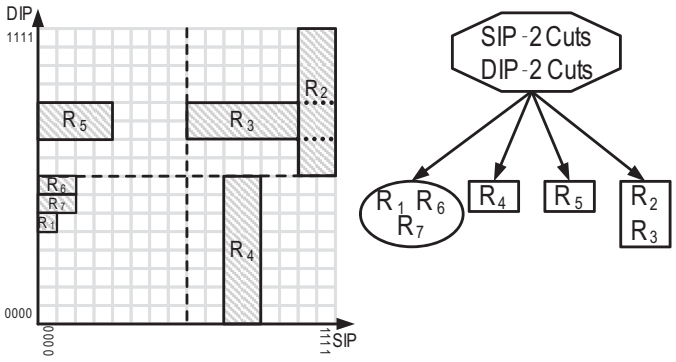

Fig. 1. Cuts performed to a decision tree's root node.

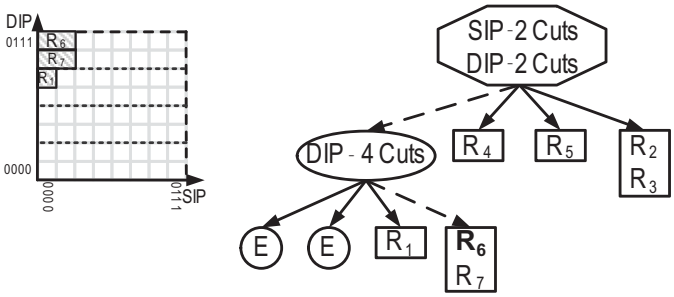

Fig. 2. Cuts performed to a decision tree's internal node and the path traversed.

both have a distinct number of range specifications greater than the mean. The maximum number of cuts that can be performed to the root node is calculated next using

$$\text{max cuts to node } i \leq \text{spfac}^{*}\text{sqrt(number of rules at } i) \quad (1)$$

where i is the internal or root node being cut. Small $spfac$ values will result in fewer cuts to nodes, creating a deep and narrow decision tree, while larger values for $spfac$ will allow more cuts, resulting in a wide but shallow decision tree. A deep and narrow decision tree will generally require less memory but will have a longer worst case processing time when matching a packet to a rule as more internal nodes will need to be traversed.

The maximum number of cuts that can be made to the root node in this example is 7.9. The number of cuts that can be made to a node is limited to be a power of 2 for ease of implementation, which means that a maximum of 4 cuts can be performed.

The next step involves trying all combinations of cuts between the chosen dimensions that are less than or equal to 4, with the maximum number of rules stored in a child node for each combination of cuts recorded. The combinations of cuts that can be made to the source and destination IP addresses are [0, 2], [0, 4], [2, 0], [2, 2], and [4, 0]. The combination that results in the smallest maximum number of rules stored in a child node is to cut both the source and destination IP addresses in two. Fig. 1 shows the decision tree after performing these cuts. It also shows a geometric representation of the source and destination IP addresses, showing the cuts made to the root node (represented by an octagon in the decision tree). It can be seen that these cuts create four subregions. Three of these subregions conform to the $binth$ value as they contain two or less rules. This means that they are leaf nodes (represented by rectangles in the decision tree). The fourth subregion contains more rules than the $binth$ value allows. This means that it is an internal node (represented by an oval in the decision tree) that must be cut further.

The first step that must be carried out when cutting the internal node is to decide which dimensions should be considered for cutting. This is done by calculating the number of distinct range specifications for each field using the rules contained within the subregion. The source IP address now has two distinct range specifications, the destination IP address has three, and the source port, destination port, and protocol number all have one, giving a mean of 1.6. The source and destination IP addresses are again considered for cutting as they both have a distinct number of range specifications greater than the mean. Equation (1) is used again to calculate the maximum number of cuts that can be performed to the internal node, which is 4 in this case. The combination of cuts that can be made to the source and destination IP addresses are the same as the combinations tried when cutting the root node. This time the combination that results in the smallest maximum number of rules stored in a child node is to perform four cuts to the destination IP address. This results in four subregions, with all subregions conforming to the $binth$ value, having two or less rules, which means that no more cutting needs to take place.

Fig. 2 shows the finished decision tree and the cuts performed to the destination IP address when cutting the internal node. It can be seen that two of the subregions contain no rules which means that they are empty nodes (represented by circles in the decision tree). The remaining two subregions are stored as leaf nodes. A packet with a header value [0001, 0111, 50, 80, UDP] would traverse the decision tree to find a matching rule in the following manner, with Fig. 2 showing the path traversed. The root node is first looked at and it can be seen that it specifies that two cuts must be performed to both the source and destination IP addresses. This is done by examining the most significant bit (MSB) of each header field. Only 1 b needs to be examined for each field, as each field only has two cuts, which can be represented by 1 b. The MSB for each field in this case is [0001, 0111]. These bits are concatenated to form the index 00, which represents the child node that must be traversed to. This child node is an internal node, meaning that more cuts need to be performed to the packet header in order to find the appropriate leaf node to search. The internal node is split by performing four cuts to the destination IP address. The next two MSBs in the destination IP address of the packet header must, therefore, be examined as 2 b are needed to represent the four possible cuts. The value of these bits is [0111], giving the index 11, which represents the child to be traversed to. This child is a leaf node, which is searched linearly by comparing each of the rules to the packet header one by one until a match is found. This will return rule R_6 as the matching rule in this example.

B. Heuristics Used to Reduce Memory Usage

The HyperCuts packet classification algorithm uses different heuristics to reduce the amount of memory needed to save a decision tree and the number of memory accesses required to match a rule. This section gives a brief description of these heuristics. One of these heuristics is called node merging and it is used to avoid the duplicated storage of identical nodes. Node merging is carried out by first searching the decision tree for leaf nodes that contain the same list of rules. The pointers to these nodes (stored in root and internal nodes) are then modified so that they point to just one of these leaf nodes, meaning that multiple copies do not need to be stored.

A second heuristic called rule overlap is used to avoid the storage of rules in leaf nodes that can never be matched. A rule can never be matched and is, therefore, removed from a leaf node if the hypercube of a rule with a higher priority completely covers the space it occupies within the leaf node's subregion.

A third heuristic used to avoid the duplicated storage of rules is called pushing common rule subset upward. This heuristic stores rules at an internal or root node that would otherwise need to be stored in all of the internal or root node's subregions.

The final heuristic is called region compaction and it is used to aid in the more efficient cutting of the hyperspace. Each node in a decision tree will cover a specific region of the hyperspace. The rules associated with a node may, however, cover a smaller region. Region compaction shrinks the area covered by a node so that it only covers the minimum amount of hyperspace that will cover all rules linked with the node. This means that a smaller region will need to be cut when dividing the hyperspace occupied by a node into subregions. This could result in fewer cuts, thus reducing memory consumption.

III. Algorithmic Modifications

The HyperCuts algorithm works well when implemented in software. It is not, however, optimized for implementation with dedicated hardware. This section explains the modifications made to the cutting scheme, region compaction heuristic, and rule storage method in order to make the algorithm better suited to hardware acceleration. The pushing common rule subset upward heuristic is not used as it was found during testing of rulesets to make only a fractional reduction in memory usage. It also results in a more complicated search structure that would slow down the classifier as it would have to be able to search root, internal and leaf nodes for matching rules. Pushing common rules upwards can also add extra memory accesses when classifying a packet. This is because a leaf node might still need to be searched even if a matching rule is found at an internal or root node. This is because a leaf node might contain another matching rule with a higher priority. Such a case would mean that the search of the rules at internal or root nodes was unnecessary.

A. Cutting Scheme

The cutting scheme was modified to improve throughput by making the decision tree as shallow as possible so as to reduce the number of memory accesses needed to classify a packet. The new cutting scheme requires the following three values to be specified before building of the decision tree can begin.

1) Number of cuts to be made to the root node.
2) Maximum number of cuts that can be made to an internal node.
3) Maximum number of rules that a leaf node can store.

The cutting scheme performs the majority of cuts to the root node because this will result in a shallow decision tree with the leaf nodes located closer to the root. The number of cuts that can be performed to an internal node is limited to only a few cuts to prevent the decision tree from using too much memory. It also means that the information needed to traverse an internal node can be placed in a single memory word, allowing them to be traversed in a single clock cycle.

The algorithm begins by first performing the required number of cuts to the root node. The number of cuts must be 2^n, where n can be any whole number in the range 1–18, limiting the maximum number of cuts that can be performed to 262 144. This is due to limitations on the amount of memory available to save the search structure. The algorithm uses the same method employed by HyperCuts to select the fields that should be considered for cutting. It only considers fields whose number of distinct range specifications is greater than or equal to the mean number for all fields. All combinations of cuts between the chosen fields that equal the 2^n limit are tried on the root node. The maximum number of rules stored in a child node is recorded for each combination of cuts, with the combination that results in the smallest number chosen.

The algorithm searches through all child nodes created from cutting the root node, with more cuts performed to the nodes whose number of rules exceed the maximum specified limit. The number of cuts that can be performed to the internal nodes is 2^m, where m can be any whole number between 1 and 4. The number of cuts that can be performed to an internal node has been capped at 16 so that all the information needed to traverse an internal node can fit in a single memory word, allowing them to be traversed in a single clock cycle.

Limiting the number of cuts also prevents excess memory usage and reduces the amount of time required to build the decision tree. The cutting of an internal node differs from the cutting of a root node in that all combinations of cuts are tried between the dimensions chosen for cutting that are less than or equal to the maximum limit. All combinations of cuts that are less than or equal to the maximum limit can be tried because there are only a few valid combinations that can be tried quickly. Cutting is complete when the number of rules in all subregions does not exceed the maximum specified limit.

1) Region Compaction: The region compaction scheme introduced in the HyperCuts algorithm is modified because it requires floating point division to be carried out when a packet traverses the decision tree. It also requires the minimum and maximum values of the area covered by all fields to be stored at a decision tree's internal and root nodes so that it

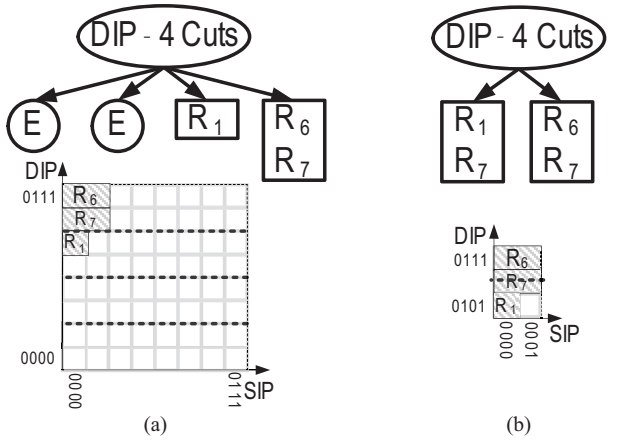

Fig. 3. Region division with and without region compaction. (a) No Region compaction. (b) Region compaction.

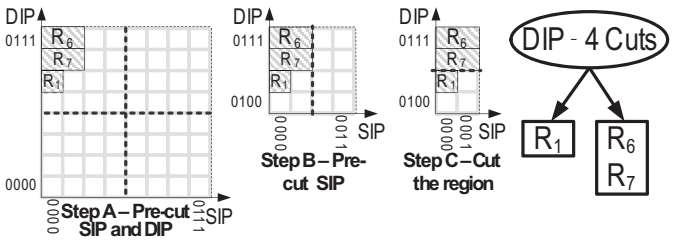

Fig. 4. Compacting of a region through pre-cutting.

is possible to calculate the child node to be traversed to. An alternative scheme is introduced here that uses pre-cutting to compact the region covered by a node more intelligently so that floating point division is not required when traversing the decision tree. Another advantage of pre-cutting is that it does not need to store the minimum and maximum values for each field, reducing memory consumption further.

A detailed explanation of the region compaction heuristic used by HyperCuts is now given so that the modifications made here can be better understood. Fig. 3(a) shows how the internal node from the decision tree shown in Fig. 2 was divided by performing four cuts to the length of the destination IP address covered by the internal node, with all resulting subregions containing two or less rules. This method of dividing the region allows for a simple scheme to be used when deciding which subregion a packet should traverse to, with only two pieces of information required for each field. This information includes the number of cuts that need to be performed to each field of a packet header and the bits in these fields where the cuts need to be performed. A packet with a destination IP address of 0111 will use its second and third MSBs to represent the index of the subregion that must be selected. The MSB is not needed as it was already used to form part of the index for the root node which cut the destination IP address in two. These 2 b represent the four possible subregions that could be selected.

Performing four cuts to the destination IP address is wasteful in this example as the rules that must be divided only span a small length of this section. The region compaction heuristic used by HyperCuts overcomes this problem as illustrated in Fig. 3(b) by only cutting the area covered by the rules and not the full region. Fewer cuts may, therefore, be needed to divide the region in a way that results in all subregions containing two or less rules. In this example, region compaction halves the number of cuts that are needed to divide the region. The use of region compaction requires three pieces of information to be stored for each field in order to calculate the subregion that must be traversed to. This information includes the minimum and maximum limits of the compacted region for a given field ($F_{\min}$ and $F_{\max}$) and the number of cuts (nc) that must be performed to this field in a packet's header (F_{header}). Equations (2) and (3) show how the index for each field is calculated, as follows:

$$((F_{\max} - F_{\min}) + 1)/nc = d \tag{2}$$

$$\lfloor (F_{\text{header}} - F_{\min})/d \rfloor = \text{index}. \tag{3}$$

A packet with a destination IP address of 0111 will have its index calculated as follows: index $= \lfloor (7-5)/1.5 \rfloor = 1$, where the denominator $d = ((7-5)+1)/2 = 1.5$. This index is the subregion that must be traversed to as only the destination IP address is used for cutting. Use of the region compaction heuristic in HyperCuts can lower memory consumption by reducing the number of subregions that need to be stored. It does not, however, work well when implemented in hardware as extra logic is needed to carry out the floating point division that is required when calculating the subregion that must be selected. The delay caused by the extra logic and additional clock cycles needed for floating point division will slow down the classifier, decreasing throughput and increasing power consumption.

2) Compacting of a Region Through Pre-Cutting: A new method for compacting the region to be cut at each internal or root node called pre-cutting is presented here. It uses the same methods employed by the scheme that uses no region compaction when calculating the subregion a packet should traverse to. This scheme only requires an internal or root node to store the number of cuts that must be performed to each field of a packet header and the bits in these fields where the cuts are to be performed. The simplicity of this scheme helps to improve throughput and decrease power consumption. The region that needs to be divided is compacted by recursively cutting all fields in two. This cutting of a specific field in two stops and will not be carried out if it results in rules being contained in more than one subregion. Each precut to a field used to divide the region will halve the number of subregions that need to be stored and the number of cuts that need to be performed to a packet header when selecting the subregion to traverse to. Each precut to a field also means that the bits which need to be inspected in that field of a packet's header are shifted to the right by one place.

Fig. 4 shows an example where pre-cutting is used to compact the area covered by the internal node from the decision tree shown in Fig. 2 so that it can be cut more efficiently. The process begins by performing precuts to the source and destination IP addresses as shown in step A, reducing the area that needs to be considered for cutting by 75%. Precuts can be performed to both fields as it results in

Prefix Length	Bit[34:7]	Bit[6:3]	Bit[2:1]	Bit[0]
32	32-Bit IP		00	0
	:			
29	32-Bit IP		11	0
28	28-Bit IP	011100		1
	:			
0	28-Bit IP	000000		1

Fig. 5. Encoding scheme used for source and destination IP addresses.

only one subregion that contains rules. In step B, only the source IP is precut as pre-cutting the destination IP address would result in more than one subregion that contains rules. Pre-cutting the source IP address in step B reduces the area that needs to be considered for cutting by another 50%. Finally, in step C no more precuts can be performed so the compacted region is cut in two, with none of the resulting subregions containing more than two rules. Pre-cutting gives the same effect as the region compaction heuristic used by HyperCuts in this example, with the number of subregions that need to be stored reduced from four to two when compared to the method where no region compaction is used.

The subregion that must be traversed to for a packet with a destination IP address of 0111 can be simply calculated by using its third MSB as an index. The MSB is not needed as it was already used to form part of the index for the root node which cut the destination IP address in two while the second MSB is not needed because of the pre-cutting that has just taken place. Only the third MSB is needed as an index as two cuts are performed to this field, meaning that 1 b can represent both possible subregions that could be selected.

B. Rule Storage

Modifications have also been made to the way that a rule is stored in a leaf node to reduce both memory consumption and the number of memory accesses needed to retrieve the information required to match a packet header to a rule. The first modification is to store the actual rule in the leaf node rather than a pointer to the rule. This was found during testing of rulesets to have only a small increase in memory consumption for some rulesets and a reduction for others as pointers to rules do not need to be stored. Storing the actual rule rather than a pointer to it allows for a large increase in throughput as data are presented to the classifier one clock cycle earlier.

A second modification is to reduce the number of bits required to store the source and destination IP addresses from 76 b down to 70 by using an encoding scheme. An IP address usually requires 32 b to store its address and 6 b to store its mask. The mask number is used to specify the number of MSBs of the address that must be an exact match to the corresponding bits in a packet header to record a match. The remaining LSBs are wildcard bits, meaning that the value of the corresponding bits in a packet header can have any value and still record a match. The encoding scheme stores the 32 b IP address and 6 b mask as a 35 b number. The least significant bit is used to indicate if more than 28 b of the IP address need to be matched exactly. If not set, 32 b are used to store the IP address, with the remaining 2 b indicating the number of bits that need to be matched. If the least significant bit is set, 28 b are used to store the IP address, with the remaining 6 b indicating the number of bits that need to be matched. The encoding scheme used by the classifier is shown in Fig. 5. This method of encoding the IP address and mask can easily be modified so that only 33 b are needed, with only a slight increase in the logic needed to decode the information.

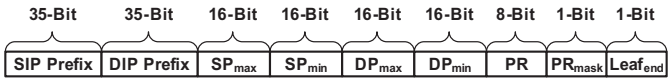

Fig. 6. Layout of information used to match a packet header to a rule (144 b).

Each rule will require 143 b to record the information needed to match it to a packet header, with the source and destination IP addresses each requiring 35 b. The source and destination port numbers both require 32 b, with each port number's minimum and maximum range values needing 16 b. A total of 9 b are required to store the information needed to match the protocol number, with 8 b used to store the protocol number and 1 b to store its mask. The mask only requires 1 b as the protocol number can only be an exact match or wildcard. Each rule also has a flag bit that is set if it is the last rule in a leaf node. The classifier will know that it has finished searching a leaf node if it comes across a rule with this bit set. Fig. 6 shows the memory layout of the information needed to match a packet header to a rule.

C. Cut Selection

The cutting information for each field consists of two pre-computed values. The first pre-computed value is called $Cuts$ and it is used to indicate how many cuts can be performed on a field. The number of cuts that can be performed on a field is limited to be a power of 2 for ease of implementation. An 8 b protocol number limited to 256 cuts, for example, can only have 0, 2, 4, 8, 16, 32, 64, 128, or 256 cuts performed to it. To save space a 4 b number for $Cuts$ can be used to represent the nine possible cut values. A maximum of 262 144 cuts can be performed to the source and destination IP addresses when dividing the hyperspace as explained in Section IV. This means that 5 b are required to store their $Cuts$ value as there are 19 possible cut values. The source and destination port numbers can have up to 65 536 cuts performed to them when dividing the hyperspace, which means that they also require 5 b to store their $Cuts$ value as they have 17 possible cut values.

The value of $Cuts$ is also the length of the bit-mask for a given field. This bit-mask will be ANDed with the appropriate bits of a field to extract the index for this field. Before the index of the child node is calculated, the $Cuts$ information is extended to form the bit-mask for each field. The second pre-computed value for each field is called $BPos$, and it is used to indicate the bits that the bit-mask should be ANDed with. In the calculation of a child node index, $BPos$ is the number of lower bits in a packet field that need to be removed by shifting the field right, before the operation of ANDing with the bit-mask can be performed. The protocol number, for example, will require three bits to store its $BPos$ value as it will need to

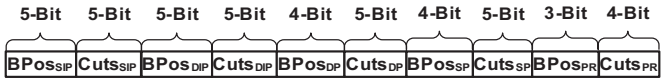

Fig. 7. Layout of root node cutting information (45 b).

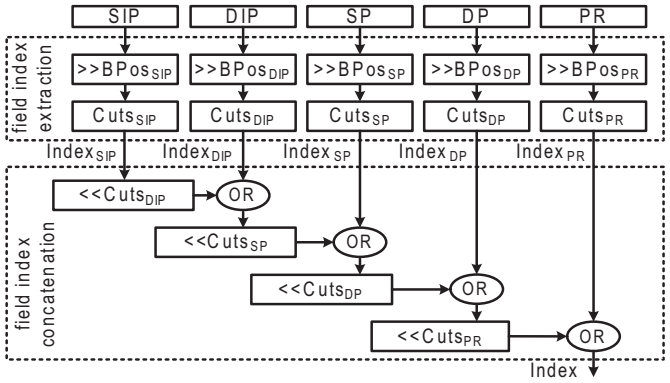

Fig. 8. Architecture of cut selection logic.

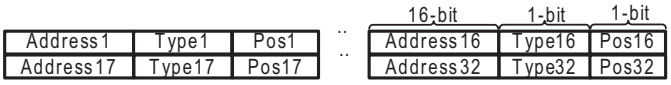

Fig. 9. Layout of root node pointers (up to 16 pointers stored in a memory word using a max of 288 b/word).

be shifted right by 0–7 places. The 32 b source and destination IP addresses will require 5 b to store their $BPos$ value and the 16 b source and destination port numbers will require 4 b to store their $BPos$ value. The layout of the root nodes cutting information is shown in Fig. 7. This cutting information is stored in a register separate from the field-programmable gate arrays (FPGAs) block RAM as it is used by every packet being classified. Having this information constantly available in a register will reduce the number of memory accesses needed to classify each packet by one.

The architecture of the cut selection logic is shown in Fig. 8. It can calculate the appropriate child node that a packet must traverse to in a single clock cycle as a result of the simplicity of the new region compaction and cutting schemes that have been presented here. These schemes can generate the appropriate child node index by performing simple shift and AND operations. The shifting of bits is carried out using multiplexers so that all shift operations can be performed in a single clock cycle. The child node index is generated in two stages. The first stage generates the subindex for each field, while the second stage concatenates these subindices together to form the final 18 b index of the child node to be selected. The subindex for a field is generated by first shifting it to the right by the number of bits specified by the field's $BPos$ value. The lower bits of this shifted value are then ANDed with the field's bit-mask to create its subindex. As mentioned, the bit-mask for a field is generated by extending its $Cuts$ value. The subindices are concatenated in the final stage to form the final index of the child node by left shifting the subindex of each field by the length of the subindex of the next field and then ORING them together. This is done until the index_{PR} is combined with the others as illustrated in Fig. 8.

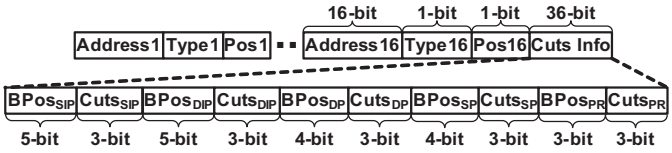

Fig. 10. Layout of internal node (324 b used if an internal node contains its max allowed limit of 16 pointers).

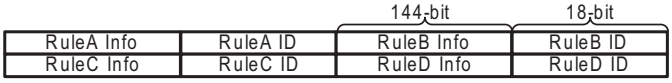

Fig. 11. Layout of leaf node (two rules stored in a memory word will use 324 b).

D. Memory Organization

This section explains how to store the root, internal and leaf nodes of the decision tree in memory. It also explains how the nodes are carefully arranged in memory after the decision tree has been built to ensure that there are no gaps of unused memory. The classifier uses the internal memory of an FPGA, exploiting the flexibility of this internal memory by using 324 b wide memory words. The root node requires 18 b to store each of its child node pointers, with each memory word used to store up to 16 pointers. This allows for a simple indexing system, with the MSBs of the child node index used to retrieve the memory word where its pointer is stored and the LSBs used to indicate its position in that memory word. The pointer uses 16 b to store the child node's address in memory, with a value of zero meaning that the child node is empty and no matching rule has been found. Another bit is used to indicate if the child is an internal or leaf node, while the final bit indicates the starting position of the node at its memory location if it is a leaf node. This bit is required because each memory word can hold two rules. Fig. 9 shows the layout of a root node's pointers. In this example, the root node has 32 child nodes, with the pointers to these nodes occupying two memory words.

Each internal node requires 36 b to store its cutting information. The only difference in the cutting information for an internal node and the root node is that the $Cuts$ value for each field only requires 3 b. This is because a maximum of 16 cuts can be performed to each field, which means that there are only five possible cut values for each field that can be represented using 3 b. An internal node will fit fully in one memory word as the cutting information and maximum of 16 pointers will require 324 b. Fig. 10 shows the layout of an internal node with the maximum allowed number of 16 child nodes. Fig. 11 shows the layout of a leaf node containing 4 rules, with these rules stored across two memory words. Each rule in a leaf node requires 162 b, of which 18 b are used to store the rule ID, 143 b to store the information needed to match a rule to a packet header, and 1 b to indicate if the rule is the last rule stored in a leaf node.

The nodes that form the decision tree are carefully rearranged after it has been built in order to obtain maximum storage efficiency and to ensure that no extra memory accesses

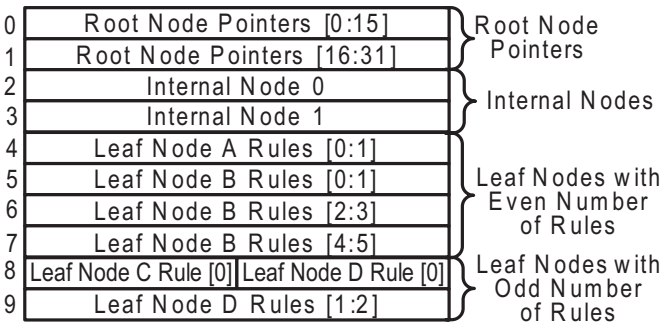

Fig. 12. Memory map showing how a decision tree is saved to memory.

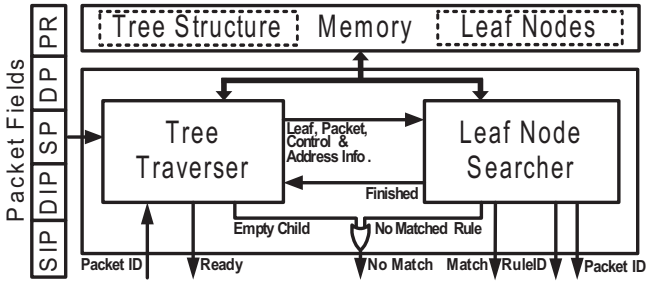

Fig. 13. Architecture used by the packet classification engines.

are added to the worst case required to classify a packet. The pointers of the root node's child nodes are stored first, followed by the internal nodes. Leaf nodes that contain an even number of rules are stored next and then the leaf nodes that contain an odd number of rules. A memory map showing how to save a decision tree with 32 cuts made to the root node, 2 internal nodes, and 4 leaf nodes containing 1–6 rules can be seen in Fig. 12.

IV. Packet Classification Engine

Fig. 13 shows the architecture of the packet classification engine which is built using two modules. The first module is a tree traverser that is used to traverse a decision tree using header information from the packet being classified. The decision tree is traversed until an empty node is reached, meaning that there is no matching rule, or a leaf node is reached. A leaf node being reached will result in the tree traverser passing the packet header and information about the leaf node reached to the second module known as the leaf node searcher. The leaf node searcher compares the packet header to the rules contained in the leaf node until either a matching rule is found or the end of the leaf node is reached with no rule matched. The leaf node searcher employs two comparator blocks that work in parallel. This allows two rules to be searched on each memory access, reducing lookup times.

Information on the decision tree's root node is stored in registers in the tree traverser, making it possible for the tree traverser to begin classifying a new packet while the previous packet is being compared with rules in a leaf node. This use of pipelining allows for a maximum throughput of one packet every two clock cycles if the decision tree is made up of only a root node and leaf nodes containing no more than two rules.

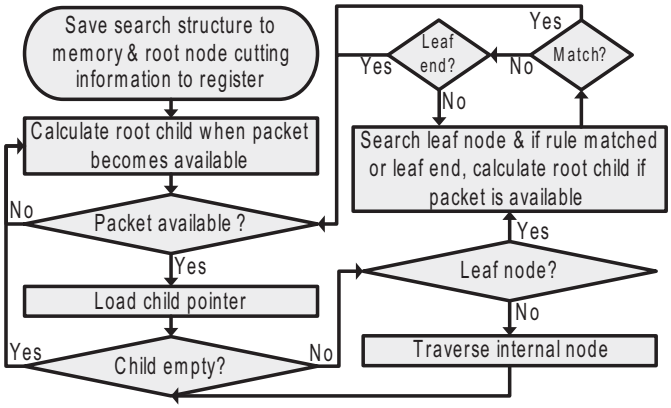

Fig. 14. Operation of a packet classification engine.

The flowchart shown in Fig. 14 explains the operation of the packet classification engine. The engine has been designed to traverse a root or internal node in one memory access. It can also search leaf nodes at a rate of two rules per memory access.

A. Architecture of the Classifier

The classifier has been implemented with multiple packet classification engines working in parallel using Stratix III and Cyclone III FPGAs. The maximum clock speed that an engine can achieve when implemented using an FPGA is much slower than the maximum clock speed of an FPGAs internal memory. This is due to logic delays in the components used by an engine such as the comparator blocks. It is, therefore, necessary to use multiple engines working in parallel so that the classifier can achieve maximum throughput. The use of multiple engines will help to ensure that the bandwidth of an FPGAs internal memory is better utilized.

Another reason for using multiple packet classification engines working in parallel is that it allows rulesets that contain many wildcard rules to be broken up into groups, with each engine used to search a group for a matching rule. Splitting rulesets that contain many wildcard rules into groups makes it easier to build shallow decision trees that have small leaf nodes, which helps to increase throughput and reduce memory usage. This is because the rules with wildcard source IP addresses can be kept in one group and the rules with wildcard destination IP addresses can be kept in another group. The group that contains the wildcard destination IP addresses is cut by performing the majority of the cuts to the source IP address, while the group that contains the wildcard source IP addresses is cut by performing the majority of the cuts to the destination IP address. The majority of the cuts are usually performed to the IP addresses because many of the ports can contain the common port range of 1024 to 65 535. The matching rule with the highest priority (rule with the lowest rule ID) will be chosen in the case where multiple engines return a matching rule. The search structure for each group can be saved to the same block of memory that is shared by the engines.

Both the Stratix III and Cyclone III implementations of the classifier use eight packet classification engines working in

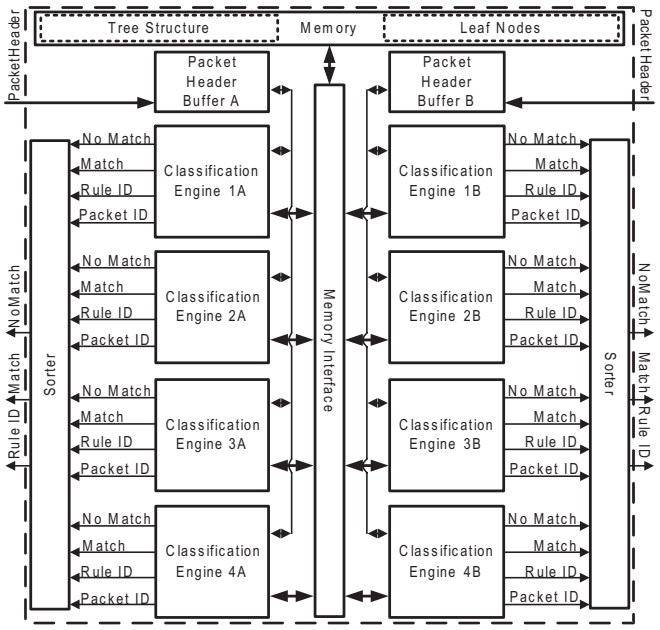

Fig. 15. Architecture of the classifier.

parallel. Fig. 15 shows the classifier's architecture, which takes advantage of the fact that the internal memory of an FPGA is dual port by placing two separate classifiers in parallel, sharing the same memory. Each classifier reads data from a separate data port and has its own packet buffer for storing the headers of incoming packets, four engines that work in parallel to maximize the bandwidth usage of a data port and a sorter logic block used to make sure that the classification results are outputted in the correct order.

The packet buffer stores the source and destination IP addresses, source and destination port numbers, and protocol number from the incoming packets. It works on a first come, first served basis, with packets being outputted from the buffer to the packet classification engines in the same order that they were inputted. The buffer also creates a packet ID for each header that is passed to the packet classification engine along with the packet header. The packet ID is used to make sure that the matching rule IDs are outputted by the classifier in the same order that the packet headers were inputted to the system.

The four engines belonging to a classifier run at the same clock speed, with the clock used by each engine 90° out of phase with the clock used by the previous engine. Memory runs at a speed equal to four times that of an engine, ensuring a simple memory interface, with each engine guaranteed access to memory on each of its clock cycles. The Stratix III implementation of this classifier has 46 080 memory words available to save the search structures required for classifying packets, while the Cyclone III implementation has 12 288 memory words available. The memory used is made up of a series of small memory blocks which are connected up so that they act as a continuous memory space. The memory ports of each memory block have their own enable signals. These enable signals are used to reduce power consumption by only activating the memory blocks that are being read from on a given clock cycle. This architecture also allows the splitting of a ruleset used to classify packets into groups of four or two in order to reduce the memory consumption and the worst case number of memory accesses needed to classify a packet for rulesets containing a large number of wildcard rules.

The sorter logic block is used to make sure that the matching rule IDs are outputted in the correct order and that the rule with the highest priority is selected when there are multiple rule matches in the case where rulesets are broken up into groups. The sorter logic block accepts the $Match$, $NoMatch$, $RuleID$, and $PacketID$ signals from each of the packet classification engines. It knows that an engine has finished classifying a particular packet when either the $Match$ or $NoMatch$ signals have been asserted. The first job the sorter logic block does is to make sure that the rule with the highest priority is selected between engines working in parallel to classify the same packet. This is done by picking the lowest rule ID between packets with the same packet ID. The sorter logic block registers the $Match$, $NoMatch$, and $RuleID$ signals for a classified packet to a chain of registers and multiplexers in series. The register selected will depend on the packet ID number. The $Match$, $NoMatch$, and $RuleID$ signals will be registered to the output register if they are next in the sequence of results to be outputted, and stored if not. All stored results are shifted toward the output register each time a result appears that is due to be outputted. This means that the classification results are outputted from the classifier in the same order that the packets were inputted.

B. Supporting IPv6 Packet Classification

The classifier can also perform IPv6 packet classification with some minor modifications. One modification would be to widen the memory words from 324 to 348 b, with a memory word storing one rule instead of two. This is because 348 b will be needed to match a packet header to a rule instead of the 162 currently used, assuming that the encoding scheme explained in Section III-B is used to store the 128 b IP addresses and their 8 b masks as 129 b numbers. This means that only one rule could be checked on each memory access. This could increase the average number of clock cycles needed to classify a packet. A second modification is that the tree traverser would use more logic resources as larger multiplexers would be needed to shift the IP addresses when creating the index used to select the node that a packet must traverse to. The logic needed for the leaf node searcher would also increase as a larger comparator block would be needed when comparing a packet header to rules in a leaf node.

The final modification is that the root and internal nodes would require an extra 4 b to store their cutting information. This is because 7 b would be needed to store the $BPos$ value for each IPv6 IP address instead of the 5 currently used. The 128 b IPv6 IP addresses might result in a deeper decision tree because smaller leaf nodes would be needed as only one rule could be compared on each clock cycle. The increased amount of logic and wider memory words will reduce the maximum achievable clock speed that the IPv6 classifier can obtain. This combined with the expected increase in the depth

TABLE II
FPGA RESOURCE UTILIZATION FOR CLASSIFIER

Device	Logic Element Usage	Memory Usage		f_{max}
Cyclone III	23 491/119 088 (19.7%)	M9Ks 432/432	(100%)	219 MHz
Stratix III	40 070/254 400 (15.7%)	M9Ks 852/864 M144Ks 48/48	(99.3%)	433 MHz

of the decision tree would slow down an IPv6 implementation of the classifier.

V. PERFORMANCE RESULTS

The classifier has been tested extensively by measuring its logic and memory usage, throughput in terms of Mpps, amount of memory it requires when storing the search structures needed to classify packets for access control list (ACL), firewall (FW), and Internet protocol chain (IPC) rulesets generated using ClassBench [14], worst case number of memory accesses needed to classify a packet, power consumption, and its performance when classifying packets using real life OC-48, OC-192, and OC-768 packet traces. These results have been benchmarked against state-of-the-art dedicated FPGA-based classifiers.

A. Hardware Implementation Parameters

The classifier was implemented in VHDL and targeted at:

1) a Cyclone EP3C120F484C7 FPGA, running at 1.2 V;
2) a Stratix EP3SE260H780C2 FPGA, running at 1.1 V.

Both of these devices are built on TSMC 65 nm process technology. The classifier was synthesized using Altera's Quartus II design software to obtain the maximum clock speeds and the logic and memory usage. The post place and route results are shown in Table II.

It can be seen from looking at the table that the memory of the classifier can achieve a maximum clock speed of 433 MHz when implemented using a Stratix III FPGA, giving it a maximum throughput of 433 Mpps. This is possible because each of its engines can classify a packet in two memory accesses and dual port memory is used, allowing two memory accesses to be made per clock cycle. A maximum throughput of 433 Mpps makes it the first packet classification hardware to the best of our knowledge that can process packets at line rates of up to 138.56 Gb/s. To meet these line speeds, the classifier needs to be able to process 433 Mpps as minimum-sized 40 byte packets can arrive back-to-back. The Stratix III implementation of the classifier can store the search structure required for rulesets containing in excess of 80 000 rules.

The Cyclone III implementation of this architecture also achieves a high throughput, with its memory obtaining a maximum clock speed of 219 MHz. This allows it to reach line speeds of up to 70 Gb/s or 219 Mpps. The Cyclone III implementation can store the search structure required for rulesets containing over 20 000 rules. These high levels of throughput makes it possible for the Stratix III and Cyclone III implementations to easily cope with core network line speeds when they are used to classify packets for rulesets containing tens of thousands of rules.

B. Memory Usage and Worst Case Number of Memory Accesses

The amount of memory required to save the ACL, FW, and IPC search structures built for the classifier using the modified HyperCuts algorithm can be seen in Table III. This table also shows the worst case number of memory accesses required to classify a packet when using these search structures. Other metrics given are the $binth$ value used when building a decision tree, maximum depth of each decision tree and the average number of memory accesses a packet header would need to reach and be matched to each rule stored in a decision tree. Results followed by an * show where a ruleset has been split into two groups in order to reduce the memory needed to save its search structure and to reduce the worst case number of memory accesses needed to classify a packet.

The results show that the classifier performs well in terms of memory usage and worst case number of memory accesses when using the ACL and IPC rulesets, achieving maximum throughput for all ACL and IPC rulesets tested, with two memory accesses needed at most to classify a packet. The classifier does not perform as well when classifying packets using the FW rulesets. This is because FW rulesets contain a large number of wildcard rules, with these wildcard rules covering a large area of the hyperspace, making it hard to divide the hyperspace into subregions that contain a small number of rules that are suitable for a linear search. This means that large leaf nodes need to be used, which slows down the classifier as longer linear searches need to be performed. It also means that a large amount of memory is used as there will be replicated storage of rules with the wildcard rules appearing in many leaf nodes.

The classifier, for example, requires 53 memory accesses at worst to classify a packet when using the search structure built for the FW ruleset with 23 087 rules. This will severely reduce the throughput of the classifier from its maximum of 433 Mpps to a worst case of 16.34 Mpps. The classifier will also require 14 747 832 b when storing this search structure. This problem can be overcome as explained in Section IV by splitting the ruleset into groups. Rules where wildcard ranges occur in the same fields can be grouped together, with these fields not used for cutting, where possible. This makes it easier to divide a ruleset into subregions that contain a small number of rules and reduces the replicated storage of rules. The throughput of the FW ruleset with 23 087 rules can be increased by splitting this ruleset into two groups, with two packet classification engines used to classify each packet. Splitting the ruleset will mean that the classifier will only require four memory accesses at worst to classify a packet, increasing the worst case throughput to 216.5 Mpps. The amount of memory needed to store the search structure is also reduced down to 4 736 232 b.

TABLE III
MEMORY USAGE IN BITS, BINTH (B), MAX TREE DEPTH (D), WORST CASE NUMBER OF MEMORY ACCESSES NEEDED TO CLASSIFY A PACKET (M), AVERAGE NUMBER OF MEMORY ACCESSES NEEDED TO CLASSIFY A PACKET (A)

Ruleset and Number of Rules	Stratix III					Cyclone III				
	Memory	B	D	M	A	Memory	B	D	M	A
ACL 5000	1 473 876	2	1	2	2	1 473 876	2	1	2	2
ACL 10 000	2 283 876	2	1	2	2	2 283 876	2	1	2	2
ACL 15000	3 093 876	2	1	2	2	3 093 876	2	1	2	2
ACL 20000	3 903 876	2	1	2	2	3 903 876	2	1	2	2
ACL 24920	4 700 916	2	1	2	2					
FW 5000	1 142 100	2	1	2	2	1 142 100	2	1	2	2
FW 10 000	7 968 456	4	3	4	2.4	3 933 360	76	2	39	11.3
	2 615 976*	**2***	**1***	**4***	**4***	**2 615 976***	**2***	**1***	**4***	**4***
FW 15000	11 708 388	4	3	4	2.4	**3 425 976***	**2***	**1***	**4***	**4***
	3 425 976*	**2***	**1***	**4***	**4***					
FW 20000	14 543 388	30	2	16	6.3	**3 567 240***	**4***	**2***	**6***	**2.3***
	4 235 976*	**2***	**1***	**4***	**4***					
FW 23087	14 747 832	104	2	53	14.8	**3 914 244***	**8***	**2***	**9***	**2.7***
	4 736 232*	**2***	**1***	**4***	**4***					
IPC 5000	1 473 876	2	1	2	2	1 473 876	2	1	2	2
IPC 10 000	2 283 876	2	1	2	2	2 283 876	2	1	2	2
IPC 15000	3 093 876	2	1	2	2	3 093 876	2	1	2	2
IPC 20000	3 903 876	2	1	2	2	3 903 876	2	1	2	2
IPC 24274	4 596 264	2	1	2	2					

The search structures built for the largest ACL, FW, and IPC rulesets tested only use about 30% of the Stratix III's memory. This means that the classifier can easily support dynamic updates as space can be left for leaf nodes to grow. The only difficulty that can occur is if a rule has to be added to an empty subregion. This is handled by creating a new leaf node and modifying the subregion's pointer so that it points to the new leaf node rather than to an empty node. A drawback to allowing the expansion of leaf nodes is that some leaf nodes may exceed the limit on the maximum number of rules that they should store, increasing linear search times and reducing throughput. This means that the decision tree may need to be rebuilt periodically so that the throughput does not suffer excessively.

C. Evaluation Against Prior Art

The area of packet classification is a well-studied field. Most research, however, has concentrated on the implementation of new packet classification algorithms tailored toward increased performance with software implementation in mind. These algorithms rarely consider the effects of power consumption, with their main aims instead being to increase the storage efficiency of rulesets while reducing the number of memory accesses needed to classify a packet. One algorithm that does tackle the issue of power consumption when implementing packet classification is EffiCuts [15]. It improves on the HiCuts [4] and HyperCuts [3] algorithms by lowering memory usage. This is done by reducing the replicated storage of rules in a decision tree, which means that a smaller amount of SRAM needs to be used when storing a ruleset's search structure, thus saving power. This power reduction, however, comes at a price, with throughput being reduced as EffiCuts requires more memory accesses to classify a packet.

Research into the increased throughput of packet classification through hardware acceleration with power consumption in mind is an increasingly important field of research as hardware accelerators have become essential when trying to meet core network line speeds. This is because line speeds are growing steadily due to advances in optical fiber technology and rulesets are expanding due to the increasing number of services that need to be performed. Packet classification is extremely difficult to implement at core network line speeds of over 40 Gb/s (125 Mpps in the worst case when 40 byte packets arrive back-to-back) using software approaches alone. Research in [12] shows that the most popular packet classification algorithms RFC [5], HiCuts [4], HyperCuts [3], TSS [9], and EGT-PC [8] can only classify packets at speeds of 400 937, 57 042, 32 242, 10 700, and 7491 p/s, respectively. This is when they are run on an RISC processor similar to the type used as the processing cores in many of today's programmable network processors. Most state-of-the-art classifiers aim to increase throughput through the use of TCAM [16]–[21]. The use of TCAM, however, makes these approaches a power hungry solution, even if power reduction techniques are used.

Classifiers targeted toward the use of FPGAs and SRAM instead of high power TCAM include the work presented in [22]–[25]. The classifier in [22] introduces a packet classification algorithm known as distributed crossproducting of field labels, with the authors claiming that their architecture could classify 100 Mpps while using rulesets containing up to 200 000 rules. These performance figures, however, assume that their logic intensive architecture could run at the maximum clock frequency of an FPGA.

TABLE IV
PERFORMANCE COMPARISON OF CLASSIFIERS

Approach	Device	No. of Rules	Speed (Mpps)	Memory Usage (bits)	Logic Usage (6-LUTs)	SF
Ultra-wide [23]	Stratix III	10 000	169	2 303 496	48 719/101 760	288
Pipelining [24]	Virtex 5	9603	250	5 013 504	41 228/122 880	82
HiCuts [25]	Stratix IV	10 000	74(100)	3 000 000	6 374/212 480	129
New classifier	Stratix III	10 000	433	2 283 876	16 028/101 760	37

The classifier from our previous work in [23] also implements a modified version of the HyperCuts packet classification algorithm that can classify packets at speeds of up to 169 Mpps while using rulesets containing tens of thousands of rules. It uses ultra-wide memory words (7704 b) that allow it to compare up to 48 rules from a leaf node in a single memory access. This allows it to use large leaf nodes that reduce the number of cuts needed to divide the hyperspace when building the decision tree, resulting in low memory usage. Large leaf nodes also allow the classifier to perform well when using FW rulesets as they do not need to be broken up into small groups of rules.

The paper in [24] implements a decision tree-based, dual pipeline architecture that can classify 250 Mpps when using rulesets containing up to 10 000 rules. It proposes optimization techniques to the HyperCuts algorithm such as a precise range cutting heuristic that reduces the replicated storage of rules. It also employs a tree to pipeline mapping scheme to improve memory utilization. Drawbacks with this design include poor storage efficiency for rulesets containing many wildcard rules, meaning that very large rulesets cannot be supported. Another drawback is that the architecture must be reconfigured if the depth of the decision tree constructed exceeds the worst case depth allowed by the implemented architecture.

A classifier that implements the decision tree-based packet classification algorithm HiCuts [4] is presented in [25], with HiCuts being the algorithm that HyperCuts is based on. Table IV compares the performance of the Stratix III implementation of the classifier presented here against the classifiers from [23]–[25]. The performances of [23] and [24] and the classifier presented here are compared when classifying packets using an ACL ruleset with 10 000 rules, generated using ClassBench. The authors of [25] tested their classifier using their own artificial ruleset containing 10 000 rules. The performance metrics examined are their speed or throughput in terms of worst case number of packets that they can classify per second, amount of memory needed to save the search structure required to classify packets, and their logic usage. A comparison of the power consumption is given in the next section.

The classifier presented here and in [23] were implemented on a Stratix EP3SE260 FPGA, while the approach employed in [24] used a Virtex XC5VFX200T FPGA. A direct comparison is fair as the performance of both FPGAs is similar due to the fact that both are manufactured using 65 nm process technology and both devices also have similar amount of internal memory and logic available. The classifier from [25] was implemented on a larger Stratix EP4SGX530 FPGA manufactured using 40 nm process technology, which Altera claim is 35% faster than other devices such as Virtex 5 FPGAS. The throughput of this classifier has, therefore, been reduced in the table, with the original data shown in brackets. The logic usage of the classifiers has been compared using six input lookup tables (LUT) as Virtex FPGAs give the logic utilization in slices, with each slice capable of implementing four 6 input LUT, while Stratix FPGAs give logic utilization in adaptive logic modules, with each capable of implementing one 6 input LUT. It can be seen that our classifier outperforms all others in the important metrics of throughput and amount of memory needed to save the search structures for the rulesets with 10 000 rules.

D. Throughput Versus Power Consumption

The two main causes of power consumption in an FPGA-based packet classifier are static power and dynamic power. Static power is consumed as a result of leakage currents and dynamic power is consumed because of switching activity. The classifier presented here consumes less power than other FPGA-based packet classifiers as it has been designed to use less dynamic power. The classifier's dynamic power is lowered by reducing the amount of logic that there is to switch and the number of times that this logic is switched. A new metric is introduced here called *switching factor (SF)* that measures a classifier's dynamic power efficiency. The *SF* is calculated by dividing the number of 6-LUTs a classifier uses to classify 1 Mpps by the number of packets classified per clock cycle. Fewer 6-LUTs will mean that there is less logic to switch, while more packets classified per clock cycle will mean that the logic is switched less often. A classifier will be more power efficient if it has a lower *SF* as there will be less switching activity.

Table IV compares the *SF* of the classifier presented here with the classifiers from [23]–[25]. It can be seen that the classifier presented here will consume less dynamic power as it has the lowest *SF*. The power consumption figures given in Fig. 16 also show this. The classifier presented here uses 37 6-LUTs to classify 1 Mpps and classifies 1 packet per clock cycle, compared with the classifier in [23] which uses 288 6-LUTs and classifies 1 packet per clock cycle, the classifier in [24] which uses 164 6-LUTs and classifies 2 packets per clock cycle, and the classifier in [25] which uses 86 6-LUTs and classifies 0.67 packets per clock cycle.

The classifier's power consumption was measured when it was used to classify packets using the rulesets shown in Table III. The power consumption was measured by carrying out post place and route simulations, with the Quartus 2 PowerPlay Power Analyzer Tool used to analyze VCD files

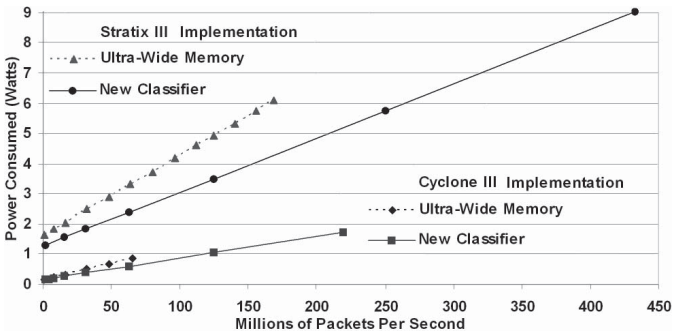

Fig. 16. Power consumed by the classifier.

generated by ModelSim. These results showed that the classifier's power consumption scales well as rulesets increase in size, with the power consumption being similar for each ruleset regardless of the size. This is because it will usually take two memory accesses to classify a packet for most rulesets, which means that the same amount of processing needs to be carried out on all packets. Two memory accesses, for example, are required to classify a packet for all ACL and IPC rulesets used. This means that the dynamic power consumption will be similar for all rulesets when classifying packets at a certain speed, while the static power consumption also remains the same. The classifier used slightly more power when classifying packets for the larger FW rulesets as they needed more memory accesses to classify a packet. The classifier's power consumption is shown in Fig. 16, with these results generated using the ACL ruleset containing 10 000 rules. These results are compared to the classifier that uses ultra-wide memory words [23]. A comparison cannot be made with the classifiers in [24] and [25] as no power consumption figures were given.

Fig. 16 shows that the classifier presented here performs far better than the classifier in [23] in terms of throughput and power consumption when both are implemented using the Cyclone III FPGA. The classifier presented here consumes 1.73 W when classifying packets at its maximum throughput of 219 Mpps (70.08 Gb/s), which means that it uses 24.69 mW/(Gb/s). This is an increase in throughput of 237% and a decrease in power consumption of 40% when compared to the classifier in [23]. The classifier in [23] consumes 0.85 W when classifying packets at its maximum throughput of 65 Mpps (20.8 Gb/s), which means that it uses 40.87 mW/(Gb/s).

Fig. 16 also shows that the classifier presented here performs better than the classifier in [23] when both are implemented using the Stratix III FPGA. The classifier presented here consumes 9.03 W when classifying packets at its maximum throughput of 433 Mpps (138.56 Gb/s), which means that it uses 65.17 mW/(Gb/s). This is an increase in throughput of 156% and a decrease in power consumption of 42% when compared to the classifier in [23]. The classifier in [23] consumes 6.08 W when classifying packets at its maximum throughput of 169 Mpps (54.08 Gb/s), which means that it uses 112.43 mW/(Gb/s).

The power consumption is much higher for the Stratix III implementation than the Cyclone III implementation because it is a much larger device, with greater amount of logic and memory resources available, leading to a larger amount of static power consumption. The larger amount of memory and logic used in the Stratix III implementations combined with the higher speeds will also cause more dynamic power consumption due to an increased amount of switching. The classifier presented here can achieve much higher throughput and lower power consumption than the classifier in [23] because it uses far less logic and narrower memory words. This allows it to achieve a higher clock speed as there are smaller routing delays, while the power savings come from the fact that there is far less logic being switched.

E. Performance on Real-Life Packet Traces

The classifier was tested extensively using synthetic OC-48, OC-192, and OC-768 packet traces that were created by aggregating Abilene, CENIC, and SCO4 backbone packet traces until peak line rates of 2.5, 10, and 40 Gb/s were reached. These traces were obtained from NLANR [26]. The OC-48 and OC-192 traces were looked at over a 6000-s period, while the OC-768 trace was looked at for a 2000-s period. The peak numbers of packets per second for the traces are 143 768 p/s for the OC-48 trace, 661 526 p/s for the OC-192 trace, and 3 302 488 p/s for the OC-768 trace. The timestamp from these traces was spliced to packet headers created using ClassBench for the ACL, FW, and IPC rulesets. The input buffer usage was then looked at when the classifier was used to classify packets using the ACL, FW, and IPC rulesets. The tests showed that the input buffer, which can store up to 256 packet headers, was kept clear at all times, with no build up of packets on both the Stratix III and Cyclone III implementations. This was the case even when the FW ruleset with 23 087 rules was used to classify packets using the OC-768 trace.

The maximum theoretical throughput of an OC-768 fiber optic link is 125 Mpps in the worst case when 40 byte packets arrive back-to-back. This means that the Stratix III and Cyclone III implementations of the classifier can cope with OC-768 line rates when their maximum clock frequency is reduced from their peaks of 433 and 219 MHz, respectively, down to 125 MHz. At this speed, the Stratix III implementation will only consume 3.5 W, while the Cyclone III implementation will only consume 1.05 W. These power consumption figures were taken from the test results shown in Fig. 16. We found in practice that the Stratix III and Cyclone III implementations can still cope with all rulesets and traces used here when their maximum clock frequency is reduced to 16 MHz, with minimal buffer usage, giving them peak power consumptions of only 1.56 and 0.26 W, respectively.

VI. Conclusion

Packet classification is usually limited to use by routers at the edge of a network where line speeds do not typically exceed a few gigabit per second. This paper introduced a new algorithm and packet classification hardware accelerator with enough processing power to allow packet classification to be implemented at the core of a network, thus improving security. It worked with rulesets containing tens of thousands

of rules at speeds of up to 138.56 Gb/s, allowing Internet service providers to perform a large plethora of tasks. The classifier consumed only 9.03 W when classifying packets at its maximum throughput of 433 Mpps. This is low when compared to other FPGA-based classifiers. The classifier ran a modified version of the HyperCuts algorithm that has been modified so that it is better suited to hardware implementation. These modifications included changing the cutting scheme so that the need for slow and logic intensive floating point division is removed when classifying a packet. This was done by replacing the region compaction scheme used by HyperCuts with a new scheme that uses pre-cutting.

REFERENCES

[1] *Usage and Population Statistics*. (2012, Jun.) [Online]. Available: http://www.internetworldstats.com/stats.htm

[2] M. Gupta and S. Singh, "Greening of the internet," in *Proc. ACM Special Interest Group Data Commun. Conf.*, Aug. 2003, pp. 19–26.

[3] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM Special Interest Group Data Commun. Conf.*, Aug. 2003, pp. 213–224.

[4] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," *IEEE Micro*, vol. 20, no. 1, pp. 34–41, Feb. 2000.

[5] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. ACM Special Interest Group Data Commun. Conf.*, Sep. 1999, pp. 147–160.

[6] F. Baboescu and G. Varghese, "Scalable packet classification," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, pp. 2–14, Feb. 2005.

[7] T. V. Lakshman and D. Stiliadis, "High-speed policy based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM Special Interest Group Data Commun. Conf.*, Sep. 1998, pp. 203–214.

[8] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?" in *Proc. IEEE Int. Conf. Comput. Commun.*, Apr. 2003, pp. 53–63.

[9] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search, " in *Proc. ACM Special Interest Group Data Commun. Conf.*, Sep. 1999, pp. 135–146.

[10] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Netw. Mag.*, vol. 15, no. 2, pp. 24–32, Mar. 2001.

[11] T. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. IEEE Int. Conf. Comput. Commun.*, Mar. 2000, pp. 1213–1222.

[12] A. Kennedy, D. Bermingham, X. Wang, and B. Liu, "Power analysis of packet classification on programmable network processors," in *Proc. IEEE Int. Conf. Signal Process. Commun.*, Nov. 2007, pp. 1231–1234.

[13] *Cypress Ayama 10000 Network Search Engine*. (2004, Mar. 10) [Online]. Available: http: //www.datasheetarchive.com/CYNSE10512-datasheet.html

[14] D. E. Taylor and J. S. Turner, "Classbench: A packet classification bench-mark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.

[15] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "Efficuts: Optimizing packet classification for memory and throughput." in *Proc. ACM Special Interest Group Data Commun. Conf.*, Aug. 2010, pp. 207–218.

[16] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *Proc. 11th IEEE Int. Conf. Netw. Protocols*, Nov. 2003, pp. 120–131.

[17] K. Zheng, H. Che, Z. Wang, B. Liu, and X. Zhang, "DPPC-RE: TCAM-based distributed parallel packet classification with range encoding," *IEEE Trans. Comput.*, vol. 55, no. 8, pp. 947–961, Aug. 2006.

[18] D. Pao, Y. Li, and P. Zhou, "Efficient packet classification using TCAMs," *Comput. Netw.*, vol. 50, no. 18, pp. 3523–3535, 2006.

[19] J. V. Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 4, pp. 560–571, May 2003.

[20] K. Zheng, H. Che, Z. Wang, and B. Liu, "TCAM-based distributed parallel packet classification algorithm with range-matching solution," in *Proc. IEEE Int. Conf. Comput. Commun.*, Mar. 2005, pp. 293–303.

[21] C. Meiners, A. Liu, and E. Torng, "Topological transformation approaches to TCAM-based packet classification," *IEEE/ACM Trans. Netw.*, vol. 19, no. 1, pp. 237–250, Feb. 2011.

[22] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducting of field labels," in *Proc. IEEE Int. Conf. Comput. Commun.*, Mar. 2005, pp. 269–280.

[23] A. Kennedy, Z. Liu, X. Wang, and B. Liu, "Multi-engine packet classification hardware accelerator," in *Proc. 19th Int. Conf. Comput. Commun. Netw.*, Aug. 2009, pp. 1–6.

[24] W. Jiang and V. K. Prasanna, "Large-scale wire-speed packet classification on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Feb. 2009, pp. 219–218.

[25] T. Zhang, Y. Wang, L. Zhang, and Y. Yang, "High throughput architecture for packet classification using FPGA," in *Proc. 5th ACM/IEEE Symp. Archit. Netw. Commun. Syst.*, Oct. 2009, pp. 62–63.

[26] National Laboratory for Applied Network Research. (2006). *Passive Measurement and Analysis Project*, San Diego, CA, USA [Online]. Available: http://www.caida.org/home/about/research/nlanr/

Alan Kennedy received the Diploma degree from the Dundalk Institute of Technology (DKIT), Dundalk, Ireland, in 2004, and the B.Eng. and Ph.D. degrees from Dublin City University, Dublin, Ireland, in 2006 and 2010, respectively, all in electronic engineering.

He is currently a Lecturer with the School of Electronic Engineering, DKIT. He was an FPGA Hardware Design Engineer with CréVinn and The Now Factory from 2009 to 2010. His current research interests include energy-efficient, high-throughput hardware accelerators for packet classification and deep packet inspection.

Xiaojun Wang received the B.Eng. degree in computer and communications and the M.Eng. degree in computer applications from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 1984 and 1987, and the Ph.D. degree in electronic engineering from Staffordshire University (then Staffordshire Polytechnic), Staffordshire, U.K., in 1993.

He was a Lecturer with BUPT from 1987 to 1989. He joined the School of Electronic Engineering, Dublin City University, Dublin, Ireland, as a Faculty member in 1992, where he is currently a Senior Lecturer. His current research interests include network security, energy-efficient networking, and network intrusion detection methods using traffic statistics, content, and behavioral analysis.