Speedy FPGA-Based Packet Classifiers with Low On-Chip Memory Requirements

Chih-Hsun Chou[1], **Fong Pong**[2], and **Nian-Feng Tzeng**[1]

[1]Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, Louisiana 70504, USA
vesaliusmac@gmail.com
tzeng@cacs.louisiana.edu

[2]Broadcom Corp.
2451 Mission College Boulevard
Santa Clara, California 95054, USA
fpong@broadcom.com

ABSTRACT

This article pursues speedy packet classification with low on-chip memory requirements realized on Xilinx Virtext-6 FPGA. Based on hashing round-down prefixes specified in filter rules (dubbed HaRP), our implemented classifier is demonstrated to exhibit an extremely low on-chip memory requirement (lowering the byte count per rule by a factor of 8.6 in comparison with its most recent counterpart [2]), taking only 50% of Virtex-6 on-chip memory to store every large rule dataset (with some 30K rules) examined. In addition, it achieves a higher throughput than any known FPGA implementation, reaching more than 200 MPPS (millions packet lookups per second) with 8 processing units and 8 memory banks in the HaRP pipeline to support the line rate over 130 Gbps under bi-directional traffic in the worst case with 40-byte packets. By reducing memory probes per lookup, enhanced HaRP can further boost the classification speed to 255 MPPS.

CATEGORIES AMD SUBJECT DESCRIPTORS

B.4.1 [**Input/Output and Data Communications**]: Data Communications Devices – *receivers*.

GENERAL TERMS

Performance, design, experimentation.

KEYWORDS

FPGAs, filter datasets, hash tables, memory efficiency, pipelined design, set-associativity.

1. INTRODUCTION

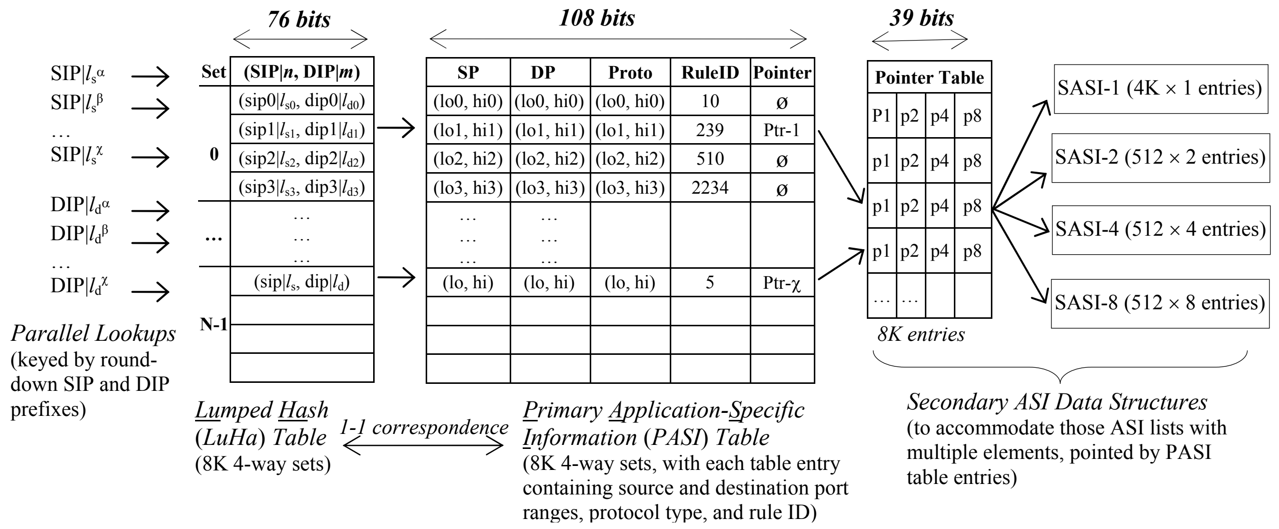
Packet classification is the technique for classifying the packets into different categories based on a set of pre-defined rules according to multiple fields in the packet header [13, 15]. It is an essential function for traffic management, access control, intrusion prevention, and many other network services. Typical packet header fields involved in a classifier are: source IP (SIP) address, destination IP (DIP) address, source port address range, destination port address range, protocol type, among others. A rule dataset in a core router may contain tens of thousands of rules ordered by priority, with each rule having its involved header fields specified or unspecified (as a wildcard). Upon receiving a packet, the classifier searches over its rule dataset for the matching rule with the highest priority, according to packet header field data. Classification tends to be time-consuming because of multiple fields involved and of large rule datasets commonly found in core routers, which operate at the line rates up to 100Gbps (e.g., Juniper's T160 Core Router [5]). The line rate of 100 Gbps requires classification to perform one lookup of a 40-byte packet in 6.4 *ns* (considering equal bi-directional traffic), under the worst case with the packet length of 40 bytes. As a result, classification can easily become the performance bottleneck of the Internet, calling for fast classifier design.

Hardware-based solutions are favorable for high-speed classification, utilizing ternary content addressable memory (TCAM), field programmable gate array (FPGA), or application specific integrated circuit (ASIC) in support of fast lookups. TCAM solutions are popular in the industry, and algorithms have been introduced to deal with their notorious problems related to range expansion [7] and incremental updates [14]. However, TCAMs are relatively expensive (due to low bit density), slow in operation, and inherently power hungry [1], as will be demonstrated by examples in Section 2.3. On the other hand, an ASIC solution usually involves considerable development time and effort and also suffers from limited flexibility, unable to easily adapt to changes to the classification procedure.

FPGA hardware has gained attention for high-performance classification lately, due to its potential in support of the 100Gbps line rate and its flexibility in accommodating various classification algorithms [2, 11]. For an FPGA-based classifier to achieve the desirable performance level, its employed classification algorithm ought to address the following design issues thoughtfully:

- memory efficiency: preferred for housing the rule dataset to get high performance, on-chip memory is rather limited, thereby favoring a memory-efficient classification algorithm so as to fit the whole rule dataset in on-chip memory,

FPGA'12, February 22–24, 2011, Monterey, California, USA.
Copyright 2012 ACM 978-1-4503-1155-7/12/02...$10.00.



76 bits
108 bits
39 bits
SIP|l_s^α
SIP|l_s^β
SIP|l_s^χ
DIP|l_d^α
DIP|l_d^β
DIP|l_d^χ
Set
(SIP|n, DIP|m)
(sip0|l_{s0}, dip0|l_{d0})
(sip1|l_{s1}, dip1|l_{d1})
(sip2|l_{s2}, dip2|l_{d2})
(sip3|l_{s3}, dip3|l_{d3})
(sip|l_s, dip|l_d)
0
N-1
SP
DP
Proto
RuleID
Pointer
Ptr-1
Ptr-χ
Pointer Table
P1 p2 p4 p8
8K entries
SASI-1 (4K × 1 entries)
SASI-2 (512 × 2 entries)
SASI-4 (512 × 4 entries)
SASI-8 (512 × 8 entries)
Parallel Lookups
(keyed by round-down SIP and DIP prefixes)
Lumped Hash (LuHa) Table
(8K 4-way sets)
1-1 correspondence
Primary Application-Specific Information (PASI) Table
(8K 4-way sets, with each table entry containing source and destination port ranges, protocol type, and rule ID)
Secondary ASI Data Structures
(to accommodate those ASI lists with multiple elements, pointed by PASI table entries)

Fig. 1. Overview of the HaRP* data structures implemented for 30K rules.

- parallelism exploitation: memory accesses dominate the lookup process, making it essential to exploit their available parallelism aggressively,
- pipelined design: a high clock rate requires the algorithm to be implemented in pipelining, with low and balanced complexity for pipeline stages.

A previous FPGA-based packet classification algorithm decomposes each lookup into three steps [11], with the first step for longest prefix matching over the SIP and the DIP fields and the next two steps for mapping LPM (longest prefix matching) results to the rule number followed by rule validation. The first step is performed over the compressed rule table held in on-chip memory, whereas the mapping step involves the hash table stored in off-chip memory and the validation step requires the complete rule dataset kept in off-chip memory as well. As a result, each packet classification requires two external memory accesses in the last two steps [11]. As off-chip memory is slower than its on-chip counterpart, with its bandwidth constrained, keeping the rule dataset in off-chip memory limits its achievable classification throughput. A recent packet classification aims to keep the rule dataset entirely in on-chip memory without resorting to any off-chip memory for high performance, made possible by rule grouping to reduce rule duplications for a low memory requirement [2]. It is shown on Xilinx Virtex-5 to achieve a high throughput of over 100 Gbps, without off-chip memory accesses during classification lookups.

Lately, a packet classification algorithm based on hashing round-down SIP and DIP prefixes specified in rules, dubbed HaRP as shown in Fig. 1, has been introduced [10], where superior memory efficiency results from (1) hashing round-down prefixes and (2) collapsing all hash units into one lumped hash (LuHa) table. HaRP takes less than 750KB of memory for holding any of the three large filter datasets examined (each with 30K rules) [10]. Given its low memory requirement, HaRP lends itself particularly suitable for FPGA implementation, with an entire rule dataset held in FPGA on-chip memory to exhibit high performance classification. Being hash-based, HaRP entails multiple table accesses per packet classification and those accesses can be parallelized effectively, when the LuHa table is composed of multiple memory banks, as will be detailed in later sections. The number of parallel access units involved dictates classification performance, and more memory banks support more access units to attain higher parallelism. Additionally, for the HaRP implementation to get a throughput level required by the top line rate of 100Gbps present in modern core routers, a balanced pipelining design is necessary, with its stages kept simple enough and memory banks properly situated near those parallel access units (without undesirably long routing paths) to get fast timing.

FPGA has become attractive for realizing real-time network processing engines [3, 9, 12], due to its ability to reconfigure and to offer massive parallelism. However, if a large data volume is to be stored in on-chip memory for high performance (like a rule dataset for classification), it is crucial to properly specify those FPGA memory blocks which constitute memory banks in support of parallel data accesses, given that memory blocks are distributed along strips on FPGA chips [16]. Also, LuHa table accesses are indexed by hash results and are likely to experience collision (when multiple hash results fall into the same memory bank). Collisions are unavoidable and will increase the average lookup time. In addition, among all physical elements on an FPGA chip, memory blocks usually have the largest gate delay [11], thus making the routing paths from/to the memory blocks dictate the overall speed (and thereby lookup performance) of any FPGA design with a high on-chip memory requirement.

This article deals with design and implementation of fast classification on the Xilinx Virtext-6 FPGA board based on HaRP and its enhanced variant to lower the number of LuHa table accesses per packet. Three real-life seed filter sets obtained from the public [13], namely, covering Access Control List (ACL), Firewall (FW), and IP Chain (IPC), are employed to evaluate our implemented classifier. Due to its superior memory efficiency, our implemented classifier utilizes only 50% of Virtex-6 on-chip memory to store large rule datasets (each with up to 30K rules). It is demonstrated by evaluation results to exhibit an extremely low on-chip memory requirement (reducing the byte count per rule by a factor of 8.6 in comparing with its most recent counterpart reported in [2]). The implemented HaRP pipeline with 8 processing units and 8 memory banks achieves the highest throughput among known FPGA implementations (reaching more than 200 MPPS, to support the line rate exceeding 130 Gbps under bi-directional traffic in the worst case with 40-byte packets). This is in sharp contrast to earlier implementations where memory-efficiency is often traded for throughput, rendering our implemented classifier to be *four times higher in its efficiency* (defined as the ratio of throughput to byte count per rule) than that of the second best implementation known so far [4]. In addition, enhanced HaRP is considered by reducing memory probes per lookup to further elevate the throughput level, attaining up to 40+% throughput gains but subject to far more overflows in the hash table. With an aid of pseudo set-associativity, enhanced HaRP can exhibit a lookup speed of 255 MPPS (to support the line rate beyond 160 Gbps) while containing hash table overflows.

The rest of this article is organized as follows. Section 2 gives pertinent background, including a brief review of HaRP packet classification introduced earlier [10]. Our FPGA-based design and implementation details are provided in Section 3, following by resource use and performance results presented in Section 4. Enhanced HaRP for further boosting its lookup performance by lowering the mean number of table probes is stated in Section 5, with its evaluation results included and discussed therein. Section 6 concludes this article.

2. Pertinent Background

This section provides brief reviews of HaRP for packet classification and the Xilinx Virtex-6 device, pertinent to subsequent discussion. TCAM implementation examples on the Xilinx device are also presented.

2.1. Review of HaRP for Packet Classification

A classification rule usually involves multiple fields. This work assumes five classification fields present in each rule: (1) source network IP prefix of length n, denoted as SIP|n, (2) destination network IP prefix of length m, represented by DIP|m, (3) source port range, SP[low, $high$], (4) destination port range, DP[low, $high$], and (5) protocol type range, Proto[low, $high$]. The first two fields specify a pair of communicating networks, and the next three fields apply application-specific constraints.

A memory-efficient classification method by means of generalized <u>ha</u>shing <u>r</u>ound-down <u>p</u>refixes (denoted by HaRP*) following a two-stage pipeline design has been demonstrated [10] to outperform earlier software-oriented techniques. In the first stage of HaRP*, a single set-associative hash table, referred to as the LuHa (<u>lu</u>mped <u>ha</u>sh) table is used to keep the network-prefix pair (SIP|l_s, DIP|l_d)

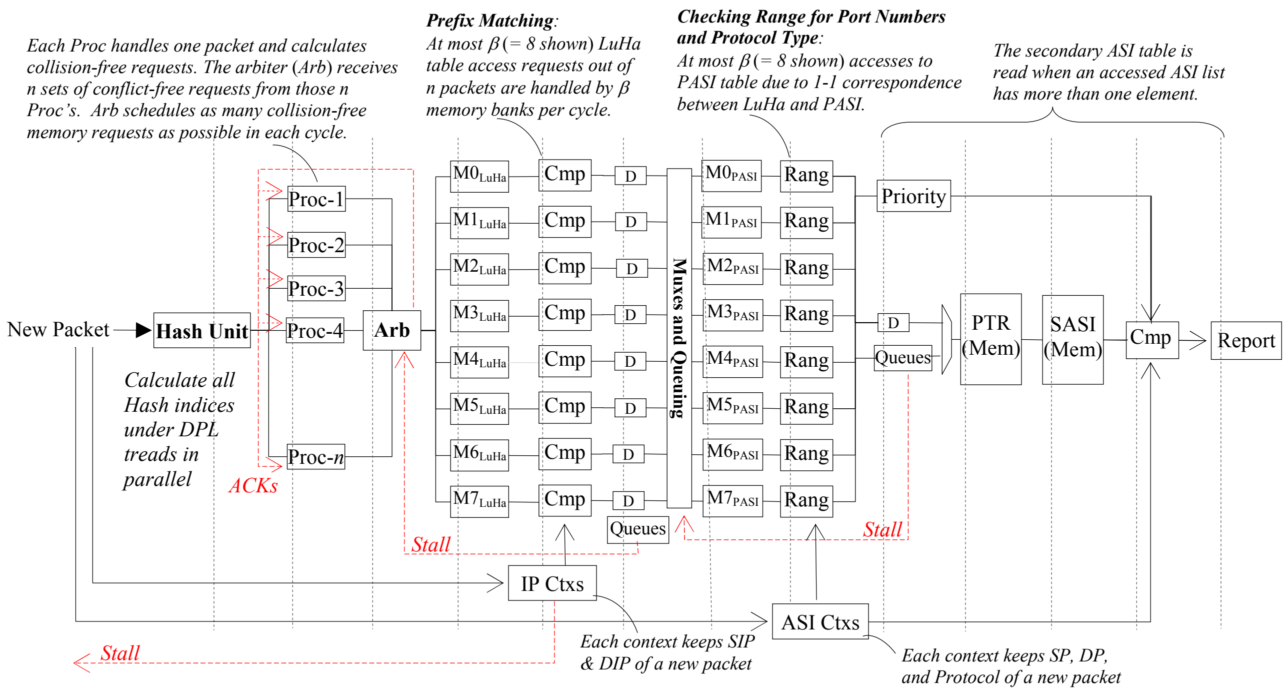

Each Proc handles one packet and calculates collision-free requests. The arbiter (Arb) receives n sets of conflict-free requests from those n Proc's. Arb schedules as many collision-free memory requests as possible in each cycle.
Prefix Matching: At most β (= 8 shown) LuHa table access requests out of n packets are handled by β memory banks per cycle.
Checking Range for Port Numbers and Protocol Type: At most β (= 8 shown) accesses to PASI table due to 1-1 correspondence between LuHa and PASI.
The secondary ASI table is read when an accessed ASI list has more than one element.
New Packet
Hash Unit
Proc-1
Proc-2
Proc-3
Proc-4
Proc-n
Arb
Calculate all Hash indices under DPL treads in parallel
ACKs
M0LuHa
M1LuHa
M2LuHa
M3LuHa
M4LuHa
M5LuHa
M6LuHa
M7LuHa
Cmp
D
Muxes and Queuing
M0PASI
M1PASI
M2PASI
M3PASI
M4PASI
M5PASI
M6PASI
M7PASI
Rang
Priority
D
Queues
PTR (Mem)
SASI (Mem)
Cmp
Report
Queues
Stall
Stall
IP Ctxs
ASI Ctxs
Stall
Each context keeps SIP & DIP of a new packet
Each context keeps SP, DP, and Protocol of a new packet

Fig. 2. Highly parallel and pipelined implementation of HaRP.

part of the rules. The other three fields involved in the second stage are stored in corresponding ASI (*Application Specific Information*) data structures.

LuHa Table Construction. Consider the set of *designated prefix length*, DPL: $\{l_1, l_2, \ldots, l_i, \ldots, l_m\}$, where l_i denotes a prefix length, for the following explanation. As depicted in Fig. 1 and detailed in [10], the LuHa table achieves efficient hash table utilization by permitting *multiple candidate sets* to accommodate the prefix pair (SIP|l_s, DIP|l_d) of a given filter rule, and yet maintaining fast search over those possible sets in parallel during the classification process. It is made possible by (1) rounding down Prefix P|w to P|l_t, for all DPL elements $l_\lambda \le w$, $\lambda \in \{1, \ldots, m\}$, before used to hash the LuHa table for identifying t (= the largest λ) candidate sets, and (2) storing (SIP|l_s, DIP|l_d) of a filter rule in *one of those multiple* LuHa candidate sets identified by either SIP|l_s (if not wild carded) or DIP|l_d (if not wild carded), after being rounded down and hashed, as stated in (1).

Since elements (called *treads*) in DPL are determined in advance, the numbers of bits in an IP address of a packet used for hash calculation during classification are clear and their hashed values can be obtained *in parallel* for concurrent search over the LuHa table. This permits parallel access units to look up the table, if multiple memory banks constitute the table, arriving at a high performance parallel design on the FPGA board.

HaRP* works because it takes advantage of the "*transitive property*" of prefixes – for a prefix P|w, P|t is a prefix of P|w for all $t < w$, considerably boosting its pseudo set-associative degree [8, 10]. Under the special case where P|w (with $l_i \le w < l_{i+1}$) is rounded down to P|l_b, for $i \le b \le i$, the method is denoted by **HaRP^1**. When the input prefix is further allowed to be rounded down to the next tread l_i (i.e., $i\text{-}1 \le b \le i$), a scheme called **HaRP^2** results. It means that the input prefix P|w can be stored in hash buckets indexed by either P|l_i or P|l_{i-1}. Accordingly, **HaRP*** is defined to allow as many candidate hash buckets (in existence) as possible for holding a given filter rule. Given DPL with 4 treads: {28, 16, 12, 8}, for example, HaRP^1 rounds down the prefix of 011010010001111001× (w = 18) to 0110100100011110 (ζ = 16) for hashing, whereas HaRP* rounds down the prefix to 0110100100011110 (ζ = 16), 011010010001 (ζ = 12), and 01101001 (ζ = 8) for hashing. With more candidate sets available potentially, HaRP* makes it possible to choose a small number of treads, which in turns involves fewer hash probes per lookup, thus improving lookup performance [10].

Construction of ASI Lists. The second stage of HaRP* comprises an ASI table, keeping the application-specific fields of filter rules. If rules share the same IP prefix pair, their application-specific fields are stored in contiguous ASI entries packed in chunks (of a fixed size) [10]. In essence, the ASI table is logically a collection of lists (with various lengths), one corresponding to one LuHa table entry, as depicted in Fig. 1.

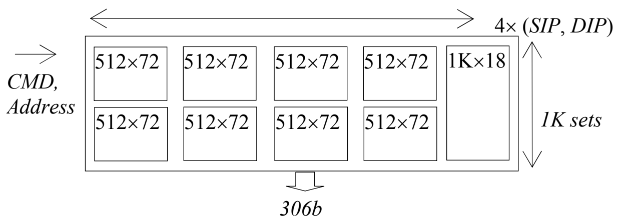


Fig. 3. Constitutive memory bank (module) of LuHa table.

2.2. Overview of Xilinx Virtex-6 Device

Our hardware HaRP* design targets the Xilinx Virtex-6 device (XC6VLX240T) [16], making efficient use of its on-chip memory and logic cells for high-performance packet classification. A Virtex-6 device contains approximately 240,000 logic cells, forming 37,680 slices, each with four LUTs (lookup tables) and eight flip-flops (FFs). A logic cell possesses combinational logics for realizing such functions as AND, OR, NAND, and addition. Flip flops and the connections to adjacent cells are also implemented by logic cells. The high-speed logic fabric of Virtex-6 permits effective pipelining [16].

Every FPGA device includes reconfigurable on-chip memory blocks (known as *block RAMs*) for implementing anything from random access storage to dual-port architectures, to FIFOs. There are 416 block RAMs in the Virtex-6 (XC6VLX240T) device for use, each containing 36Kb for a total of 14,976 Kb on-chip memory. Those block RAMs are distributed over 8 stripes, so as to have shorter routing delay for designs implemented on it. However, this layout of block RAMs can yield a design with excessively long connections, thereby hurting its performance, if large on-chip memory is required (like packet classification). Particular attention has to be paid for such situations.

2.3. TCAM Implementation Examples

We have employed the Xilinx IP core for implementing the TCAM tables of various sizes to find out hardware resource utilization and associated access timing details on the Xilinx Virtex-6 device (XC6VLX240T). A TCAM table involving 32 entries (or 128 entries), each with a 64-bit input (for an SIP and a DIP under IPv4) plus a 64-bit mask, consumes 3.1% (or 6.6%) of FPGA slices available on the Virtex-6 device. The access time to such a TCAM table with 32 entries (or 128 entries) is 4.80 *ns* (or 5.94 *ns*). If the TCAM table size rises to 512 (to hold 512 rules), its occupied slice ratio exceeds 24% (higher than that of our whole HaRP pipeline implementation able to hold some 30K classification rules; see the case of φ = 8 and ζ = 8 in Table 2) and its access time extends to 6.5 *ns* (limiting its clock rate to 154 MHz, in contrast to 300+ MHz for the HaRP pipeline). Those implementation examples confirm that TCAM indeed is expensive and relatively slow, suitable only for small-sized implementation (with no more than, say, 256 entries).

3. FPGA-Based Design and Implementation

We have implemented HaRP* on a Vertex-6 XC6VLX240T device, carefully mapping out constitutive tables to keep routing delays checked. The Xilinx Block memory generator [15] was employed to produce appropriate interfaces to on-chip memory for high speed memory accesses.

3.1. Layouts of Data Structures in Memory Blocks

There are four major tables involved, the LuHa table, the PASI table, the SASI table, and the pointer table, with the last three together realizing the ASI functionality of HaRP [10] for efficient on-chip memory utilization and fast accesses to ASI lists of variable length (as detailed later in this subsection). They are all implemented by on-chip Block memory for high-performance classification. Being the largest one, the LuHa table has a total of 8192(sets)×4(ways) entries. Each entry includes 32b SIP and DIP prefixes and two 6-bit prefix length indicators, giving rise to 76 bits in total. The LuHa table shown in Fig. 2 consists of β (= 8 shown) memory modules, with each module holding 1K sets. Such a design permits as many as β collision-free memory accesses to be served by the LuHa table. Each of the memory module is made to output 306 bits per read access, rendering four (SIP|n, DIP|m) pairs resided in one set of the 4-way LuHa table as depicted in Fig. 3.

Because the vast majority of ASI lists contain no more than one element each, the ASI table is realized by three data structures for storage efficiency and fast lookups, as shown in Fig. 1. The primary ASI (PASI) table has a 1-1 correspondence relationship to the LuHa table. Given a LuHa table with k entries, there are k corresponding PASI entries. For each classification, the PASI table is consulted only for those entries matched by prefix pair (SIP|l_s, DIP|l_d) during the first HaRP stage. Each PASI entry is 108-bit wide (see Fig. 1). In our design, each LuHa memory module outputs 4 prefix pairs per cycle for comparison, while PASI supports a more targeted access for one matched LuHa lookup. This discrepancy may require temporary queueing of requests to PASI (when there are multiple matches reported by the same LuHa set, albeit pretty rare to happen). As a result, an economy-wise decision is made to keep the PASI module narrow, instead of widening it to the output of 4×108 bits (which is likely to slow down the pipeline).

Noticeably, each PASI entry contains a pointer to a pointer table. This pointer is active when multiple filter rules share the same (SIP, DIP) prefix pair, resulting in an ASI list that has more than one element. While the PASI table keeps the first element, the additional elements of an ASI entry are kept in the four secondary SASI component tables, which are referred to as the SASI table for simplicity. Each of SASI-1 (or SASI-j, for $j \in \{2, 4, 8\}$) entries contains 1 element (or j elements). A PTR (Pointer Table) entry contains four fields to store the indexes of ASAI-i, for $i \in \{1, 2, 4, 8\}$, as depicted in Figs. 1 and 2.

By this flexible design, memory resources are better utilized to accommodate ASI lists of varying lengths. Together, one PASI entry plus its pointed SASI table entry can hold up to 16 filter rules sharing a given (SIP, DIP) LuHa entry. If the number of filter rules for the same (SIP, DIP) pair exceeds 16, another (SIP, DIP) LuHa entry is created. This can happen, for example, when a large number of filer rules are specified to put constraints on accesses to R (> 16) applications between two communicating sub-networks. During lookups, all (SIP, DIP) LuHa entries are matched such that their corresponding ASI lists are all examined to identify classification rules with the highest priority for use.

Under our design, an ASI lookup works as follow: (1) it fetches the PASI table entry corresponding to the LuHa entry indexed by the (SIP, DIP) pair of an arrival packet, compares the fields of the fetched PASI entry, and then obtains the PTR table pointer, if existing and the compared fields all matched, (2) it fetches the PTR entry via the obtained pointer for indexes to SASI tables, and (3) it gets access to SASI tables via indexes in parallel for all candidate rules, which are then examined to identify the one with the highest priority for use.

3.2. Pipelined Implementation

Fig. 2 illustrates our highly parallel and pipelined implementation under Virtex-6. There are five major functionalities, as stated in sequence below.

Hashing and Tracking States. Upon a new packet arrival, IP addresses, port numbers and the protocol type are stored in an available *context*. If there are n (= 8 shown in Fig. 2) *Proc's* in existence, at most n packets can be processed simultaneously. In the first cycle, all hash indices keyed by SIP and DIP of any arrival packet are calculated by the hash unit under the chosen DPL, as depicted in Fig. 2. For a DPL with m treads, totally $2m$ hashes are performed[†]. The $2m$ hash indices are written into an allocated *Proc*, and those indices in the *Proc* are referred to as a *context*. Given n *Proc's*, our design can handle n concurrent contexts. In other words, a context and its associated *Proc* keep track of the progress of their assigned packet.

Scheduling for LuHa Table Accesses. For n packets each with $2m$ hash table accesses, a two-level scheduler is adopted to avoid a time-critical path (when scheduling n×$2m$ accesses totally). In the first level, each *Proc* performs local scheduling: at an odd cycle, memory-collision free requests out of the m hash accesses keyed by an SIP address are chosen; at an even cycle, the same operation is performed for

[†] Note that different DPLs with different numbers of treads may be chosen for SIP and DIP prefixes/addresses. Using the same DPL for both SIP and DIP here is meant to keep our discussion simple and focused.

hash accesses keyed by a DIP address. Thus, every *Proc* can forward at most *m* access requests to the second-level *Arbiter* in each cycle. The *Arbiter* selects up to β (being the number of memory banks) collision-free accesses from up to *n*×*m* access requests forwarded by those *n* independent *Proc's*, and it lunches those selected accesses to the LuHa table. Acknowledgements are returned back to *Proc's* to nullify table access requests being served (see Fig. 2), making room for subsequent packets. As a result, this two-level scheduler permits up to β LuHa table accesses in one cycle, and a larger β tends to yield higher lookup throughputs.

LuHa Table Accesses. As demonstrated in Fig. 2 and explained before, all β LuHa memory banks may be accessed at the same time. Each memory bank outputs 4 prefix pairs, which are compared against the packet inputs. When a match is found, the corresponding PASI entry is then probed. If multiple entries of the same LuHa set report a match, they need to be serialized due to the single port PASI memory and a queue is thus provided for such a purpose, as shown in Fig. 2.

PASI Table Accesses. The β PASI memory banks can accept β reads per cycle. Each memory bank output ASI stored in the entry being accessed. If a match to the port range and the protocol type is found, the rule ID is reported. If the range check fails, searches to the secondary ASI tables continue, provided that the PASI entry contains a valid pointer, which indexes to Pointer Table (as demonstrated in Fig. 1 and marked as "PTR (Mem)" in Fig. 2) where the SASI table entries are specified. This way permits as many as 16 rules and their associated ASI values to be fetched quickly from those SASI tables in parallel, as illustrated in Figs. 1 and 2. Our design preserves superior memory efficiency of HaRP and also achieves fast accesses to a varying number of candidate rules (associated with one LuHa table entry) in a uniform time.

SASI Table Accesses. Accesses to the SASI tables are targeted, fetching only entries specified by the address pointers reported by the pointer table. Range checking on port numbers and the protocol type are performed for all fetched entries in parallel.

4. IMPLEMENTATION EVALUATION

The LuHa table in our implemented classifier is 4-way set-associative. Our evaluation is under the default DPL with 4 treads of {8, 20, 23, 27} for SIP and of {8, 20, 24, 30} for DIP, chosen conveniently, not necessary to yield the best results. From real filter datasets (containing up to 1,550 rules) available in the public domain [13], three synthetic datasets, each with about 30K rules (see Table 1), have been obtained for evaluation, including Access Control List (ACL), Firewall (FW), and IP Chain (IPC), for evaluation under various numbers of *Proc's* and memory banks.

Table 1. On-chip memory usage

Rule dataset	# of rules	Taken on-chip memory (Kbytes)	Usage ratio
ACL	28240	655.2	35.0 %
FW	28473	697.6	37.3 %
IPC	29876	776.8	41.5 %

4.1. Memory Requirement

Details of on-chip memory taken by the four tables in support of HaRP classification under the three datasets are listed in Table 1. Each LuHa entry is 76-bit long, comprising two 32b IP address prefixes and two 6b prefix length indicators. Each PASI entry needs 10 bytes to keep the port ranges and the protocol type, 15-bit rule number (for priority decision), and a pointer to the PTR table. Each PTR table entry contains 4 pointers for indexing SASI component tables, i.e., SASI-1, SASI-2, SASI-4, and SASI-8, whose entries each contain 1, 2, 4, and 8 ASI elements of 95 bits, respectively (see Fig. 1). The memory sizes of the PTR table and the SASI component tables depend on the dataset size.

Our FPGA on-chip memory usage results reveal that each filter rule (under any one of the three datasets examined) on an average takes no more than 26 bytes. They are far favorable in comparison to a recent packet classification design based on SPMT (Set Pruning Multi-Bit Trie) [2], where 10K FW rules utilize 16.88 Mb on-chip memory on a Xilinx Virtex-5 (XC5VFX200T) FPGA device, signifying that each FW rule requires 211 bytes. Our classifier indeed has an extremely low memory, thus enabling a far larger rule dataset to fit in on-chip FPGA memory for speedy classification. More comparative details among various recent FPGA-realized classifiers will be provided in Section 4.3. Clearly, overall on-chip memory required by our FPGA implementation is dictated solely by rule datasets, irrespective of the numbers of Proc's (denoted by φ) and memory banks (denoted by ζ) involved. On the other hand, hardware logic requirement for HaRP implementation is proportional to φ and ζ, which in turn determine lookup throughput outcomes, as stated next.

4.2. Requirement of Hardware Logics

Hardware logics used to realize the HaRP classifier shown in Fig. 2 include LUTs and flip flops (FFs), which are from FPGA slices. The numbers of FFs and LUTs consumed by the implemented classifier under various φ and ζ are depicted in Table 2, where the number of occupied slices (out of 37,680 on the VLX240T FPGA device) is also included. An increase in either φ or ζ leads to higher consumed slice FF and slice LUT counts monotonically, as expected. When (φ, ζ) rises from (4, 4) to (4, 8), the numbers (or percentages) of taken slice FFs and slice LUTs grow respectively to 18,572 and 21,699 (or to 6% and 14%), from 12,414 and 13,977 (or from 4% and 9%), as unveiled in the table. If (φ, ζ) is elevated further to (8, 8), 8% of slice FFs and 20% of slice LUTs will be occupied. Given each slice contains four LUTs and eight FFs, which are not always utilized in full by an implementation, it is useful to know the number of slices involved (either partially or fully) in our classifier implementation. From Table 2, the number

Table 2. Usage of hardware logics

(φ, ζ) (no. of proc's, memory banks)	Consumed hardware breakdowns		
	Slice registers	Slice LUTs	Occupied slices
(1, 1)	4,313 (1 %)	5,124 (3 %)	1,763 (4 %)
(2, 2)	6,995 (2 %)	7,597 (5 %)	2,574 (6 %)
(2, 4)	9,604 (3 %)	10,954 (7 %)	3,548 (9 %)
(4, 4)	12,414 (4 %)	13,799 (9 %)	4,465 (11 %)
(4, 8)	18,572 (6 %)	21,699 (14 %)	6,706 (17 %)
(8, 8)	25,944 (8 %)	31,048 (20 %)	9,374 (24 %)
(8, 16)	41,923 (13 %)	49,048 (32 %)	14,452 (38 %)
(16, 16)	63,856 (21 %)	85,253 (56 %)	24,888 (66 %)

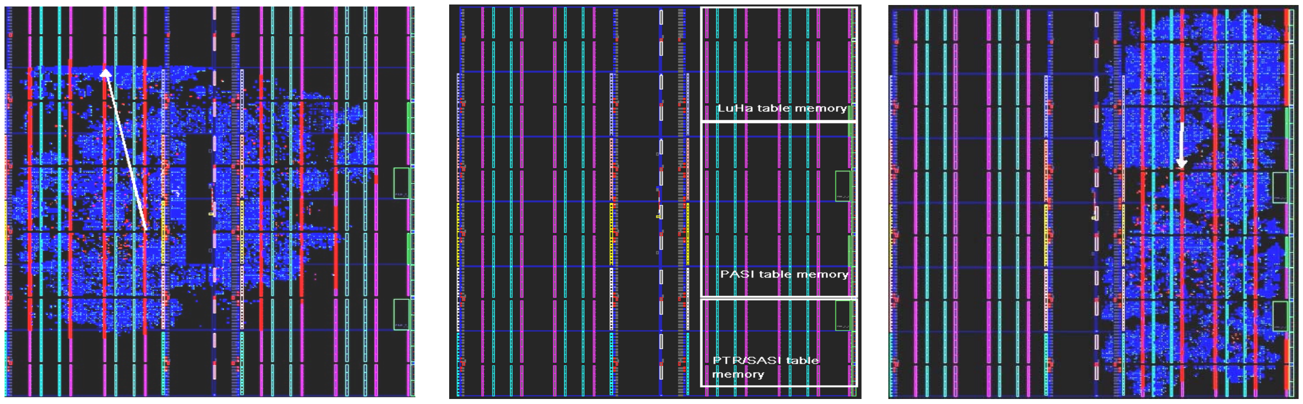
of involved slices is seen to grow from 4,465 (~ 11%) under (φ, ζ) = (4, 4) to 9,374 (~ 24%) under (φ, ζ) = (8, 8). This signifies that one Virtex-6 board can easily accommodate two copies of HaRP classifier with (φ, ζ) = (8, 8). If φ or ζ increases to 16, the percentage of occupied slices rises to 38% or beyond, reflecting that a good portion of available slices on the VLX240T FPGA device is taken. This high usage of hardware logics in fact will make the implemented HaRP classifier run slower due to longer routing paths between its consecutive pipeline stages when mapped to FPGA logics and block RAMs, as detailed next.

On-chip memory on FPGA devices is organized in blocks (i.e., block RAMs/FIFOs), which are scattered over the whole Virtex-6 chip. As hardware logics are to operate on dataset contents stored in on-chip memory, the achievable memory access rate is critical for the pipeline clock rate, dictated by the worst-case routing path delay. An undesirably lengthy delay may result from utilizing FPGA slices unduly distant from those block RAMs employed to hold the filter dataset. To avoid an inefficient HaRP implementation, the design is constrained within one half of the FPGA device to arrive at a compact layout without any excessive path delay. This way makes those occupied FPGA slices stay near to slices which realize the HaRP functions, rendering a design with the main clock exceeding 300 MHz.

Our implementation puts constraints on those tables basic to HaRP classification, as depicted in Fig. 4(b). For comparison, layout results with and without constraints are demonstrated respectively in Fig. 4(a) and Fig. 4(c). Without a constraint, on-chip block RAMs are seen to be often distant from those taken slices that realize on-chip memory access and processing logics, leading to long route delays for paths from a pipeline stage to the next, as shown in Fig. 4(a), where an exceedingly long path marked by a white line with arrow indicates a memory access route (i.e., from the arbiter stage to the LuHa table access stage, see Fig. 2). If constraints are put to tables realized by blocks of on-chip memory, as illustrated in Fig. 4(b), an improved layout results, with its worst-case path (denoted by the white line with arrow in Fig. 4(c) to indicate the routes from the LuHa Table output to the comparator in the next pipeline stage) shrunk drastically. Hence, the constrained use of memory blocks yields a better layout with more uniform and shorter routes, permitting a faster clock to deliver higher lookup throughput.

4.3. Implementation Results and Discussion

The major performance metric of interest is lookup throughput, which equals the number of classification lookups per second. As mentioned earlier, the line rate of 100 Gbps necessitates one lookup per 6.4 *ns* in the worst case under bi-directional traffic, amounting to some 156 MPPS (million packets per second) for the shortest packets of 40 bytes in length. The throughput results under various φ and ζ (i.e., numbers of Proc's and memory banks) are listed in Table 3 for the three rule datasets considered, with φ ≤ ζ to avoid unnecessary access conflicts at memory blocks, thus capitalizing on parallelism fully. They indicate that the desired lookup speed of 156 MPPS can be achieved for φ = 8 and ζ = 8, where the implemented HaRP design



(a) Layout without constraint (b) Imposed memory constraints (c) Layout with constraints

Fig. 4. Layout of occupied memory blocks and taken FPGA slices.

(Taken FPGA slices are denoted in blue, while occupied memory blocks and their access logics are represented by red stripes and nearby red dots, respectively. Those stripes in purple and in green denote respectively unused memory blocks and the DSP blocks irrelevant to our implementation. Similarly, objects in other colors are unneeded chip components.)

Table 3. HaRP performance under rule datasets considered

(φ, ζ) (no. of proc's, memory banks)	**Lookup Throughput** (MPPS)		
	ACL	FW	IPC
(1, 1)	39.7	42.5	40.6
(2, 2)	57.4	57.5	58.1
(2, 4)	69.1	68.3	68.9
(4, 4)	106	105	105
(4, 8)	126	127	128
(8, 8)	204	213	213
(8, 16)	171	178	177

operates in excess of 300 MHz. According to Table 2, the occupied slices then account for only 24% of what are available on the Virtex-6 FPGA device. On the other hand, the rate of occupied slices rises to 38% for φ = 8 and ζ = 16, where the implemented classifier slows down to 220 MHz only, resulting in smaller throughputs to drop as despite its doubled memory banks for fewer access conflicts (see Table 3). This is because those slices employed for realizing HaRP functions then cannot all be situated near their fetched memory blocks. As a result, detrimentally long routes exist in the implementation and thus the main clock rate is dropped (to 220 MHz from more than 300 MHz under φ, ζ ≤ 8), lowering the overall throughput. For φ = 16 and ζ = 16, a far slower clock rate is obtained because the HaRP implementation then occupies 66% of available slices. While not covered in this article, a rectified HaRP pipeline with one extra stage dedicated to route latency reduction can be added under φ = ζ = 16 for improving its throughput (assorted to a deep pipelining design for a higher clock rate).

In general, the throughput figure of an implemented design is dictated by four factors: the main clock rate, numbers of Proc's and memory blocks, and effectiveness of scheduling memory accesses. If the slices are not heavily utilized (say, with the taken rate < 30%), the main clock rate stays almost identically, since the longest route length then remains unchanged. Under that situation (of a low occupied rate), the overall throughput depends only on the remaining three factors. Our two-stage scheduler for memory accesses intends to select as many conflict-free accesses as possible per cycle for maximal throughput, as detailed in Section 3.2. Under the given scheduler, a higher throughput value results from either a larger φ or a larger ζ, provided that φ is no more than 8. With φ = 4, for example, the implemented classifier enjoys a throughput increase of roughly 25% when ζ grows from 4 to 8, as a result of more conflict-free accesses to memory blocks per cycle (see Table 3). Likewise, for ζ = 8, its throughput has a leap exceeding 60%, if φ rises from 4 to 8, directly benefiting from twice memory access requests generated by Proc's per cycle.

In addition to its high throughput, our implemented classifier enjoys a much lower on-chip memory requirement when compared with a counterpart introduced recently, dubbed SPMT (Set Pruning Multi-Bit Trie) [2], as mentioned in Section 4.1. This high throughput plus low memory requirement makes our HaRP design especially preferred over other known classification approaches implemented using FPGA, as reflected by the "efficiency" measure (defined in [4] as the ratio of throughput to on-board memory per rule, accounting for both time and space factors) listed in Table 4, where HaRP(8, 8) indicates the HaRP classifier with φ = 8 and ζ = 8. While the table includes only the results under the ACL rule dataset, the result trends hold under other rule datasets. Apparently, high efficiency may result from an approach with either a large throughput or a small memory amount per rule. Unlike the SPMT design which trades the memory requirement for an increased throughput in comparison to Improved HyperCuts [4] (see the 3rd and 4th rows of Table 4), the HaRP classifier enjoys both memory requirement reduction (per rule) and a throughput hike when compared with any prior approach, yielding significantly better efficiency. An enhanced HaRP design able to further elevate the lookup throughput is treated next.

Table 4. Performance comparison of classifiers on FPGA

Approaches	**# of ACL rules**	**Total memory taken** (Kbytes)	**Memory per rule**	**Throughput** (Gbps)	**Efficiency** (Gbps/B)
HaRP(8, 8)	28240	655.2	23.2 bytes per rule	130.6	5.63
SPMT [2]	9603	1930	201 bytes per rule	110.7	0.55
Improved HyperCuts [4]	9603	612	63.7 bytes per rule	88.2	1.38
Simplified HyperCuts [6]	10000	286	28.6 bytes per rule	7.22	0.25
2sBFCE [9]	4000	178	44.5 bytes per rule	2.06	0.046
Memory-Based DCFL [3]	128	221	1727 bytes per rule	24.0	0.014

5. Enhanced HaRP and Its Performance

The HaRP classifier considered so far always involves 2×|DPL| memory accesses to the LuHa table for every packet during its lookup. In each cycle, only those non-conflict accesses to different memory banks are chosen to proceed by the developed two-stage scheduling mechanism, with those remaining conflict accesses, if any, scheduled in the subsequent cycle (together with those 2×|DPL| memory accesses of the next packet). The lookup of a packet is done only after its associated memory accesses are all served. While a large φ or ζ can yield a higher throughput (as demonstrated in the last section), an enhanced HaRP design is treated here by *lowering the number of memory accesses per lookup dynamically* (instead of having a fixed 2×|DPL| memory accesses for each packet lookup) under given φ and ζ. This is made possible by taking advantage of the unique property of HaRP in that the filter table lookup process is dictated by the filter rule installation procedure, which can

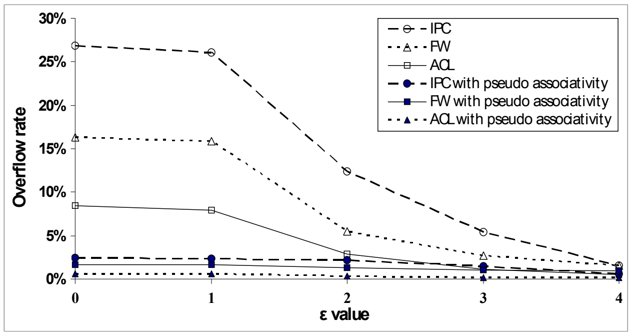


Fig. 5. Overflow rate versus ε under enhanced HaRP.

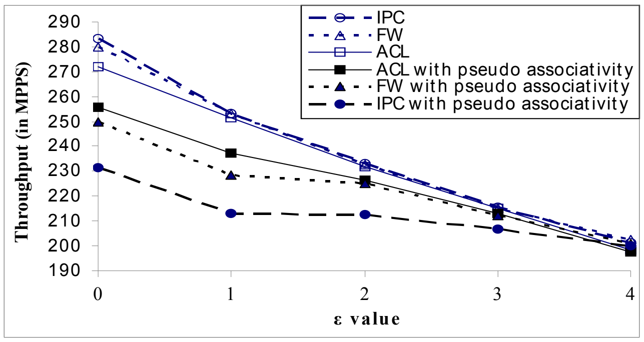


Fig. 6. Lookup throughput versus ε under enhanced HaRP.

follows a control guide to lower the number of candidate sets dynamically for each rule so that the lookup process then involves fewer memory probes accordingly based on the same guide.

In general, the number of candidate sets under enhanced HaRP may range from 1 to 2×|DPL|, determined by the control guide that specifies which treads in DPL are to be "conditioned" before applied for hashing. Two components are involved in the guide: (1) which treads in DPL to be conditioned and (2) how to condition those treads. Let the number of conditioned treads in given DPL be denoted by (|DPL| − ε), for $0 \le \varepsilon \le |\mathrm{DPL}|$, and the condition simply be "rightmost tread bit being "1." Naturally, other conditions are possible for throughput improvement as well, e.g., "rightmost tread bit equal to 0" or "rightmost two tread bits being '01' (if two bits are for conditioning)." The following deals with only single-bit conditioning as one example enhancement.

5.1. Example Enhancement Guide and Results

Under enhanced HaRP, an IP prefix (or address) determines its candidate sets for holding the prefix (or for probing the best filter rule to apply) according to the given guide. Apparently, fewer candidate sets result in a higher lookup throughput and also in more LuHa table overflows (during rule installation, since fewer alternative sets are then available for a given prefix) as well. Installing filter rules without excessive overflows thus calls for a larger ε, which in turn contains the throughput as a result. The overflow rates under different ε values will be explored.

The guide may be in different forms, with one considered as follows. Given DPL = $\{l_1, l_2, \ldots, l_m\}$ with m elements, HaRP rounds down Prefix P of length w (or IP address of length 32) to $P|l_t$, for all DPL elements $l_\lambda \le w$, $\lambda \in \{1, \ldots, m\}$, before employed to hash the LuHa table for identifying candidate sets, provided that *the rightmost bit of* $P|l_t$ *equal to 1 for* $t > \varepsilon$. For Prefix P during filter rule installation (or IP address P during filter lookup) under DPL = {8, 20, 23, 27} and $\varepsilon = 1$, as an example, enhanced HaRP considers only those candidate sets (1) indexed by P|20 with its rightmost bit equal to 1, by P|23 with its rightmost bit equal to 1, and by P|27 with its rightmost bit equal to 1 and (2) indexed by P|8, regardless of their rightmost bits (i.e., no conditioning). This guide for rule installation and lookups lowers the mean number of LuHa table probes (thereby elevating the lookup throughput).

Enhanced HaRP can be accommodated easily in the "Hash Unit" of the implementation pipeline shown in Fig. 2, with the unit deciding whether or not a round-down address should be used for probing a candidate set in the LuHa table. The miss rate versus ε values for the three filter datasets (listed in Table 1) during filter rule installation under enhancement HaRP is shown by the top three curves in Fig. 5, where the LuHa table is 4-way set-associative with 8K sets for a total of 32K entries and the DPL for source IP (or destination IP) prefixes/addresses is {8, 20, 23, 27} (or is {8, 20, 24, 30}). As expected, a larger ε leads to a lower overflow rate for every rule dataset, thereby requiring a smaller TCAM to store those overflow rules (if a small TCAM is assumed to handle the spillovers). Given its |DPL| equal to 4, enhanced HaRP becomes regular HaRP under $\varepsilon = 4$, where all treads are used for indexing candidate sets. The overflow rates of three datasets all drop quickly when ε exceeds 1.

Throughput as a function of ε during lookups under enhanced HaRP with $\varphi = 8$ and $\zeta = 8$ is shown in Fig. 6. As seen in the figure, the throughput of every filter dataset is lower for a larger ε, and it reaches the result of regular HaRP with $\varepsilon = 4$ (i.e., being the value listed in the row of $(\varphi, \zeta) = (8, 8)$ in Table 3). Enhanced HaRP with $\varepsilon = 0$ (or 1) under ACL, for example, enjoys some 36% (or 25%) improvement in the lookup throughput, at the expense of a far higher overflow rate (see Fig. 5), and thus a substantially larger spillover TCAM, during LuHa table installation.

5.2. Enhanced HaRP with Pseudo Set-Associativity

A pseudo associative technique [8] can be applied to contain overflows during rule installation, making the 4-way set-associative LuHa table behave like 8-way set-associativity by treating both a set (indexed by a hash value) and its companion (indexed by the 2's complement of the hash value) as candidates to store a filter rule during installation, thus lowering the overflow rate. A companion candidate set is checked only if its associated candidate set is unavailable to hold the rule. During rule lookups, an indexed candidate is probed first before its companion candidate set is examined in the next cycle, should the earlier probe fail to match a rule. When both candidate sets are unavailable for a filter rule (or fail to match any rule), an overflow (or a miss) happens during rule installation (or lookups).

With pseudo set-associativity, enhanced HaRP exhibits far smaller overflow rates under the three filter datasets examined throughout the ε range shown in Fig. 5. The bottom three curves in the figure clearly signify that doubling candidate sets for each filter rule avoids the need of large TCAMs to hold overflow rules for enhanced HaRP. Meanwhile, pseudo set-associativity is seen to let enhanced HaRP outperform its regular HaRP counterpart smartly for a small ε (say, ≤ 1), as demonstrated by the bottom three curves in Fig. 6. Under ACL, for example, enhanced HaRP with pseudo set-associativity still enjoys a throughput gain of some 27% with $\varepsilon = 0$, reaching a lookup rate of 255 MPPS (to support the wire speed over 160 Gbps under bi-directional traffic in the worst case with 40-byte packets). Despite elevating the bandwidth requirement for LuHa table accesses upon classification lookups (due to the possible need of examining companion candidate sets), pseudo set-associativity indeed benefits enhanced HaRP in both overflow reduction and throughput improvement.

6. CONCLUSION

This article has investigated into design and implementation of speedy classification based on HaRP, whose constitutive lookup tables are mapped carefully onto on-chip block RAMs of an Xilinx Vertex-6 FPGA device (XC6VLX240T) to keep routing delays checked. The implemented classifier has an extremely low memory requirement, based on evaluation results obtained using three large datasets generated by real-life seed filter sets available to the public [13]. Its *efficiency* (defined as the ratio of throughput to byte count per rule) *is four times higher* than that of the second best FPGA implementation known so far [4]. An enhanced variant to lower the number of hash table accesses per packet is considered, shown to further elevate the classification throughput with overflows in the hash table contained via pseudo set-associativity.

ACKNOWLEDGEMENT

The authors thank Itthichok Jangjaimon for his assistance in obtaining some FPGA implementation results included in this article.

REFERENCES

[1] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an alternative to CAMs," *Proceedings of 22nd IEEE International Conference on Computer Communications* (*INFOCOM 2003*), pp. 53–63, Mar./Apr. 2003.

[2] Y.-K. Chang, Y.-S. Lin, and C.-C. Su, "A High-Speed and Memory Efficient Pipeline Architecture for Packet Classification," *Proceedings of 18th IEEE International Symposium on Field-Programmable Custom Computing Machines* (*FCCM 2010*), pp. 215-218, May 2010.

[3] G. S. Jedhe, A. Ramamoorthy, and K. Varghese, "A Scalable High Throughput Firewall in FPGA," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines* (*FCCM 2008*), Apr. 2008.

[4] W. Jiang and V. K. Prasanna, "Large-Scale Wire-Speed Packet Classification on FPGAs," *Proceedings of 17th ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays* (*FPGA '09*), Feb. 2009.

[5] Juniper Networks, Inc., "DataSheet of 100-Gigabit Ethernet PIC," 2011. URL − http://www.juniper.net/us/en/local/pdf/datasheets/1000346-en.pdf.

[6] A. Kennedy et al., "Low Power Architecture for High Speed Packet Classification," Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems (*ANCS 2008*), Nov. 2008.

[7] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CANs," *Proceedings of ACM Annual Conference of Special Interest Group on Data Communication* (*SIGCOMM '05*), pp. 193–204, Aug. 2005.

[8] Y. J. Lee and B.-K. Chung, "Pseudo 3-way Set Associative Cache: A Way of Reducing Miss Ratio with Fast Access Time," *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering*, May 1999.

[9] A. Nikitakis and I. Papaefstathiou, "A Memory-Efficient FPGA-Based Classification Engine," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines* (*FCCM 2008*), Apr. 2008.

[10] F. Pong and N.-F. Tzeng, "HaRP: Rapid Packet Classification via Hashing Round-Down Prefixes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 1105-1119, July 2011.

[11] V. Pus and J. Korenek, "Fast and Scalable Packet Classification Using Perfect Hash Functions," *Proc. of 17th International ACM Symposium on Field Programmable Gate Arrays* (*FPGA '09*), pp. 229-235, Feb. 2009.

[12] D. E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238–275, Sept. 2005.

[13] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," *Proceedings of 24th IEEE International Conference on Computer Communications* (*INFOCOM 2005*), Mar. 2005.

[14] G. Wang and N.-F. Tzeng, "TCAM-Based Forwarding engine with Minimum Independent Prefix Set (MIPS) for Fast Updating," *Proceedings of IEEE International Conference on Communications* (*ICC '06*), June 2006.

[15] Xilinx Corporation, "IP-SysLogic-BlockMem Generator: Block Memory Generator v3.3," *AR#* 33298, Sept. 2009. URL − http://www.xilinx.com/support/answers/33298.htm.

[16] Xilinx Corporation, "Virtex-6 Family Overview," March 2011. URL − http://www.xilinx.com/product/silicon-devices/fpga/virtex-6/index.htm.