HaRP: Rapid Packet Classification via Hashing Round-Down Prefixes

Fong Pong, *Senior Member*, *IEEE*, and Nian-Feng Tzeng, *Fellow*, *IEEE*

Abstract—Packet classification is central to a wide array of Internet applications and services, with its approaches mostly involving either hardware support or optimization steps needed by software-oriented techniques (to add precomputed markers and insert rules in the search data structures). Unfortunately, an approach with hardware support is expensive and has limited scalability, whereas one with optimization fails to handle incremental rule updates effectively. This work deals with rapid packet classification, realized by hashing round-down prefixes (HaRP) in a way that the source and the destination IP prefixes specified in a rule are rounded down to "designated prefix lengths" (DPL) for indexing into hash sets. HaRP exhibits superb hash storage utilization, able to not only outperform those earlier software-oriented classification techniques but also well accommodate dynamic creation and deletion of rules. HaRP makes it possible to hold all its search data structures in the local cache of each core within a contemporary processor, dramatically elevating its classification performance. Empirical results measured on an AMD 4-way 2.8 GHz Opteron system (with 1 MB cache for each core) under six filter data sets (each with up to 30 K rules) obtained from a public source unveil that HaRP enjoys up to some $3.6\times$ throughput level achievable by the best known decision tree-based counterpart, HyperCuts (HC).

Index Terms—Classification rules, decision trees, filter data sets, hashing functions, incremental rule updates, IP prefixes, packet classification, routers, set-associative hash tables, tuple space search.

1 Introduction

Packet classification is performed at routers by applying "rules" to incoming packets for categorizing them into flows. It employs multiple fields in the header of an arrival packet as the search key for identifying the best suitable rule to apply. Rules are created to differentiate packets based on the values of their corresponding header fields, constituting a filter set. A field value in a filter can be an IP prefix (e.g., source or destination subnetwork), a range (e.g., source or destination port numbers), or an exact number (e.g., protocol type or TCP flag). A real filter data set often contains multiple rules for a pair of communicating networks, one for each application. Similarly, an application is likely to appear in multiple filters, one for each pair of communicating networks using the application. Therefore, lookups over a filter set with respect to multiple header fields are complex [8] and can easily become router performance bottlenecks.

Various classification mechanisms have been considered, and they aim to quicken packet classification through hardware support or the use of specific data structures to hold filter data sets (often in SRAM and likely with optimization) for fast search [19]. Hardware support frequently employs field programmable gate arrays (FPGAs) or ASIC logics [4], [17], plus ternary content addressable memory (TCAM) to hold filters [21] or registers for rule caching [7]. A classification mechanism with hardware support usually cannot handle incremental rule updates well, since any change to the mechanism (in its search algorithm or data structures) is expensive. Additionally, it exhibits limited scalability, as TCAM (or registers) employed to hold a filter set would dictate the maximal set size allowable. Likewise, software-oriented search algorithms dependent on optimization via preprocessing (used by recursive flow classification [8]) or added markers and inserted rules (stated in rectangle tuple space search (TSS) [18], binary TSS on columns [22], diagonal-based TSS [12], etc.) for speedy lookups often fail to deal with incremental rule updates effectively. The inherent limitation of classifiers (be hardware- or software-oriented ones) in handling incremental rule updates (deemed increasingly common due to such popular applications as voice-over-IP, gaming, and video conferencing, which all involve dynamically triggered insertion and removal of rules in order for the firewall to handle packets properly) will soon become a major concern [24].

This paper treats hashing round-down prefixes (HaRP) for rapid packet classification, where an IP prefix with l bits is rounded down to include its first ζ bits only (for $\zeta \leq l, \zeta \in$ DPL, "designated prefix lengths" [14]). With two-staged search, HaRP achieves high classification throughput and superior memory efficiency by means of **1**) rounding down prefixes to a small number of DPL (denoted by m, i.e., m possible designated prefix lengths), each corresponding to one hash unit, for fewer (than 32 under IPv4, when every prefix length is permitted without rounding down) hash accesses per packet classification, and **2**) collapsing those hash units to one lumped hash (LuHa) table for better utilization of table entries, which are set-associative. Based on a LuHa table keyed by the source and destination IP prefixes rounded down to designated lengths, HaRP not

- *F. Pong is with the Broadcom Corporation, 2451 Mission College Boulevard, Santa Clara, CA 95054. E-mail: fpong@Broadcom.com.*
- *N.-F. Tzeng is with the Center for Advanced Computer Studies, University of Louisiana, Lafayette, LA 70504-4330. E-mail: tzeng@cacs.louisiana.edu.*

Manuscript received 9 Sept. 2009; revised 18 Feb. 2010; accepted 30 June 2010; published online 8 Nov. 2010.
Recommended for acceptance by J. Zhang.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-09-0416.
Digital Object Identifier no. 10.1109/TPDS.2010.195.

only enjoys fast classification (due to a small number of hash accesses to SRAM) but also handles incremental rule updates efficiently (without precomputing markers or inserting rules often required by typical TSS). With its required SRAM size dropped considerably (to less than 800 KB for all six filter data sets examined), generalized HaRP (denoted by HaRP^*) permits all its search data structures possibly to be held in the local cache of each core in a processor, further boosting its classification performance.

Our LuHa table yields high storage utilization via identifying multiple candidate sets for each rule (instead of just a single one under a typical hash mechanism), like the earlier scheme of d-left hashing [1]. However, the LuHa table differs from d-left hashing in three major aspects: **1**) the LuHa table requires just one hash function, as opposed to d functions needed by d-left hashing (which divides storage into d fragments), one for each fragment, **2**) the hash function of the LuHa table under HaRP^* is keyed by $2m$ different prefixes produced from each pair of the source and the destination IP addresses, and **3**) a single LuHa table obtained by collapsing separate hash units is employed to attain superior storage utilization, instead of one hash unit per prefix length to which d-left hashing is applied.

Extensive evaluation of HaRP has been conducted on our AMD system, which comprises four 2.8 GHz Opteron processors with 1 MB cache each, under six filter data sets obtained from a public source [23]. The proposed HaRP was made multithreaded so that up to four threads could be launched to take advantage of the four AMD cores. Measured throughput results of HaRP are compared with those of HyperCuts (deemed the best known decision tree-based classifier [16]), whose source codes were downloaded from a public source [23], and then made multithreaded for executing on the same platform to classify millions of packets generated from the traces packaged with the filter data sets. Our measured results reveal that HaRP^* boosts classification throughput by some $3.6\times$ over HyperCuts (or HC for short), when its LuHa table has a total number of entries equal to $1.0n$ and there are five designated prefix lengths, for a filter data set sized n. HaRP attains superior performance, on top of its efficient support for incremental rule updates lacked by previous techniques, making it a highly preferable software-oriented packet classification technique.

2 Pertinent Work

Classification lookup mechanisms may be categorized, in accordance with their implementation approaches, as being hardware-centric and software-oriented, depending upon if dedicated hardware logics or specific storage components (like TCAM or registers) are used. Different hardware-centric classification mechanisms exist. In particular, a mechanism with additional registers to cache evolving rules and dedicated logics to match incoming packets with the cached rules was pursued [7]. Meanwhile, packet classification using FPGA was considered [17] by using the BV (Bit Vector) algorithm [11] to look up the source and destination ports and employing a TCAM to hold other header fields, with search functionality realized by FPGA logic gates. Recently, packet classification hardware accelerator design based on the HiCuts and HyperCuts (HC) algorithms [3], [16] (briefly reviewed in Section 2.1), has been presented [10]. Separately effective methods for dynamic pattern search were introduced [4], realized by reusing redundant logics for optimization and by fitting the whole filter device in a single Xilinx FPGA unit, taking advantage of built-in memory and XOR-based comparators in FPGA.

Hardware approaches based on TCAM are considered attractive due to the ability for TCAM to hold the don't care state and to search the header fields of an incoming packet against all TCAM entries in a rule set simultaneously [13], [21]. Widely employed storage components in support of fast lookups, TCAM has such noticeable shortcomings (listed in [19]) as lower density, higher power consumption, and being pricier and unsuitable for dynamic rules, since incremental updates usually require many TCAM entries to be shifted (unless provision like those given earlier [15], [21] is made). As a result, software-oriented classification is more attractive, provided that its lookup speed can be quickened by storing rules in on-chip SRAM.

2.1 Software-Oriented Classification

Software-oriented mechanisms are less expensive and more flexible (better adaptive to rule updates), albeit to slower filter lookups when compared with their hardware-centric counterparts. Such mechanisms are abundant, commonly involving efficient algorithms for quick packet classification with an aid of caching or hashing (via incorporated SRAM). Their classification speeds rely on efficiency in search over the rule set (stored in SRAM) using the keys constituted by corresponding header fields. Several representative software classification techniques are reviewed in sequence.

Recursive flow classification (RFC) carries out multistage reduction from a lookup key (composed of packet header fields) to a final *classID*, which specifies the classification rule to apply [8]. Given a rule set, preprocessing is required to decide memory contents so that the sequence of RFC lookups according to a lookup key yields the appropriate *classID* [8]. Based on a precomputed decision tree, Hierarchical Intelligent Cuts (HiCuts) [9] holds classification rules merely in leaf nodes and each classification operation needs to traverse the tree to a leaf node, where multiple rules are stored and searched sequentially. During tree search, HiCuts relies on local optimization decisions at each node to choose the next field to test. HyperCuts (HC) is an improvement over HiCuts by allowing cuts to be made over multiple dimensions at a node [16], as opposed to just a single dimension each in HiCuts. At every node, there can be totally $\prod_{i=1}^{D} nc(i)$ child nodes, where $nc(i)$ is the number of splits made in the ith dimension. Like HiCuts, HC is also a decision tree-based classification mechanism, but each of its tree nodes splits associated rules possibly based on multiple fields. It is shown to enjoy substantial memory reduction while considerably quickening the worst-case search time under core router rule sets [16], when compared with HiCuts and other earlier classification solutions.

Given that decision tree-based methods (like HiCuts and HC) in general are notorious for the *tree size explosion* problem (with the decision tree size highly depending on the data sets and possibly to grow exponentially), refinement techniques are introduced to reduce their storage

requirements. The tradeoff between storage and lookup performance can be measured by the *space factor* (***SF***), with a larger *SF* value yielding a wider and shallower decision tree. A larger *SF* is expected to consume more storage but support faster lookups.

Meanwhile, storage-saving for decision tree-based classifiers can be achieved by pushing common rules upwards, aiming to keep a common set of rules at the parent node if the rules hold true for all of its child nodes. Although this way lets rules be associated with nonleaf nodes to save storage by avoiding replicas at the leaves, it can degrade lookup performance, as a lookup then has to examine internal nodes. Additionally, since decision trees are known to involve excessive (child node) pointers, a common fix lets the parent node keep merely the starting base address pointing to its first child plus an additional n-bit "Extended Path Bitmap" (EPB) to remember existing child nodes [16]. This pointer compression reduces space greatly from n pointers to one plus $(n-1)$ bits. Also, practical decision tree implementation stores all filter rules in a memory array of consecutive locations; the decision tree only keeps indices in the tree nodes. This reduces storage by avoiding replicas of filter rules which hold true for multiple subtrees due to wildcard addresses or port ranges. Adversely, such compression techniques make incremental updates very difficult. The linear memory array representation leaves holes upon rule deletions and is hard to accommodate new rules. The resulting indirect lookup process becomes inefficient because memory, in principle, exhibits maximal bandwidth under continuous bursts of requests. When data objects are accessed in a random (or nonburst) manner, memory bandwidth efficiency dwindles rapidly, so is the packet classification rate.

TSS [18] has a potential for speedy classification, based on decision tree variations (such as tries). A tuple under TSS specifies the involved bits of those fields used for classification, defining its tuple search space for efficient probes by hashing to obtain appropriate rules. To discover the number of bits used to form the tuple (or hash keys), a TSS method builds a separate tree for each one of the header fields. Guiding searches to the decision trees, the cross-product of the result tuple lists signifies the hash tables (or tuples) to be explored via hashing. Therefore, a practical implementation of TSS utilizes both decision trees and hash tables. Enhanced versions of TSS (including binary TSS on columns [22], diagonal-based TSS [12], etc.) are later considered for fast identifying candidate hash tables, where appropriate filters are then probed. While enhanced TSS is promising, it generally suffers from the following limitations: **1**) expensive incremental updates, **2**) poor parallelism, and **3**) limited extensibility to additional fields, in particular, if markers and precomputation for best rules are to be applied [14].

An efficient packet classification algorithm was introduced [2] by hashing flow IDs held in digest caches (instead of the whole classification key comprising multiple header fields) for reduced memory requirements at the expense of a small amount of packet misclassification. Recently, fast and memory-efficient (2D) packet classification using Bloom filters was studied [6] by dividing a rule set into multiple subsets before building a cross-product table for each subset individually. Each classification search probes only those subsets that contain matching rules (and skips the rest) by means of Bloom filters, for sustained high throughput. The mean memory requirement is claimed to be some 32-45 bytes per rule [6]. As will be demonstrated later, our mechanism achieves faster lookups (involving 8-16 hash probes plus four more SRAM accesses, which may all take place in parallel, per packet) and consumes fewer bytes per rule (taking 15-25 bytes per rule).

A dynamic packet filter, dubbed Swift [24], comprises a fixed set of instructions executed by an in-kernel interpreter. Unlike packet classifiers, it optimizes filtering performance by means of powerful instructions and a simplified computational model, involving a kernel implementation. Lately, HyperSplit has been pursued for its superior classification speed and memory usage [25]. Based on recursive space decomposition, HyperSplit has the tree size explosion problem and requires 66 MB storage for 10K ACL rules, in sharp contrast to less than 250 KB under HaRP^* (see Section 4.5).

3 Proposed HaRP Architecture

3.1 Fundamentals and Pertinent Data Structures

As eloquently explained earlier [19], [20], a classification rule is often specified with a pair of communicating networks, followed by the application-specific constraints (e.g., port numbers and the protocol type). Our HaRP exploits this situation by considering the fields on communicating networks and on application-specific constraints separately, comprising two search stages. Its first stage narrows the search range via communicating network prefix fields, and its second stage checks other fields on only entries chosen in the first stage.

3.1.1 Generalized HaRP

As depicted in Fig. 1, the first stage of HaRP comprises a *single* set-associative hash table, referred to as the LuHa table. Unlike typical hash table creation using the object key to determine one single set for an object, our LuHa table aims to achieve extremely efficient table utilization by permitting *multiple candidate sets* to accommodate 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**) adopting *designated prefix length*, DPL: $\{l_1, l_2, \ldots l_i, \ldots l_m\}$, where l_i denotes a prefix length, such that for any prefix P of length w (expressed by $\mathrm{P}|w$) with $l_i \leq w < l_{i+1}$, P is rounded down to $\mathrm{P}|l_i$ before used to hash the LuHa table, and **2**) storing a filter rule in the LuHa table hashed by either its source IP prefix (sip, if not wild carded) or destination IP prefix (dip, if not wild carded), after they are rounded down. Each prefix length ζ, with $\zeta \in$ DPL, is referred to as a *tread*. Given P, it is hashed by treating $\mathrm{P}|l_i$ as an input to a hash function to get a d-bit integer for (2^d) sets in the LuHa table. The round-down prefixes of P are aimed to index hash buckets (i.e., sets) suitable for keeping the prefix. Since 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

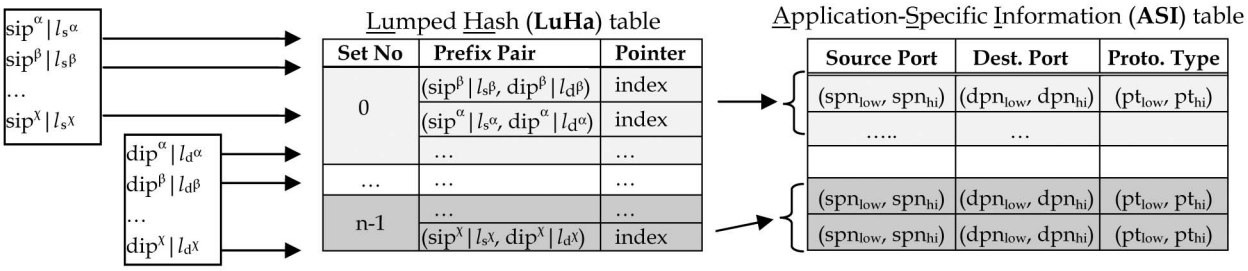

Fig. 1. HaRP classification mechanism comprising a set-associative hash table and an application-specific information table, with a prefix pair stored in any one of hash buckets indexed by round-down prefixes, suitable for parallel lookups.

values can be obtained in parallel for concurrent search over the LuHa table. This classification mechanism results from HaRP during filter rule installation and packet classification search [14], denoted by **HaRP***. The mechanism works because it takes advantage of the "*transitive property*" of prefixes—for a prefix $\mathrm{P}|w, \mathrm{P}|t$ is a prefix of $\mathrm{P}|w$ for all $t < w$, considerably boosting its pseudo set-associative degree. Under the special case where $\mathrm{P}|w$ (with $l_i \le w < l_{i+1}$) is rounded down to $\mathrm{P}|l_b$, 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., $i-1 \le b \le i$), we have a scheme called HaRP^2. It means that the input prefix $\mathrm{P}|w$ can then be stored in hash buckets indexed by either $\mathrm{P}|l_i$ or $\mathrm{P}|l_{i-1}$; namely, there are two candidate hash buckets to choose from. Accordingly, our HaRP* scheme is defined to permit as many candidate hash buckets (in existence) as possible. In summary, HaRP^1 is the basic scheme, where only one hash bucket can keep an object. HaRP^2 means two hash buckets available to hold the hashed object. HaRP^k permits k hash buckets to choose from to keep the hashed object, with HaRP* being the utmost situation where all possible buckets are available for consideration.

A classification lookup for an arrived packet under DPL with m treads involves m hash probes via its source IP address and m probes via its destination IP address, therefore allowing the prefix pair of a filter rule (say, $(\mathrm{P}_s|w_s, \mathrm{P}_d|w_d)$, with $l_i^s \le w_s < l_{i+1}^s$ and $l_i^d \le w_d < l_{i+1}^d$) to be stored in **any one** of the i^s sets indexed by round-down P_s (i.e., $\mathrm{P}_s|\{l_1, l_2, \ldots, l_i^s\}$, if P_s is not a wildcard), or **any one** of the i^d sets indexed by round-down P_d (i.e., $\mathrm{P}_d|\{l_1, l_2, \ldots, l_i^d\}$, if P_s is not a wildcard). Those indexed sets are referred to as *candidate sets*. HaRP* balances the prefix pairs among many candidate sets (each with α entries), making the LuHa table behave like an $(i^s + i^d) \times \alpha$ set-associative design under ideal conditions to enjoy high storage efficiency. Note that HaRP still benefits from elevated set-associativity even when P_s (or P_d) is a wildcard, as long as i^d (or i^s) is larger than 1.

Given DPL with five treads: {1, 12, 16, 24, 28}, for example, HaRP* rounds down the prefix of 010010001111001× ($w = 15$) to 010010001111 ($\zeta = 12$) and 0 ($\zeta = 1$) for hashing. Clearly, HaRP* experiences overflow *only when all candidate sets in the LuHa table are all full*. The following analyzes the LuHa table in terms of its effectiveness and scalability, revealing that for a fixed, small α (say, 4), its overflow probability is negligible, provided that the ratio of the number of LuHa table entries to the number of filter rules is a constant, say ρ.

3.1.2 Effectiveness and Scalability of LuHa Table

From a theoretic analysis perspective, the probability distribution could be approximated by a Bernoulli process, assuming a uniform hash distribution for round-down prefixes. (As round-down prefixes for real filter data sets may not be hashed uniformly, we performed extensive evaluation of HaRP* under publicly available nine real-world data sets, with the results provided in Section 4.3.) The probability of hashing a round-down prefix $\mathrm{P}|l_i$ randomly to a table with r sets equals $1/r$. Thus, the probability for k round-down prefixes, out of n samples (i.e., the filter data set size), hashing to a given set is $\binom{n}{k}(1/r)^k(1-1/r)^{n-k}$. As each set has α entries, we get probability(overflow | k round-down prefixes mapped to a set, for all $k > \alpha) = 1 - \sum_{k=0}^{\alpha}\binom{n}{k}(1/r)^k(1-1/r)^{n-k}$, with $r = (n \times \rho)/\alpha$.

The above expression can be shown to give rise to almost identical results over any practical range of n, for given ρ and α. When $\rho = 1.5$ and $\alpha = 4$, for example, the overflow probability equals 0.1316 under $n = 500$, and it becomes 0.1322 under $n = 100{,}000$. Consequently, under a uniform hashing distribution of round-down prefixes, the set overflow probability of HaRP* holds virtually unchanged as the filter data set size grows, indicating good scalability of HaRP* with respect to its LuHa table. We therefore provide in Fig. 2, the probability of overflowing a set with $\alpha = 4$ entries versus ρ (called the dilation factor) for one filter data set size (i.e., $n = 100{,}000$) only. The overflow probability dwindles as ρ rises (reflecting a larger table).

HaRP^1 achieves better LuHa table utilization, since it permits the use of either sip or dip for hashing, effectively yielding "*pseudo 8-way*" if sip and dip are not wildcards. It selects the less occupied set in the LuHa table from the two

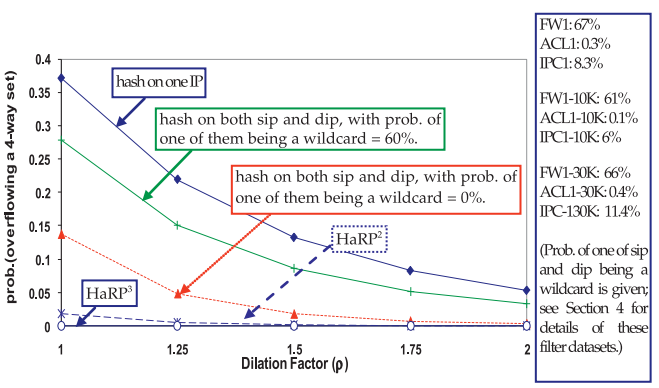

Fig. 2. Overflow probability versus ρ for a 4-way set-associative table.

candidate sets hashed on the nonwild carded sip and dip. The overflowing probability of HaRP^1 can thus be approximated by the likelihood of both candidate LuHa table sets (indexed by sip and dip) being fully taken (i.e., each with four active entries). In practice, the probability results have to be conditioned by the percentage of filter rules with wild carded IP addresses. With a wild carded sip (or dip), a filter rule cannot benefit from using either sip or dip for hashing (since a wild carded IP address is never used for hashing). The set overflowing probability results of HaRP^1 with wild carded IP address rates of 60 percent and 0 percent are depicted in Fig. 2. They are interesting due to their representative characteristics of real filter data sets used in this study (as detailed in Section 4.1; the rates of filter rules with wild carded IP addresses for nine data sets are listed with the right box). With a dilation factor $\rho = 1.5$, the overflowing probability of HaRP^1 drops to 1.7 percent (or 8.6 percent), for the wildcard rate of 0 percent (or 60 percent).

Meanwhile, HaRP^2 and HaRP^3 are seen in the figure to outperform HaRP^1 smartly. In particular, HaRP^2 (or HaRP^3) achieves the overflowing probability of 0.15 percent (or 1.4 E-07 percent) for $\rho = 1.5$, whereas HaRP^3 exhibits the overflowing probability less than 4.8 E-05 percent even under $= 1.0$ (without any dilation for the LuHa table). These results confirm that HaRP^* indeed leads to virtually no overflow with $\alpha = 4$ under $* > 2$, thanks to its exploiting the high set-associative potential for effective table storage utilization. As will be shown in Section 4, HaRP^* also achieves great storage efficiency under real filter data sets, making it possible to hold a whole data set in local cache practically for superior lookup performance.

3.1.3 Application-Specific Information (ASI) Table

The second stage of HaRP involves a table, each of whose entry keeps the values of application-specific filter fields (e.g., port numbers, protocol type) of one rule, dubbed the application-specific information (ASI) table (see Fig. 1). If rules share the same IP prefix pair, their application-specific fields are stored in contiguous ASI entries packed as one chunk pointed by its corresponding entry in the LuHa table. For fast lookups and easy management, ASI entries are fragmented into chunks of a fixed size (say eight contiguous entries). Upon creating a LuHa entry for one pair of sip and dip, a free ASI chunk is allocated and pointed to by the created LuHa entry. Any subsequent rule with an identical pair of sip and dip puts its application-specific fields in a free entry inside the ASI chunk, if available; otherwise, another free ASI chunk is allocated for use, with a pointer established from the earlier chunk to this newly allocated chunk. In essence, the ASI table comprises linked chunks (of a fixed size), with one link for each (sip, dip) pair.

The number of entries in a chunk is made small practically (say, eight), so that all the entries in a chunk can be accessed simultaneously in one cycle, if they are put in one word line (of 1,024 bits) of SRAM modules. This is commonly achievable with current on-chip SRAM technologies. The ASI table requires a comparable number of entries as the filter data set to attain desirable performance, with the longest ASI list containing 30+ entries, according to our evaluation results based on real filter data sets outlined in Section 4.

As demonstrated in Fig. 1, each LuHa table entry is assumed to have 96 bits for accommodating a pair of sip and dip together with their 5-bit length indicators, a 16-bit pointer to an ASI list, and a 6-bit field specifying the ASI list length. Given the word line of 1,024 bits and all entries of a set being put inside the same word line with on-chip SRAM technology for their simultaneous access in one cycle, the set-associative degree (α) of the LuHa table can easily reach 10 (despite that $\alpha = 4$ is found to be adequate in practice).

3.2 Installing Filter Rules

When adding a rule under HaRP, the source (or the destination) IP prefix is used for finding a LuHa entry to hold its prefix pair after rounded down according to chosen DPL, if its destination (or source) IP field is a don't care ($\times$). Under HaRP^*, the number of round-down prefixes for a given nonwildcard IP prefix is up to * (dependent upon the given IP prefix and chosen DPL). When both source and destination IP fields are specified, they are hashed separately (after rounded down) to locate an appropriate set for accommodation. The set is selected as follows: **1**) If a hashed set contains the (sip, dip) prefix pair of the rule in one of its entry, the set is selected (and thus, no new LuHa table entry is created to keep its (sip, dip) pair). **2**) If none hashed set has an entry keeping such a prefix pair, a new entry is created to hold its (sip, dip) pair in the set with least occupancy; if all candidate sets are with the same occupancy, the last candidate set (i.e., the one indexed by the longest round-down dip) is chosen to accommodate the new entry created for keeping the rule. Note that a default table entry exists to hold the special pair of $(\times, \times)$, and that entry has the lowest priority since every packet meets its rule.

3.2.1 Rebalancing Load Distributions in LuHa Table

As (sip, dip) rule pairs are inserted sequentially into the LuHa table and the hashed sets are likely to be different for different keys, the initial load distributions of the LuHa table after inserting all rules may not be ideal. In other words, even though we always pick the least loaded bucket when inserting the ith rule, the load distribution is a function of the past history containing the first $(i-1)$ rules, but not of the entire rule set. Fortunately, the transitive property of prefixes is an independent characteristic, which allows postmortem *migrations* or *rebalance* of (sip, dip) entries in the LuHa table. To this end, a simple migration scheme is followed. For each bucket (i.e., set) whose load is larger than a given threshold, attempts are made to move its elements to other buckets until the load of the bucket falls below the threshold or nothing can be done, as outlined in the pseudocode of Fig. 3. It is understood that optimal results can be obtained by more sophisticated schemes involving optimized table entry *scheduling*, which may call for moving many entries. This work presents only the random behavior of a simple scheme, leaving out other schemes which strive for ideal distributions of hashed elements at high levels of complexity.

The remaining fields of the rule are then put into an entry in the ASI table, indexed by the pointer stored in the selected LuHa entry. As ASI entries are grouped into chunks (with all entries inside a chunk accessed at the same time, in the way like accesses to those set entries in the LuHa table), the rule will find any available entry in the indexed chunk for keeping the contents of its remaining fields. Should no entry be available in the indexed chunk, a new chunk is allocated for use (and this newly allocated chunk is linked to the earlier chunk, as described in Section 3.1).

```
One-hop migration (threshold):
for all sets s in LuHa table {
    k = s.last_element
    while (s.load > threshold && k >=0) {
        Given the (dip, sip) pair in entry k, find a bucket b,
        which is identified by shorter prefixes dip' or sip'
        under DPL such that b has the lightest load.
        If bucket b exists, move entry k from s to b; de-
        crease (/increase) load of s (/b) by 1.
        k--.
    }}
```

Fig. 3. Simple one-hop migration.

3.3 Classification Lookups

Given the header of an incoming packet, a two-staged classification lookup takes place. During the LuHa table lookup, two types of hash probes are performed, one keyed with the source IP address (specified in the packet header) and the other with the destination IP address. Since rules are indexed to the LuHa table using the round-down prefixes during installation, the type of probes keyed by the source IP address involves m hash accesses, one associated with a length listed in $\mathrm{DPL} = \{l_1, l_2, \ldots l_i, \ldots l_m\}$. Likewise, the type of probes keyed by the destination IP address also contains m hash accesses. This way, taking exactly $2m$ hash probes under DPL with m treads, ensures that no packet will be misclassified regardless of how a rule was installed, as illustrated by the pseudocode given in Fig. 4.

Lookups in the ASI table are guided by the selected LuHa entries, which have pointers to the corresponding ASI chunks. The given source and destination IP addresses could match multiple entries (of different prefix lengths) in the LuHa table. Each matched entry points to one chunk in the ASI table, and the pointed chunks are all examined to find the best matched rule. As all entries in one pointed chunk are fetched in one cycle, they are compared concurrently with the contents of all relevant fields in the header of the arrival packet. If a match occurs to any entry, the rule associated with the entry is a candidate for application; otherwise, the next linked chunk is accessed for examination, until a match is found or the linked list is exhausted. When multiple candidate rules are identified, one with the longest matched (sip, dip) pair, or equivalently the lowest rule number, if rules are sorted accordingly, is adopted. On the other hand, if no match occurs, the default rule is chosen.

3.4 Handling Rule Updates and Additional Fields

HaRP admits dynamic filter data sets very well. Adding one rule to the data set may or may not cause any addition to the LuHa table, depending upon if its (sip, dip) pair has been present therein. An entry from the ASI table will be needed to hold the remaining fields of the rule. Conversely, a rule removal requires only to make its corresponding ASI entry available. If entries in the affected ASI chunk all become free after this removal, its associated entry in the LuHa table is released as well.

Packet classification often involves many fields, subject to large dimensionality. As the dimension increases, the search performance of a decision tree-based approach tends to degrade quickly while needed storage may grow exponentially due to the combinatorial specification of many fields. By contrast, adding fields under HaRP does not affect the LuHa table at all, and they only need longer ASI entries to accommodate them, without increasing the number of ASI entries. Search performance hence holds unchanged in the presence of additional fields.

```
Input: Received packet, with dip (destination IP address), sip,
       sport (source port), dport (destination port), proto (pro-
       tocol type)
#define  mask(L)  ~((0x01 <<L) -1)
int  match_rule_id = n_rules;

Hash_Probe (key_select) ::
key = (key_select == USE_DIP) ? dip : sip;
for each tread t in DPL {
   h = hash_func(key&mask(t), t); /* round down & hash */
   for each entry s in hash set LuHa[h] {
      if (PfxMatch((s.dip, dip),  s.dip_length) &&
          PfxMatch((s.sip, sip),  s.sip_length) {
         /* a prefix-pair matched, continue on checking ASI */
       for each asi entry e in the chunk pointed by s.asi {
          if (e.sport_low <= sport <= e.sport_high &&
             e.dport_low <= dport  <= e.dport_high &&
             e.proto_low <= proto <= e.proto_high) {
             /* Match! Choose rule with lower rule number */
               if (match_rule_id >= e.ruleno)
                  match_rule_id = e.ruleno;
}}}}}}

/* Pass 1: hash via dip,  Pass 2:  hash via sip */
  Hash_Probe(USE_DIP);
  Hash_Probe(USE_SIP);
```

Fig. 4. Pseudocode for prefix-pair lookups.

4 Empirical Evaluation and Results

This section evaluates HaRP using the publicly available filter databases, focusing on the distribution results of prefix pairs in the LuHa table. Our evaluation assumes a 4-way set-associative LuHa table design. Overflows are handled by linked lists, and each element in the linked list contains four entries able to hold four additional prefix pairs. HaRP is compared with other algorithms, including the Tuple Space Search [18], BV [11], and HyperCuts [16] in terms of the storage requirement and measured execution time on a multicore AMD server. It is found in our study using both the Broadcom BCM-1480 multicore platform [14] and the AMD server, that HC enjoys substantial memory reduction while considerably quickening the worst-case search time in comparison to other known decision tree-based algorithms, as claimed by the HyerCuts article in [16]. Therefore, HC is chosen as a representative classifier based on the decision tree, for subsequent contrast against HaRP.

To facilitate discussion, we use the notation of HaRP^1_m (or $\mathbf{HaRP}^*_m$) to denote a HaRP^1 (or HaRP^*) scheme with m treads involved in DPL for the rest of this paper.

4.1 Filter Data Sets

Our evaluation employed the filter database suite from the open source of ClassBench [20]. The suite contains three seed filter sets: covering Firewall (FW1), Access Control List (ACL1), and IP Chain (IPC1), made available by service providers and network equipment vendors. By their different characteristics, large synthetic filter data sets of 10K and 30K rules are generated in order to study the scalability of

TABLE 1
Filter Data Sets (# of Filters)

Seed Filters	Synthetic Filters	
FW1 (269)	FW1-10K (9311)	FW1-30K (28529)
ACL1 (752)	ACL1-10K (9603)	ACL1-30K (29916)
IPC1 (1550)	IPC1-10K (9037)	IPC1-30K (29629)

classification mechanisms. For assistance in, and validation on, implementation of different classification approaches, the filter suite is accompanied with traces, which can also be used for performance evaluation as well [23].

In a brief summary of the rule characteristics, the majority of ACL filters contain explicit IP prefixes; on the other hand, many firewall rules have their source IP addresses specified as wildcards. IPC contains prefixes of mixes of lengths. In addition to the above rule data sets, the ClassBench suite [20] provides four other sets of seed rules. We had tried all of them plus large rule databases generated from them, with representative results included in this paper. As general results for smaller seed filters on the Broadcom BCM-1480 multicore platform can be found in our previous publication [14], we emphasize solely the results of large filter data sets here, in particular, those data sets listed in Table 1.

4.2 Selections of DPL Treads

As the LuHa table is consulted $2m$ times for DPL with m treads, the distribution of prefix pairs plays a critical role in hashing performance. Since the hashing keys are round-down prefixes accordingly to $\mathrm{DPL} = \{l_1, l_2, \ldots, l_i, \ldots, l_m\}$, a simple heuristic can be applied to select the treads, namely, treads are selected such that the number of unique prefixes between two consecutive treads are abundant. For example, if $l_{i+1} = 28$ is the current tread, we select the next tread l_i from candidates in $\{27, 26, 25, \ldots\}$ such that the number of unique prefixes (after rounded down to l_i) is maximal. The heuristic works by assuming that the hashing function gives reasonably uniform distributions on unique keys.

When applying this heuristic to the source and destination IP prefixes, two very different DPLs may be generated. Our previous work [14] had always assumed the same DPL applied to both source and destination IP prefixes. In this paper, however, different DPLs fitting the rule characteristics are employed, usually yielding better results.[1]

4.3 Load Distribution in LuHa Table

The hash function is basic to HaRP. In this paper, a simple hash function is developed for use. First, a prefix key is rounded down to the nearest tread in DPL. Next, simple XOR operations are performed on the prefix key and the found tread length as follows:

$$\begin{aligned}
\mathrm{tread} &= \mathrm{find_tread_in_DPL}(\mathrm{length\ of\ the\ prefix_key});\\
\mathrm{pfx} &= \mathrm{prefix_key}\ \mathrm{0xffffffff} \ll (32\text{-}\mathrm{tread})); //\mathrm{round\ down}\\
\mathrm{h} &= (\mathrm{pfx})^{\wedge}(\mathrm{pfx} \gg 7)^{\wedge}(\mathrm{pfx} \gg 15)^{\wedge}\mathrm{tread}^{\wedge}(\mathrm{tread} \ll 5)^{\wedge}\\
&\quad (\mathrm{tread} \ll 12)^{\wedge} \sim (\mathrm{tread} \ll 18)^{\wedge} \sim (\mathrm{tread} \ll 25);\\
\mathrm{set_num} &= (\mathrm{h}^{\wedge}(\mathrm{h} \gg 5)^{\wedge}(\mathrm{h} \ll 13))\%\ \mathrm{num_of_set};
\end{aligned}$$

While outcomes may be improved by using more sophisticated hash functions (such as cyclic redundancy codes, for example), it is beyond the scope of this paper. Instead, we show that a single lumped LuHa table can be effective, and most importantly, HaRP^* works satisfactorily under a simple hash function.

The results of hashing prefix pairs into the LuHa table are shown in Table 2, where the LuHa tables are properly sized. Specifically, the first column of Table 2 is the load, or the number of prefix pairs hashed into a set. Given a 4-way LuHa table design, overflow happens when more than four prefix pairs are to be stored in a bucket. The numbers in the results columns are the percentage of buckets (to the total number of buckets) with the specified loads. For each of the filters, the LuHa table is first provisioned with $\rho = 2$ (*dilated* by a factor of 2 relative to the number of filter rules) for HaRP^1_8, under DPL with eight conveniently selected treads, {1, 8, 12, 16, 20, 24, 28, 32}. The LuHa table size is then aggressively reduced by 100 percent (i.e., $\rho = 1.0$) to show how the single set-associative LuHa table performs with respect to HaRP^*_5 and HaRP^*_8 under fewer or an equivalent number of treads in DPLs, selected according to the heuristic method of Section 4.2 for HaRP^*. The results clearly show that HaRP^1 exhibits no more than 4 percent of overflowing sets in a 4-way set-associative LuHa table. Only the IPC1-10K data set happens to have 10 percent overflow sets, caused partly by the nonideal hash function and partly by the round-down mechanism of HaRP. Nevertheless, the single 4-way LuHa table exhibits good resilience in accommodating hash collisions for the vast majority (96 percent) cases.

When the hash table size is aggressively shrunk (to $\rho = 1.0$), improved and well-balanced results can be observed under HaRP^*. Under HaRP^*_8, all data sets now experience less than 1 percent (or 1.6 percent for IPCs) overflowing sets. Because the opportunities for hashing migration are abundant, table utilization results displayed in the last row show that the LuHa table can be almost fully populated in many cases. Nevertheless, this is achieved with a time cost of $16\ (= 2m)$ hash probes. When the number of treads is lowered to five, results of Table 2 show that more overflowing buckets sprout. The worst case occurs for IPC1-10K, where as many as 30 percent of buckets exhibit overflows. This is due to combination of an imperfect hash function, a small hash table, and the round-down mechanism used by HaRP. Since the hash values are calculated over round-down prefixes, and fewer treads lead to wider strides between consecutive treads, likely to make more prefixes identical in hash calculation after being rounded down. Furthermore, fewer treads in DPL implies a

1. It can be further shown that two DPLs may hold different numbers of treads, that is, lookup time complexity is not always restricted to $2m$ hash probes. Instead, fewer $(m_1 + m_2) < 2m$ probes are possible for one DPL with m_1 treads and another with m_2 treads. Such a scheme could lead to better results because filter rules may have fewer unique destination (or source) IPs specified in relation to a large number of source (or destination) IPs. When two key spaces have different sizes, more treads naturally shall be applied to the larger key space to produce better hashing distributions. Nevertheless, to simplify discussion and save space, we assume in this article that two DPLs always have the same number of m treads.

TABLE 2
Load Distributions of Large Rule Sets in the LuHa Table

	FW1-10K			ACL1-10K			IPC1-10K			FW1-30K			ACL1-30K			IPC1-30K		
Load	HaRP[1]	HaRP*, ρ=1		HaRP[1]	HaRP*, ρ=1		HaRP[1]	HaRP*, ρ=1		HaRP[1]	HaRP*, ρ=1		HaRP[1]	HaRP*, ρ=1		HaRP[1]	HaRP*, ρ=1	
	ρ=2, m=8	m=5	m=8	ρ=2, m=8	m=5	m=8	ρ=2, m=8	m=5	m=8	ρ=2, m=8	m=5	m=8	ρ=2, m=8	m=5	m=8	ρ=2, m=8	m=5	m=8
0	12.4%	0.0%	0.0%	33.6%	1.6%	0.8%	49.6%	13.6%	5.8%	13.2%	0.0%	0.0%	6%	0.0%	0.0%	13%	0.7%	0.1%
1	28.4%	0.0%	0.0%	26.8%	4.8%	3.1%	17.0%	9.4%	6.3%	27%	0.0%	0.0%	23.5%	0.0%	0.0%	25.1%	1.6%	0.1%
2	27.6%	0.0%	0.0%	18.9%	18.2%	16.8%	9.8%	8.7%	7.6%	28%	0.0%	0.0%	40%	0.1%	0.0%	29.3%	3.1%	0.3%
3	17.7%	7.6%	3.6%	10.7%	56.1%	66.2%	7.2%	10.5%	24.5%	18.3%	8.4%	4.9%	26.3%	4.6%	1.2%	22.3%	11.4%	12.5%
4	9.6%	87.8%	95.9%	5.6%	19.1%	13.1%	4.7%	27.4%	54.2%	9.1%	86.7%	93.7%	4.2%	91.0%	97.9%	8.6%	70.3%	86.2%
5	3.1%	4.6%	0.5%	3.3%	0.2%	0.0%	3.4%	29.6%	1.6%	3.1%	4.7%	1.3%	0.08%	4.4%	0.9%	1.4%	13.0%	0.9%
6	1.1%			0.8%			3.2%	0.8%		1%	0.1%		0.01%			0.22%	0.01%	
7	0.2%			0.3%			2.1%			0.27%			0.01%			0.007%		
8+	0.06%						1.5%			0.07%			0.007%					
Utiliz.	48%	98%	99%	34%	70%	72%	31%	72%	80%	48%	98%	99%	50%	99%	99%	48%	94%	97%

Resuts for All Other Large Rule Sets with 30K Rules under m=8 Treads

Load	FW2-30K		ACL2-30K		IPC2-30K		FW3-30K		ACL3-30K		FW4-30K		ACL4-30K		FW5-30K		ACL5-30K	
	HaRP[1]	HaRP*	HaRP[1]	HaRP*	HaRP[1]	HaRP*	HaRP[1]	HaRP*	HaRP[1]	HaRP*	HaRP[1]	HaRP*	HaRP[1]	HaRP*	HaRP[1]	HaRP*	HaRP[1]	HaRP*
	ρ=2	ρ=1	ρ=2	ρ=1	ρ=2	ρ=1	ρ=2	ρ=1	ρ=2	ρ=1	ρ=2	ρ=1	ρ=2	ρ=1	ρ=2	ρ=1	ρ=2	ρ=1
0	14.4%	0.0%	8.2%	0.0%	13.1%	0.0%	19.6%	0.0%	48.4%	2.2%	18.9%	0.0%	43.8%	1.1%	19.8%	0.0%	59.7%	4.3%
1	31.0%	0.0%	31.6%	0.0%	30.8%	0.0%	32.1%	0.0%	18.3%	5.1%	35.0%	0.0%	23.4%	3.6%	33.0%	0.0%	20.1%	17.4%
2	29.2%	0.0%	40.3%	0.0%	29.5%	0.0%	26.4%	0.0%	12.0%	10.4%	26.8%	0.0%	12.6%	8.6%	26.0%	0.0%	9.3%	45.5%
3	16.3%	2.0%	16.5%	3.5%	16.9%	1.2%	13.8%	3.3%	8.6%	51.8%	13.3%	6.7%	7.3%	50.0%	13.7%	3.0%	5.1%	31.8%
4	6.6%	96.1%	3.0%	94.2%	6.7%	97.6%	5.5%	95.0%	6.5%	30.2%	4.3%	92.8%	4.8%	35.9%	5.3%	95.5%	2.9%	1.1%
5	1.9%	2.0%	0.3%	2.3%	2.2%	1.2%	1.9%	1.7%	4.1%	0.1%	1.2%	0.5%	3.3%	0.9%	1.6%	1.5%	1.6%	0.0%
6	0.5%		0.0%		0.6%		0.5%		1.7%		0.3%		2.2%		0.5%		0.7%	
7	0.1%				0.2%		0.1%		0.5%		0.0%		1.2%		0.1%		0.4%	
8+	0.0%				0.0%		0.0%		0.0%		0.0%		1.5%		0.0%		0.1%	
Utiliz.	43.6%	99.5%	43.8%	99.1%	44.8%	99.7%	40%	99.2%	30%	76%	38%	98.3%	31%	80%	39%	99.2%	20%	52%

smaller number of LuHa table candidate sets in which prefix pairs can be stored.

Results for all other large rule sets with 30K rules under $m = 8$ are included in Table 2 as well. Again, HaRP_8^* exhibits no more than 2.3 percent overflowing probabilities for all data sets, under a dilation factor of 1, due to its abundant opportunities for hashing migration. By a slight increase of roughly 4 percent of the total storage with $\rho = 1.1$ (i.e., the LuHa table with its size equal to 1.1 times of the number of rules), we have observed elimination of almost all overflows. As outcomes for all rule data sets follow the same trends, we restrict our subsequent discussion to only the results of those data sets listed in Table 1.

To eliminate overflows, more DPL treads and/or a larger table can be deployed as shown by Fig. 5 for the IPC1-10K data set. The percentage of overflowing buckets (with respect to the total buckets) consistently drops toward 0 when DPL with seven treads is deployed and the hash table is enlarged by 1.2 times. Without enlarging the hash table (for $\rho = 1.0$), DPL needs to contain more than eight treads to avoid bucket overflows. These results indicate that a single lumped set-associative LuHa table for HaRP^* (with load rebalancing) is promising in accommodating prefix pairs of filter rules in a classification data set effectively.

4.4 Search over ASI Table

The second stage of HaRP probes the ASI (application-specific information) table, each of whose entry holds values of all remaining fields, as illustrated in Fig. 1. As described in Section 3.1, we adopt a very simple design which puts rules with the same prefix pairs in an ASI chunk, and hence, the ASI distribution is orthogonal to the selection of DPL and to the LuHa table size. Filter rules are put in the same ASI list only if they have the same prefix pair combination. The LuHa table search has eliminated all rules whose source and destination IP prefixes do not

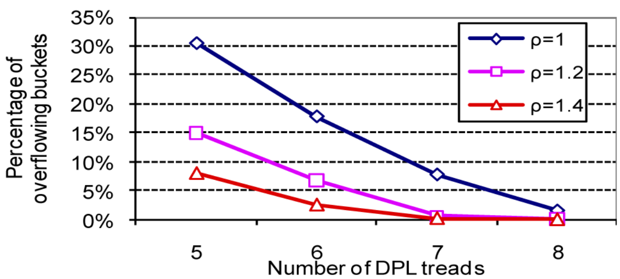

Fig. 5. Overflow rates versus the number of DPL treads under HaRP^* with different hash table sizes for IPC1-10K.

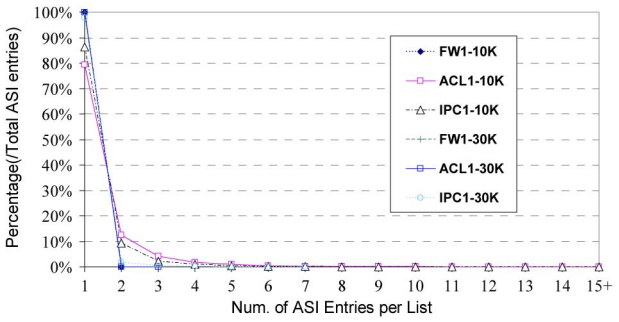

Fig. 6. Length distribution of ASI link lists.

match, pointing solely to those candidate ASI entries for further examination. It is important to find out how many candidate ASI entries exist for a given incoming packet, as they govern search complexity involved in the second stage.

The ASI lists are generally short, as depicted in Fig. 6. Over 98 percent of them have less than five ASI entries each, and hence, linear search is adequate. This is well understood and representatively shown by the cache designs with block granularity and by DRAMs to deliver highest bandwidth for burst accesses on active open pages. Occasionally, a few long lists of some low 30 elements appear for ACL and IPC data sets. By scrutinizing the outcome, we found that this case is caused by a large number of rules specified for a specific host pair leading to a poor case, since those rules for such host pairs fall in the same list. This is believed to represent a situation, where a number of applications at the target host rein accesses from a designated host. Nevertheless, fetching all ASI entries within one chunk at a time (achievable by placing them in the same memort word line, or a cache line) helps to address long ASI lists, if present (since one ASI chunk may easily accommodate eight entries, each with 80 bits, as stated in the next section).

Note that we study the effectiveness of HaRP by using basic linear lists because of their simplicity and empirical results demonstrate that they are adequate for short ASIs. However, it does not preclude the use of advanced techniques and data structures to achieve more optimized design with smaller storage and higher lookup performance. For example, exceedingly long ASI lists can be handled by typical k-ary search for faster results. Given an involved rule field, one may fragment the range of that field into k nonoverlapped segments, so that each rule is kept in a segment when the value of its associated field falls into the segment range. A segment can, in turn, be further partitioned into subsegments, each of which contains a small fraction of rules in an original ASI list. This simple k-way partition can be applied to other involved rule fields as well, yielding fast search over multiple fields. For example, the source port number ranges, $(\mathrm{spn}_{\min}, \mathrm{spn}_{\max})$, and the destination port number ranges, $(\mathrm{dpn}_{\min}, \mathrm{dpn}_{\max})$, of rules on a long ASI list specify regions in the 2D $2^{16} \times 2^{16}$ space defined by the port numbers. It is easy to select two gauge port numbers so that the port space is divided into four subspaces within which rules on the ASI list fall evenly. It is also worth noting that this way of optimization is local to, and tuned for, each long ASI.

TABLE 3
Memory Size and Efficiency for HaRP^* and for HC (with Bucket $\mathrm{Size} = 16$ and Space Factors of Two)

	Total Storage (in KB, or otherwise MB as specified)			Memory Efficiency		
	HaRP* (ρ=1)	HaRP* (ρ=1.2)	Hyper-Cuts	HaRP* (ρ=1)	HaRP* (ρ=1.2)	Hyper-Cuts
FW1-10K	219	240	25 MB	0.83	0.76	0.007
ACL1-10K	225	248	279	0.83	0.75	0.670
IPC1-10K	220	238	649	0.80	0.74	0.270
FW1-30K	672	736	185 MB	0.83	0.76	0.003
ACL1-30K	705	771	818	0.83	0.76	0.710
IPC1-30K	706	764	11 MB	0.82	0.75	0.050

4.5 Storage Requirements and Memory Efficiency

Table 3 displays the consumed storage size and the memory efficiency of different methods, where the dilation factor (ρ) refers to the ratio of the number of LuHa table entries to the data set size. Memory efficiency is defined as the ratio between the minimal storage required to keep all filter rules (as in a linear array of rules, each takes up 20 bytes) and the total storage of constituent data structures (which include the provisioned but not occupied entries in the LuHa table of HaRP), namely,

$$\text{Memory efficiency} = \frac{Minimum\ memory\ for\ all\ rules}{Total\ memory\ used\ (and\ provisioned)}.$$

It is clear that an ideal solution has memory efficiency equal to 1.0; the closer it is toward 1.0, the better.

Under HaRP^*, each LuHa entry is 12-byte long, comprising two 32b IP address prefixes, two 5b prefix length indicators, a 16b pointer to the ASI table, and a 6b integer indicating the length of its associated linked list. Each ASI entry needs 10 bytes to keep source and destination port ranges and the protocol type, plus two bytes for the rule number (i.e., the priority). Under the HC implementation (obtained from [13]), each node is assumed to take 4 bytes. All rules are stored in consecutive memory locations and each rule requires 20 bytes.

The results in Table 3 clearly demonstrate that HaRP^* is much more memory-efficient than HC. Often, hash tables have low memory efficiency and are over-provisioned in order to achieve good hashing results (without many conflicts). Under HaRP^*, its large "pseudo set-associative" degree makes the dilation factor possible to be as low as 1, given that each prefix pair can be stored in any one of multiple candidate LuHa table sets, effectively alleviating the over-provision shortcomings usually associated with hash table-based methods. HaRP^* consistently enjoys high LuHa table utilization (i.e., ratio of the total number of active entries to that of total entries), as unveiled in Table 3.

In contrast to a hash table-based method, decision tree search (such as that under HC) tries to eliminate as many rules as possible for further consideration at each search step. Unlike the LuHa table, where all rules with identical (sip, dip) pair but distinct other fields share one single entry therein, the decision tree data structure keeps all rules separately without consolidation. This naturally leads to lower memory efficiency for decision tree-based classifiers, whose search times are also impacted adversely. Also

decision-tree- or trie-based algorithms, such as HyperCuts, exhibit rather unpredictable outcomes because the size of a trie largely depends on if data sets have comparable prefixes to enable trie contraction; otherwise, a trie can grow quickly toward full expansion. Those listed outcomes in Table 3 generally indicate what can be best achieved by the cited HyperCuts technique, with its refinement options (including *rule overlapping* and *rule pushing*) all turned on for the best results [16]. Also, this implementation assumes a 64-bit EPB (Extended Path Bitmap for at most 64 child nodes), yielding a constant node size immune to the "dead space" pointers; hence, the results of Table 3 reveal the lower bounds for HC under memory-conserving implementation.

4.5.1 Effects of Wildcard and Low-Explicit Rules on HC

The memory size for HC swings wildly for different data sets. For the firewall application, its plentiful wildcard IP addresses cause explosion of the decision tree. Generally speaking, the less specific the filter rules are, the larger the decision tree becomes, as expected and shown. This is due to the fact that wildcard rules end up in many leaves, enlarging storage dramatically. Consider k rules with their source IP being 16-bit prefixes in the form of $x.y.0.0|16$. Assume all other fields of the rules are ignored, with expansion performed on the source IP address, so that the path from the root to Node N_λ is determined exactly by the prefix string of $x.y$. Obviously, all k rules hold true for the eventual subtree rooted at N_λ, and such a subtree can have up to 2^{16} leaves. If k is small, the expansion may stop and linear search over the k rules during lookups may suffice. On the other hand, for a large k, the expansion continues and the decision tree size can grow rapidly, because the source IP prefix can no longer be a factor helpful to differentiate and eliminate rules for further expansion consideration. This signifies that with most explicit rules, the ACL data set invokes the least amount of memory. When the number of explicit rules drops slightly (as in IPC), the tree size increases rapidly.

4.5.2 Effects of Algorithmic Parameters on HC

As a heuristic parameter, *SF* (space factor) governs the expansion of the decision tree at each node so that tradeoffs between space and time can be made. As suggested in HC [16], the total number of splits at a node is bounded by $SF \times \sqrt{N}$, where N is the number of rules holding true at that node.

In general, a larger *SF* leads to a wider and shallower decision tree, but it does not exhibit a conclusive trend on the total numbers of tree nodes, rules stored, and rules pushed and kept in intermediate nodes, as listed respectively in the fourth, the fifth, and the last columns of Table 4. The matter is complicated by all three factors: *SF*, selection of dimension cuttings, and specificity of the filter rule data sets (not just *SF* alone). There is really no consummate way to guide the construction of decision trees for minimal storage and fast lookups within a reasonable computing time. Most importantly, we have observed in our studies that the storage size and the lookup speed can be two conflicting goals under HC. For example, *common rule pushing* that pushes rules holding true for all child nodes to their common parent node is an important technique for HC to help contain forward tree expansion. However, it can cause adversely long lists of rules stored in the intermediate (nonleaf) tree nodes. Those pushed rules make lookup performance crawl to a halt because every lookup requires inspecting those pushed rules. Such an undesirable consequence results from making tradeoffs between storage and time under the decision tree method. It also enlightens the potential pitfalls, where rule pushing trades lookup performance for storage space.

TABLE 4
Breakdowns of Storage for HC

	SF	Tree Depth	Total Nodes	Total Stored Rules	Total Pushed Rules
FW1-30K	2	10	7,204,641	39,249,464	2,228,255
	4	7	3,828,209	33,671,114	451,788
	8	6	6,608,495	45,130,564	264,169
ACL1-30K	2	7	8,993	47,795	546
	4	6	14,458	63,224	507
	8	5	18,686	70,301	503
IPC1-30K	2	13	439,226	2,049848	87,571
	4	11	185,753	727,936	24,423
	8	8	306,160	1,010,830	26,523

5 Scalability and Lookup Performance on Multicores

As each packet can be handled independently, packet classification suits a multicore system well [5]. Given a multicore processor with *np* cores, a simple implementation may assign a packet to any available core at a time so that *np* packets can be handled in parallel by *np* cores.

This section presents and discusses performance and scalability of HaRP^* in comparison with those of HyperCuts. By the results of Table 2, basic HaRP^1 can never outperform HaRP^*, and studies thus emphasize lookup performance under HaRP^* here. Two configurations, HaRP^*_5 and HaRP^*_8, with the LuHa table sized to a dilation factor of $\rho = 1.0$, are considered to reflect the results given in Table 2.

For gathering measures of interest on our multicore platform, our HaRP code was made multithreaded for execution. With the source code for HC implementation taken from the public source [23], we closely examined and polished them by removing unneeded data structures and also replacing some poor code segments with their efficient equivalences in order to get best performance levels of those referenced techniques. All those program codes were made multithreaded to execute on the same multicore platform, with their results presented in next sections.

5.1 Data Footprint Size

Because search is performed on each hashed set sequentially by a core, it is important to keep the footprint small so that the working data structure can fit into its caches, preferably the L1 (level-one) or L2 cache dedicated to a core. According to Table 2, HaRP^* requires the least amount of memory provisioned. By our measurement, it requires less than 800 KB to keep 30K data sets, making it quite possible to hold the entire data structure in the L2 cache of a today's core. This advantage in containing the growth of its data footprint size as the number of rules increases is unique to

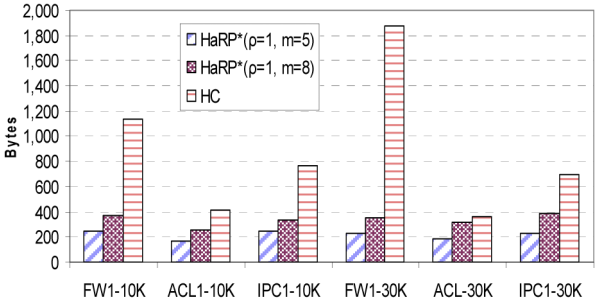

Fig. 7. Average number of bytes fetched per lookup.

TABLE 5
Search Performance (in Terms of Mean Number of Entries) Per Lookup under HaRP^* with $\rho = 1$

	HaRP*						HC	
	LuHa Search mean number of prefix pairs				ASI Search mean number of entries		Nodes	Rules
	m= 5		m = 8		m = 5	m = 8		
	Check	Match	Check	Match	Check	Check		
FW1-10K	28	1.1	43	1.0	1.1	1.0	5.7	47
ACL1-10K	17.8	1.3	28.5	1.2	2.0	1.9	8.2	12.6
IPC1-10K	26.3	1.7	37.3	1.7	2.6	2.7	8.2	19
FW1-30K	24.4	1.1	40.0	1.1	2.3	2.3	6.2	87
ACL1-30K	21.4	1.0	38.0	1.0	1.0	1.0	4.9	13
IPC1-30K	26.3	1.1	45.8	1.1	1.2	1.2	4.5	31.8

HaRP^* (and not shared by any prior technique), rendering it particularly suitable for multicore implementation to attain high performance.

5.1.1 Average Footprint

The behavior of HaRP^* driven by the traces provided with filter data sets [23] was evaluated to obtain the first order of measurement on the data footprint for lookups. Fig. 7 depicts the mean number of bytes fetched per packet lookup, a conventionally adopted metric for comparing classification methods [16]. It is clearly shown that HaRP^*_5 touches fewer data than HaRP^*_8. This is because HaRP always probes $2 \times \mathrm{m}$ LuHa sets (irrespective of the data set size), and thus keeping a small m is important. Although results in Table 2 show that HaRP^*_5 causes more overflowing buckets than HaRP^*_8, the working data set tells an interestingly different story. HaRP^*_5 saves six hash probes to the LuHa table per classification lookup in comparison to HaRP^*_8 (namely, 10 probes to more occupied sets versus 16 probes to less occupied sets).

Table 5 provides advanced insight into the search performance under HaRP^*_5 and HaRP^*_8, involving accesses to both LuHa table and ASI table. The mean numbers of matched entries differ only a little, as listed in Table 5, where the first and the third result columns give the average numbers of prefix pairs inspected per packet classification under HaRP^*_5 and HaRP^*_8, respectively. Clearly, HaRP^*_5 touches and inspects fewer prefix pairs than HaRP^*_8, due to fewer hash probes. The second and the fourth column contain the average numbers of prefix pairs matched. On average, less than two prefix pairs match in the LuHa table per classification lookup, signifying that the two-stage lookup procedure of HaRP^* is effective. Finally, the 5th and 6th result columns list the mean numbers of ASI tuples inspected per matched prefix pair. Since the mean numbers are small, linear search as being performed in this work may suffice.

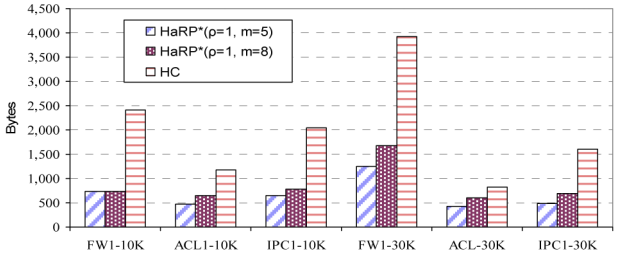

Fig. 8. Worst-case number of bytes accessed.

In summary, results depicted in Fig. 7 and Table 5 reveal an interesting fact that *fewer treads with manageable LuHa table overflows may prevail under a software implementation*. As we will demonstrate in the next section, lookup performance is affected predominately by the number of hash probes or the data footprint that impacts the cache behavior. On the other hand, *hardware support for HaRP* to allow parallel probes to the candidate LuHa sets may benefit from more DPL treads*. With adequate memory bandwidth, all hash probes can be launched in parallel, as depicted by the model of Fig. 1. When overflows present, they complicate the (hardware) table management and slow down classification because now multiple cycles are required to chase overflowed entries for completing hash lookups. And hence, selection of DPL treads to minimize LuHa table overflows or maximize parallelism is a practical implementation tradeoff requiring careful evaluation.

Fig. 7 also illustrates that HaRP^* consistently enjoys much lower average footprint per lookup than HyperCuts. Particularly for the firewall rule data sets, HC fetches five to nine times of data than HaRP^*_5, clearly putting pressures on the cache systems of modern processors and thereby lowering their performance (as to be shown in the next section). The second to the last column of Table 5 displays that the average search depth of the decision tree is between 4.5 and 8.2. As expected, it is not deep when multiple bits (cuts) are considered at each tree node. However, to truly understand how HC may perform, the last column reveals that many rules are inspected per classification lookup. This is because many rules are pushed to the intermediate nodes as shown by the last column of Table 4, especially for the Firewall data sets.

5.1.2 Worst Cases

Besides demonstrating a smaller mean data footpriont, the deterministic procedure to probe $2 \times \mathrm{m}$ LuHa sets under $\mathrm{HaRP}^*_{\mathrm{m}}$ also yields more stable worst-case results across various rule data sets, as shown in Fig. 8. For HyperCuts, the results fluctuate, depending on the depth of the decision tree and the number of rules which are pushed up from the leaves and stored at the intermediate nodes. As we have described before, pushing common rule subsets upward the trie structure is an important technique for saving storage in HC [16]. While keeping common rules at nonleaf nodes saves

TABLE 6
Measured Throughput Results on an AMD System (in Relative Scale to HC under One Core)

	1 core			2 cores			4 cores		
	HC	HaRP*, ρ=1		HC	HaRP*, ρ=1		HC	HaRP*, ρ=1	
		m = 5	m = 8		m = 5	m = 8		m = 5	m = 8
FW1-10K	1	2.47	1.68	1.87	5.24	3.27	4.00	9.58	6.40
ACL1-10K	1	3.19	2.29	1.99	6.66	4.30	4.01	11.74	8.62
IPC1-10K	1	2.91	2.01	1.93	6.00	4.22	3.99	11.09	8.02
FW1-30K	1	3.90	2.55	1.99	7.50	5.01	3.92	14.30	9.52
ACL1-30K	1	1.68	1.13	1.93	3.52	2.27	4.01	6.34	4.28
IPC1-30K	1	2.20	1.45	1.99	4.26	2.95	4.07	8.59	5.26

storage by avoiding replicas at the leaves, this optimization heuristic requires inspection of rules kept at the nonleaf nodes upon traversing the trie during lookups, possibly leading to a large data footprint, as revealed in Fig. 8.

5.2 Performance on AMD Opteron Server

While data footprint results presented in the last section might unveil relative performance of different classification techniques (given the memory system is generally deemed as the bottleneck), computation steps or the mechanisms involved in dealing with the data structures are equally important and have to be taken into consideration. To arrive at more accurate evaluation, we executed all classification programs on an AMD 4-way 2.8 GHz Opteron system with 1 MB cache for each core. Throughput performance is measured, with the results for HaRP and HC listed in Table 6. Values in the table are all relatively scaled to **one thread** HyperCuts performance, which is shown as a reference of one in the second column for clear and system configuration-independent comparison. The second to the fourth columns include the single core results. The fifth to the seventh columns contain the speedups relative to one thread HyperCuts performance, with the rest for results under four cores.

When the number of threads rises from one to two and then to four, HC shows nearly linear or superlinear scalability (in terms of raw classification rates) with respect to the number of cores. The superlinear phenomenon may be due to the fact that one tread can benefit from a warmed up cache by another thread, or marginal errors occurred during statistic collection. This scalability trend indeed exists for HaRP^* as well, because packet classification is inherently parallel, as expected. Overall, HaRP demonstrates much higher throughput than HyperCuts for all data sets. On a per core basis, HaRP^*_5 (or HaRP^*_8) consistently delivers 1.7 to 3.6 (or 1.1 to 2.5) times improvement over HC. According to the average footprint results given in Fig. 7, HC clearly puts more pressure on the cache system. With random touching of cache lines, it is anticipated that HC will perform worse than HaRP^*. However, while data footprint can indeed give first-order estimation on how well a technique could perform, the code path during execution is equivalently critical. By inspecting the disassembled HC code, we found that the code path for HC could be rather long. For example, at each step traversing the decision tree, the number of bits extracted from a field needs to be determined, and next the extracted bits are used to calculate the location of the next child in the decision tree. In brief, the total number of splits (i.e., children) of a node is specified by $NC = \Pi_i nc(i)$, where $nc(i)$ is the number of cuts performed on the ith header field. During search, $\log_2(nc(i))$ bits are extracted from the appropriate positions in the ith field; assuming the decimal value represented by the extracted bits is v_i, the number of child positions in the linear array covering the *NC* space is then expressed by $\sum_{i=1}^{D-1} v_i \times \Pi_{j=i+1}^{D} nc(j) + v_D$ for D dimensions. These operations seem simple, but in fact, they can take hundreds of cycles to complete, causing a significant performance loss, as observed.

For HaRP^*, it is worth noting how the performance level changes according to the number of DPL treads. Due to fewer hash probes and the smaller data footprint (see Fig. 7) enjoyed by HaRP^*_5, it performs better than HaRP^*_8. It stems from the fact that HaRP^*_5 employs DPL with five treads, as opposed to eight treads for HaRP^*_8. This brings the number of hash probes per lookup from 16 down to 10, incurring less hashing overhead. Most importantly, HaRP^*_5 is expected to be more caching-friendly than HaRP^*_8, because accessing prefix pairs located in 10 sets should enjoy better caching locality than prefix pairs spread across 16 sets.

The above results raise an interesting question about choosing the number of DPL treads under HaRP^*. If fewer treads lead to less hash probes, does it always deliver higher performance? The answer is a complex matter, involving many factors, like the number of overflows (or the number of elements hashing to the same buckets), the overhead in hashing calculation, and the degree of parallelism of capability for inspecting multiple elements simultaneously. Clearly, as shown in Tables 2 and 5, HaRP^*_5 involves fewer hash probes to more congested hash buckets in the LuHa table. By contrast, HaRP^*_8 incur more hash probes, but to lightly-filled buckets. While HaRP^*_5 is observed to outperform HaRP^*_8, it is not realistic to expect the trend to continue favoring fewer and fewer treads. At a certain point, when the number of DPL treads is too few to yield good hashing distribution (due to rounding down prefixes under HaRP to let many prefixes mapped to the same hash bucket), the hash table will degenerate to a few overly congested buckets, slashing its performance. Furthermore, when the system has adequate bandwidth and is capable of launching all hash probes in parallel, a larger DPL is favorable.

6 Lookup Access Analysis

In the previous section, the measured results have shown that HaRP^* outperforms decision tree-based methods represented by HC [16]. Here, an analytical model is built to capture the fundamental property of HaRP^* relative to HC [16] under different system capabilities, in particular, when both methods are realized by hardware using on-chip SRAM, off-chip SRAM, or both together.

6.1 Memory System Consideration

HaRP^* is suitable for high performance hardware implementation, 2m hash probes to the LuHa table possibly executed in parallel. Consider a LuHa table design under

today's 65 nm ASIC technology, realized by 64 on-chip $256 \times 144\text{b}$ SRAM blocks (each with $46339\ \mu\text{m}^2$), for a total of some $3\ \text{mm}^2$; this small-area table suffices to keep 30K filtering rules. When 2m ($= 16$ under $\text{m} = 8$) hash probes are issued to the 64 blocks, all accesses are in parallel except for those to the same memory blocks (which are then serialized). While the worst-case is bounded by 2m cycles, our extensive simulation reveals the the worst-case to take 5 cycles in practice, with the mean of 1.38 cycles. With a scheduler to orchestrate filtering requests made by multiple interfaces, an average one-lookup-per-cycle rate is attainable using the memory system. At 500 MHz, for example, such a LuHa table design can serve O(100 MPPS) lookups. It is also feasible to realize the ASI table by on-chip RAMs, given that most ASI lists inspected after matches reported in the LuHa table are short, per the results in Table 5.

Additionally, our HaRP* design matches the commodity RAM technology well. For today's advanced QDR SRAM, as an example, it delivers the maximum throughput when continuous requests are made to the device every other cycle. Because addresses for the 2m sets in the LuHa table are calculated simultaneously, the 2m accesses can be issued to QDR SRAM one-by-one in a truly pipelined manner without interruption. After a prefix pair is found, its ASI list is fetched in the same way. Therefore, the HaRP* design can really benefit from the maximum potential bandwidth offered by QDR SRAM. This is in sharp contrast to attempting a pipelined decision tree, because upon traversing down a tree, each step then requires to fetch the node contents, select the correct bits from fields to make cuts as specified in the node contents, and calculate the next nonleaf node address before issuing a request. It is clearly challenging to complete all the tasks in one cycle without causing pipeline bubbles for a high performance design. Furthermore, the memory size required for holding nodes at each level grows exponentially toward the bottom of the trie.

6.2 The Analytical Model

To provide insights into the advantages gained potentially by the parallel nature of HaRP* under different hardware characteristics, we have recourse to an analytical model for its generality. The model contains the following four parameters:

1. Lag, $\alpha = t_n/t_h$, reflecting the ratio between the average time to move through a decision tree node (t_n) and the average time to fetch and check a LuHa entry (t_h),
2. Delay, $\beta = t_f/t_h$, being the ratio between the mean time to access and inspect a filter rule under a decision tree-based method (t_f) and the mean time to access and inspect an ASI entry under HaRP (which is the same as t_h),
3. Parallelism, γ, denoting the number of LuHa table entries and ASI entries being fetched and processed in parallel; the number of rules checked in parallel at each node in the decision tree also equal to γ, and
4. Bandwidth, ω, referring to the number of such elements as LuHa entries, ASI elements, and filter rules stored in consecutive locations that can be fetched simultaneously due to a wide memory data path. Note that ASI entries are stored in consecutive memory locations, β captures the possible efficiency gain experienced by HaRP over a decision tree-based design, where filter rules, stored and indexed in an array, are mostly accessed in a random and scattered fashion.

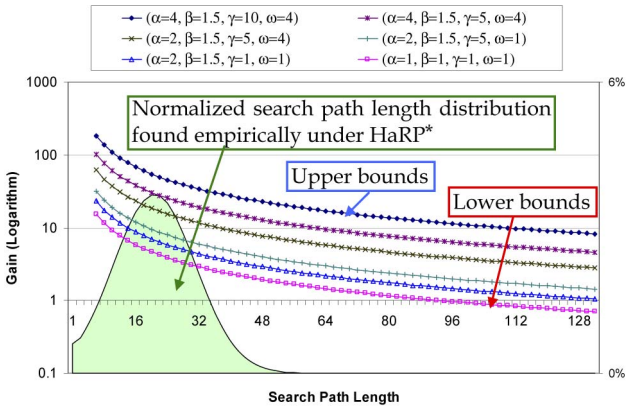

Fig. 9. Throughput gain by HaRP* over its decision tree-based counterpart, whose average search depth and mean number of inspected rules are 6 and 87, respectively, under the FW1-30K rule data set.

In addition, the storage-saving technique for decision tree-based design by pushing common rules upwards (to avoid holding the same rules repeatedly in all of its child nodes) requires inspection of rules kept at the nonleaf nodes while traversing the trie during lookups, often involving a much longer trie traversal time.

The expected packet lookup gain (G) enjoyed by HaRP* over its decision tree-based counterpart can then be expressed as

$$G = \frac{E[T^T]}{E[T^H]} = \frac{t_n \times E\left[L_n^T\right] + t_f \times (1/\gamma\omega) \times E\left[L_f^T\right]}{t_h \times (1/\gamma\omega) \times \left[2m \times \left(E\left[L_n^H\right] + E\left[L_f^H\right]\right)\right]}$$
$$= \frac{\alpha \times E\left[L_n^T\right] + \beta \times (1/\gamma\omega) \times E\left[L_f^T\right]}{(1/\gamma\omega) \times \left[2m \times \left(E\left[L_n^H\right] + E\left[L_f^H\right]\right)\right]}$$
$$= \frac{\alpha\gamma\omega \times E\left[L_n^T\right] + \beta \times E\left[L_f^T\right]}{2m \times \left(E\left[L_n^H\right] + E\left[L_f^H\right]\right)},$$

where $E[T^T]$ (or $E[T^H]$) denotes the expected lookup time per packet under the decision tree-based method (or HaRP*), $E[L_n^T]$ and $E[L_f^T]$ refer, respectively, to the mean search depth in a decision tree and to the average number of rules fetched and inspected per packet lookup, while $E[L_n^H]$(or $E[L_f^H]$) is the mean number of matched prefix pairs (or of ASI entries checked) per hash probe under HaRP. In other words, G under HaRP* is given by a function of average search path length (in terms of the number of visited prefix pairs and ASI entries) for varying degrees of delay (α, β), hardware parallelism (γ), and the bandwidth ratio (ω).

Fig. 9 demonstrates the HaRP* throughput gain for various hardware characteristics, ranging from most aggressive (the upper bound) to least aggressive (the lower bound) scenarios. Under a decision tree method, 1) the operations performed for descending the tree are more

complex than hashing and linear search operations under HaRP*, and 2) memory bandwidth efficiency degrades (as discussed in Section 2.1), expecting to yield α times longer, with $1 \leq \alpha \leq 4$. A decision tree method fetches the filter rules via indirect indices, leading to a delay ratio of β, with $1 \leq \beta \leq 1.5$. Aggressive HaRP performs $2m$ ($\gamma = 10$, if m is 5) hash probes in parallel, and fetches $\omega = 4$ prefix pairs (or ASI entries) per memory access. Hence, the most aggressive design corresponds to $\alpha = 4, \beta = 1.5, \gamma = 10$, and $\omega = 4$, reflecting upper bounds. On the other hand, the least aggressive setting of $\alpha = \beta = \gamma = \omega = 1$ gives lower bounds. The shaded area in Fig. 9 displays the empirically obtained distribution of the search path length under HaRP, marking expected gains for HaRP with varying hardware characteristics. In essence, the largest gain of HaRP* is expected to come from parallelism and wide memory support (enabling multiple hash probes and simultaneous accesses to multiple prefix pairs and ASI entries). This signifies the advantages of keeping simple and efficient data structures, like the LuHa table and the ASI lists under HaRP*.

7 Concluding Remarks

Packet classification is essential for most network system functionality and services, but it is complex, since it involves comparing multiple fields in a packet header against entries in the filter data set to decide the proper rule to apply for handling the packet [8]. This paper has considered a rapid packet classification mechanism realized by HaRP able to not only exhibit high scalability in terms of both the classification time and the SRAM size involved, but also effectively handle incremental updates to the filter data sets. Based on a single set-associative LuHa hash table (obtained by lumping a set of hash table units together) to support two-staged search, HaRP promises to enjoy better classification performance than its known software-oriented counterpart, because the LuHa table narrows the search scope effectively based on the source and the destination IP addresses of an arrival packet during the first stage, leading to fast search in the second stage. With its required SRAM size lowered considerably, HaRP makes it possible to hold entire search data structures in the local cache of each core within a contemporary processor, further elevating its classification performance.

The LuHa table admits each filter rule in a set (i.e., bucket) with lightest occupancy among all those indexed by hash(round-down sip) and hash(round-down dip), under HaRP*. Utilizing the first ζ bits of an IP prefix with l bits (for $\zeta \leq l, \zeta \in$ DPL) as the key to the hash function (instead of using the original IP prefix), this way lowers substantially the likelihood of set overflow, which occurs only when all indexed sets are full, attaining high SRAM storage utilization. It also leads to great scalability, even for small LuHa table set-associativity (of 4) and a small table dilation factor (say, $\rho = 1.0$ or 1.2). Our evaluation results have shown that HaRP* with the set associative degree of 4, generally experiences very rare set overflow instances.

Empirical assessment of HaRP has been conducted on an AMD 4-way server with the 2.8 GHz Opteron processor. A simple hashing function was employed for our HaRP implementation. Extensive measured results demonstrate that HaRP* outperforms HC [16] to exhibit throughput of 1.7 to 3.6 times, on an average, under the six databases examined, when its LuHa table is with $\rho = 1.0$ and there are five DPL treads. Besides its efficient support for incremental rule updates, our proposed HaRP also enjoys far better classification performance than previous software-based techniques.

Note that theoretically pathological cases may occur despite encouraging pragmatic results by $\rho = 1.0$, as we have witnessed in this study. For example, a large number of (hosts on the same subnet with) prefixes P$|w$ can differ only in a few bits. Hence, those prefixes can be hashed into the same set after being rounded down, say from P$|w$ down to P$|l_i$, for $l_i \leq w < l_{i+1}$, under HaRP*. There are possible ways to deal with such cases and to avoid overwhelming the indexed set. A possible way is to use one and only one entry to keep the round-down prefix P$|l_i$, as opposed to holding all P$|w$'s in individual entries under the current design. Subsequently, the $(w - l_i)$ round-down bits can form a secondary indexing structure to provide the differentiation (among rules specific to each host) and/or the round-down bits can be mingled with the remaining fields of the filter rules. Thus, each stage narrows the search range by small and manageable structures. These possible options are being explored.

Acknowldgments

An earlier version of this manuscript was presented at the 2009 USENIX Annual Technical Conference (USENIX '09), June 2009.

References

[1] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proc. 20th Ann. Joint Conf. IEEE Computer and Comm. Soc. (INFOCOM '01),* pp. 1454-1463, Apr. 2001.

[2] F. Chang et al., "Efficient Packet Classification with Digest Caches," *Proc. Third Workshop Network Processors and Applications (NP-3),* Feb. 2004.

[3] W.T. Chen, S.B. Shih, and J.L. Chiang, "A Two-Stage Packet Classification Algorithm," *Proc. 17th Int'l Conf. Advanced Information Networking and Applications (AINA '03),* pp. 762-767, Mar. 2003.

[4] Y.H. Cho and W.H. Magione-Smith, "Deep Packet Filter with Dedicated Logic and Read Only Memories," *Proc. 12th IEEE Symp. Field-Programmable Custom Computing Machines,* pp. 125-134, Apr. 2004.

[5] H. Cheng et al., "Scalable Packet Classification Using Interpreting a Cross-Platform Multi-Core Solution," *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '08),* pp. 33-42, Feb. 2008.

[6] S. Dharmapurikar et al., "Fast Packet Classification Using Bloom Filters," *Proc. IEEE/ACM Symp. Architectures for Networking and Comm. Systems (ANCS '06),* pp. 61-70, Dec. 2006.

[7] Q. Dong et al., "Wire Speed Packet Classification without TCAMs: A Few More Registers (and a Bit of Logic) Are Enough," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '07),* pp. 253-264, June 2007.

[8] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," *Proc. ACM Ann. Conf. Special Interest Group on Data Comm. (SIGCOMM '99),* pp. 147-160, Aug./Sept. 1999.

[9] P. Gupta and N. McKeown, "Classifying Packets with Hierarchical Intelligent Cuttings," *IEEE Micro,* vol. 20, no. 1, pp. 34-41, Jan./Feb. 2000.

[10] A. Kennedy, X. Wang, and B. Liu, "Energy Efficient Packet Classification Hardware Accelerator," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS '08),* pp. 1-8, Apr. 2008.

[11] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," *Proc. ACM SIGCOMM '98,* pp. 191-202, Aug./Sept. 1998.
[12] F.-Y. Lee and S. Shieh, "Packet Classification Using Diagonal-Based Tuple Space Search," *Computer Networks,* vol. 50, pp. 1406-1423, 2006.
[13] J. van Lunteren and T. Engbersen, "Fast and Scalable Packet Classification," *IEEE J. Selected Areas in Comm.,* vol. 21, no. 4, pp. 560-571, May 2003.
[14] F. Pong and N.-F. Tzeng, "Hashing Round-Down Prefixes for Rapid Packet Classification," *Proc. USENIX Ann. Technical Conf. (USENIX '09),* June 2009.
[15] D. Shah and P. Gupta, "Fast Incremental Updates on Ternary-CAMs for Routing Lookups and Packet Classification," *Proc. Eighth Ann. IEEE Symp. High-Performance Interconnects (Hot Interconnects '08),* pp. 145-153, Aug. 2000.
[16] S. Singh et al., "Packet Classification Using Multidimensional Cutting," *Proc. ACM SIGCOMM '03,* pp. 213-114, Aug. 2003.
[17] H. Song and J.W. Lockwood, "Efficient Packet Classification for Network Intrusion Detection Using FPGA," *Proc. ACM/SIGDA 13th Int'l Symp. Field Programmable Gate Arrays (FPGA '05),* pp. 238-245, Feb. 2005.
[18] V. Srinivasan, S. Suri, and G. Varghese, "Packet Classification Using Tuple Space Search," *Proc. ACM SIGCOMM '99,* pp. 135-146, Aug./Sept. 1999.
[19] D.E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys,* vol. 37, no. 3, pp. 238-275, Sept. 2005.
[20] D.E. Taylor and J.S. Turner, "ClassBench: A Packet Classification Benchmark," *Proc. 24th IEEE INFOCOM '05,* Mar. 2005.
[21] G. Wang and N.-F. Tzeng, "TCAM-Based Forwarding engine with Minimum Independent Prefix Set (MIPS) for Fast Updating," *Proc. IEEE Int'l Conf. Comm. (ICC '06),* June 2006.
[22] P. Warkhede, S. Suri, and G. Varghese, "Fast Packet Classification for Two-Dimensional Conflict-Free Filters," *Proc. 20th IEEE INFOCOM '01,* pp. 1434-1443, Apr. 2001.
[23] Washington Univ., "Evaluation of Packet Classification Algorithms," http://www.arl.wustl.edu/~hs1/PClassEval.html. Feb. 2007.
[24] Z. Wu, M. Xie, and H. Wang, "Swift: A Fast Dynamic Packet Filter," *Proc. Fifth USENIX Symp. Networked Systems Design and Implementation (NSDI '08),* pp. 279-292, Apr. 2008.
[25] L. Xu et al., "Packet Classification Algorithms: From Theory to Practice," *Proc. 28th IEEE INFOCOM '09,* Apr. 2009.

Fong Pong (M'92-SM'10) received the MS and PhD degrees in computer engineering from the University of Southern California, in 1991 and 1995, respectively. He is currently with Broadcom Corporation, where he has developed the award-winning BCM1280/BCM1480 multicore SoCs used in many products, and a GPON device that streams multimedia data at gigabit speed. Before Broadcom, he was with several start-ups, HP Labs, and Sun Microsystems, where he developed blade servers, network and storage protocols offload products, and multiprocessor systems. He has received 36 patents and has served the US National Science Foundation (NSF), the IETF RDMA Consortium, and program committees for several conferences. His research interests include network processor designs and development of algorithmic solutions for packet classification, filtering, and QoS. He is a senior member of the IEEE.

Nian-Feng Tzeng (M'86-SM'92-F'10) has been with the Center for Advanced Computer Studies, the University of Louisiana at Lafayette, since 1987. He was on the editorial boards of the *IEEE Transactions on Computers* and the *IEEE Transactions on Parallel and Distributed Systems*, from 1994 to 1998, and from 1998 to 2001, respectively, and also was the chair of the Technical Committee on Distributed Processing of the IEEE Computer Society, from 1999 till 2002. His current research interests include computer communications and networks, high-performance computer systems, and parallel and distributed processing. He is the recipient of the Outstanding Paper Award of the 10th International Conference on Distributed Computing Systems, May 1990, and also the University Foundation Distinguished Professor Award in 1997. He is a fellow of the IEEE and the IEEE Computer Society.