J. Parallel Distrib. Comput. 73 (2013) 1170–1182

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Hint-based cache design for reducing miss penalty in HBS packet classification algorithm

Yeim-Kuan Chang *, Fang-Chen Kuo

Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan

ARTICLE INFO

Article history:
Received 14 September 2012
Received in revised form
19 January 2013
Accepted 18 March 2013
Available online 26 March 2013

Keywords:
Packet classification
Cache
Network processor

ABSTRACT

In this paper, we implement some notable hierarchical or decision-tree-based packet classification algorithms such as extended grid of tries (EGT), hierarchical intelligent cuttings (HiCuts), HyperCuts, and hierarchical binary search (HBS) on an IXP2400 network processor. By using all six of the available processing microengines (MEs), we find that none of these existing packet classification algorithms achieve the line speed of OC-48 provided by IXP2400. To improve the search speed of these packet classification algorithms, we propose the use of software cache designs to take advantage of the temporal locality of the packets because IXP network processors have no built-in caches for fast path processing in MEs. Furthermore, we propose hint-based cache designs to reduce the search duration of the packet classification data structure when cache misses occur. Both the header and prefix caches are studied. Although the proposed cache schemes are designed for all the dimension-by-dimension packet classification schemes, they are, nonetheless, the most suitable for HBS. Our performance simulations show that the HBS enhanced with the proposed cache schemes performs the best in terms of classification speed and number of memory accesses when the memory requirement is in the same range as those of HiCuts and HyperCuts. Based on the experiments with all the high and low locality packet traces, five MEs are sufficient for the proposed rule cache with hints to achieve the line speed of OC-48 provided by IXP2400.

1. Introduction

The drastic growth of Internet traffic incurred from new network services has demanded more computing power from network devices such as routers and switches. Network services such as network address translation, quality of service, access control, and so forth are required to classify incoming packets into different flows according to their headers. The routers use a classifier consisting of predefined rules or filters to make the packet classification decisions. A filter usually consists of five fields: two prefixes of Internet layer-3 IP addresses (32/128 bits for IPv4/IPv6) representing the subnets of the source and destination networks, two ranges of 16-bit numbers representing the Internet layer-4 source and destination ports used by the network applications, and an 8-bit protocol number. Moreover, each filter is associated with a priority and a corresponding action. This classifier is called a five-dimensional packet classifier. Existing packet classification schemes are implemented and compared on the IXP2400 network processor. New cache schemes for packet classification are proposed to reduce cache miss penalty. Data strcuture for cache schemes and resource usage of network processor are considered.

Performance and time-to-market are two important issues in router design. Traditionally, the routers are implemented by software. By adding new codes, software-based routers can be easily enhanced with new services. Although its development time is short, a software-based router design cannot offer sustained high performance. To solve this problem, hardware-based routers are implemented by using application-specific integrated circuit (ASIC) chips, which enable such routers to outperform the software-based ones. Hardware-based routers are, therefore, a better choice when higher processing speed is strongly demanded. Unfortunately, an ASIC-based router is not suitable for long-term use because adding a new service requires redesigning the chip, which, in turn, increases the total cost of the router.

Generally, network processors adopt multiple processing elements to process incoming packets in parallel. Each processing element is multi-threaded for efficient utilization of computation and communication resources. Similar to software-based routers, network processors are also programmable. Although network processor-based routers do not perform as well as ASIC-based ones, its time-to-market is its key advantage and the reason for network processors' increasing popularity in recent years.

* Corresponding author.
E-mail addresses: ykchang@mail.ncku.edu.tw (Y.-K. Chang), p7895107@mail.ncku.edu.tw (F.-C. Kuo).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.03.005

Therefore, the current industrial standard in router development is to use network processors that can process packets in high speed and can be programmed to add new services.

In this paper, we implement some well-known and efficient packet classification algorithms: extended grid of tries (EGT) [1], hierarchical intelligent cuttings (HiCuts) [12], HyperCuts [27], and hierarchical binary search (HBS) [3] on an Intel IXP2400 network processor. We find that none of the implemented packet classification algorithms is capable of achieving the maximum line speed supported by the IXP2400 network processor (i.e., 3.328 Gbps). The low throughput of these existing algorithms results from a large number of memory accesses for each search operation. As shown in this paper, a cache-like first-level data structure in front of the base data structure of these packet classification algorithms can boost the overall throughput. Traditionally, research on caches tends to focus on increasing the cache hit ratios. However, improving the cache hit ratios is usually difficult. Cache hit ratios in the context of packet classification are also much lower than CPU and IP lookup caches. Therefore, to overcome the performance limitation of these packet classification algorithms, a technique should be developed to reduce the cache overheads and eventually improve the overall search performance. The proposed cache schemes are designed for dimension-by-dimension packet classification schemes such as EGT, HiCuts, HyperCuts, and HBS. Given that HBS outperforms the other algorithms, our focus is mainly on HBS enhanced with the proposed cache schemes. Aside from the header and prefix caches mentioned in previous research, we propose the use of hints stored in each cache entry to reduce the search space of the original HBS data structure when cache misses occur. The contributions of this paper are as follows:

- Both header and rule caches with and without hints are proposed as the first-level data structure in front of the base packet classification data structure to improve the overall search performance. Our experiments show that the proposed rule caches with hints need only five microengines (MEs) to achieve the maximum line speed of IXP2400.
- The proposed hint-based caches can be applied to the packet classification algorithms that use hierarchical data structures.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the baseline packet classifier of HBS. Sections 4 and 5 describe the proposed cache schemes and the related issues of implementation, respectively. Section 6 evaluates the proposed cache schemes. Section 7 discusses the issues for mapping the cache schemes on recent network processors. Section 8 concludes the paper.

2. Related work

The simplest packet classification data structure organizes the rules in a linear array and in a decreasing order of priority, and performs a linear search to find the first or all the matched rules against the header fields of incoming packets. The linear search approach is only suitable for a small classifier and is very efficient in terms of memory requirement and rule updates. However, for a large classifier, the linear search approach has a long query time and, therefore, does not develop into a useful packet classification algorithm for high-speed routers. To speed up the packet classification query time, Srinivasan et al. [30] proposed a two-dimensional algorithm called grid of tries (GoT). They showed that for a classifier with 20,000 two-dimensional rules, GoT takes about 0.9 μs per query in a worst-case simulation on a PC with 300 MHz Pentium II CPU and about 7.5 MB memory space. The authors discovered that the scheme cannot be easily extended to more than two dimensions, and so proposed another generalized scheme called Cross-Producting [30]. This scheme builds a table of all possible field interval combinations (cross-products) and pre-computes the highest priority rule matching of each cross-product. Searches can be done quickly by doing separate lookups in each field, combining the results into a cross-product, and indexing these results into the cross-product table. Unfortunately, the size of this table grows astronomically with the number of rules. Srinivasan et al. also proposed another algorithm called tuple-space search [29]. This scheme partitions the rules of a classifier into different tuple categories based on the number of specified bits in the rules. Afterwards, hashing is used among rules within the same category. The main disadvantage of this scheme is the use of hashing, which results in a long search time and makes non-deterministic updates.

Lakshman and Stiliadis [19] proposed a hardware-optimized scheme known as the Lucent Bit Vector scheme. First, the query process of this scheme searches each dimension separately to yield the set of rules that match the packet for each dimension. The search algorithm can be a binary trie, binary range search (BRS), or any one-dimensional IP lookup scheme. Second, the set of matched rules in all dimensions are intersected by using bitmaps to yield the set of final matched rules in all dimensions. One drawback of this scheme is the need for a hardware-assisted parallel architecture, which is impractical for large classifiers due to its large memory consumption. Gupta and McKeown proposed a scheme called the recursive flow classification (RFC) [11], which is very fast but requires pre-computation to construct two or more levels of indexed cross-product tables. As a result, a large amount of memory and parallel hardware support are needed. Other schemes include FIS-trees [8], segment tree [31], HiCuts [12], and HyperCuts [27], which are all decision-tree-based packet classification algorithms. The list of related works described above is in no way complete. Other schemes can be found in two survey studies [13,32]. In the current study, we are interested in packet classification schemes that are related to network processors and use caches.

Several existing packet classification schemes have been implemented on network processors. In [28], Lucent bit vector [19] was implemented on IXP1200 based on two IXP software programming models: the context pipeline model and the functional pipeline model for the pipeline design and parallel design of bit vector scheme, respectively. In the functional pipeline model, different functions are performed in the same ME of IXP1200, and it is concluded that the parallel design outperforms the pipeline design.

Two methods were proposed in [35,22] to reduce the memory requirement of RFC. The difference between these two studies is that the method in [35] reduces the memory in phase 0 of RFC, whereas the method in [22] reduces memory in phase 1 of RFC. In [22], the bitmap-RFC is the modified RFC scheme implemented on the IXP2800 network processor. The bitmaps are used to compress the cross-product table of RFC. The important operation to count the number of set bits in the bitmap is solved by the built-in bit-manipulation instruction (pop_count) provided by IXP2800. For a table of 5700 rules, bitmap-RFC reduced the required memory from 149.9 MB to 43.4 MB.

In [9], the authors implemented a multidimensional multi-bit trie-based packet classification scheme on IXP2400. To solve the memory explosion problem coming from bit expansion, the level-crossing scheme was adopted. However, using this scheme may require back-tracking operations when searching the trie, which results in a greater need for memory accesses. By using the level-crossing scheme, the proposed solution only needs to consume 2.3 MB of memory for a set of 1000 rules, which is much better than the initial memory requirement of 55 MB.

In [24], the well-known HiCuts scheme was modified and the HyperSplit scheme was proposed. Instead of cutting the space to

equal-sized subspaces using the bits in prefixes, HyperSplit divides the searching space along the endpoints of ranges. Thus, when searching the decision tree, the header field is compared with the endpoint values to obtain the child link. With the rule set of about 5000 rules, the decision tree with a maximum of eight linked-list nodes consumes about 10 MB memory.

Several papers [5,7,36,4] have studied the cache schemes for packet classification. Cache architecture is proposed to support packet classification at memory access speeds [36]. The proposed cache belongs to the category of header caches. The dynamic set-associative scheme used in the proposed cache architecture employs the *n*-universal hash functions to best approximate fully the associative near-LRU cache replacement. A Bloom-filter-based cache proposed in [5] allows a small amount of classification inaccuracy, but decreases the size of packet classification cache by almost a magnitude over exact caches. Bloom filter keeps the information on whether the cache is a hit or a miss, but not the complete header information. A mathematical model was also presented to sustain the Bloom filter for the decrease in probability of a false positive situation, e.g., cache misclassification. Although the Bloom-filter-based cache uses bit vectors to indicate the matched rule, it is not efficient because there are a large number of distinct interfaces. Another proposed cache scheme is the digest cache, which performs better than the Bloom-filter-based caches in terms of extensibility, computational complexity, and memory efficiency [4]. Digest cache stores only a hash of the packet's flow identifier, not the flow identifier itself. Therefore, set-associative caches and different cache replacement policies can be easily applied to the digest cache. When the digest is matched, however, the cache entry cannot ensure an accurate classification result. Thus, the authors also proposed the two-level cache architecture to solve the misclassification problem.

Different from the previous schemes, the smart rule cache scheme stores rules instead of the packets' header field values [7]. Smart rule cache merges different flows that match rules with the same action into a single evolving rule. Smart rule cache is implemented by using additional registers to store these evolving rules and additional logic to match incoming packets to these rules.

Compared with the works described above, the proposal in [5, 4] does not require additional hardware. In other words, adopting the schemes proposed in [7,10] into IXP2400 is not easy. In contrast with [5,4], our proposed cache scheme will not lose the accuracy of classification. Instead of focusing on how to improve the cache hit ratios, we focus on reducing the search overhead caused by cache misses. The idea of hints is introduced to improve the overall throughput of the cache system without additional hardware supports (i.e., software-based cache). When the same hashing keys are used, the hit ratios of the proposed hint-based schemes remain the same as those that do not use hints.

3. Baseline packet classification

The baseline packet classification algorithm enhanced by the proposed cache designs consists of a hierarchical structure of sorted arrays. Each level of these arrays is an expansion list constructed by BRS for range fields [6,20,31] or a binary prefix search (BPS) for prefix fields [2]. A five-level hierarchical structure is usually needed for a typical five-dimensional rule table. First, we briefly describe the BPS and then the two-level hierarchical structure of sorted prefixes.

3.1. Binary prefix search

In general, the binary search works only on sorted lists. Therefore, we need to develop a mechanism to compare and sort prefixes. The prefix comparison rule that compares prefixes of different lengths is based on the inequality of $0 < * < 1$ for ternary numbers [2,3]. For example, the relationship among the three 3-bit prefixes 10^*, 1^*, and 11^* is $0^{**} < 10^* < 111$. We compare the prefixes from the leftmost bit (i.e., the most significant bit) to the rightmost bit (i.e., the least significant bit). As a result, we obtain $0^{**} < 10^*$ and $0^{**} < 111$ because of the leftmost bit and $10^* < 111$ because of the middle bit.

Fig. 1(a) shows six sorted 8-bit prefixes based on the prefix comparison rule. Given that the prefixes may enclose others (i.e., prefix enclosure property), a binary search directly on the list of sorted prefixes may fail. The binary search operation for address $Dst = 01\,011\,000$ is considered in Fig. 1(a). The first prefix for comparison is the middle prefix $F = 01^*$. Given that F matches *Dst*, F is temporarily treated as the longest prefix match (LPM). The search will continue on the sub-list of B, A, and C because *Dst* is smaller than F. By following the same search procedure, prefix A and prefix B will be compared consecutively with *Dst*. Since A and B mismatch *Dst*, the final LPM is F. The correct LPM should be $C = 01\,011^*$. However, C is not compared with *Dst* in the search process.

To solve the above mentioned problem, [2] proposed generating some auxiliary prefixes that will inherit the routing information of the original LPM (e.g., C) and then placing them at locations where the search process can find them. For example, if we insert an auxiliary prefix 01 011 000 that inherits the routing information of C, then the search operation for *Dst* will succeed. In general, if one prefix encloses another, the enclosing prefix (e.g., C) is split into two prefixes that cover both sides of the enclosed prefix (e.g., A). A detailed algorithm which generates auxiliary prefixes for binary search can be found in [2,3]. The expansion list, which is the final sorted prefix list from Fig. 1(a), is shown in Fig. 1(b) with four auxiliary prefixes. The total number of prefixes and auxiliary prefixes needed is at most $2N - 1$ for a routing table of N prefixes in [2].

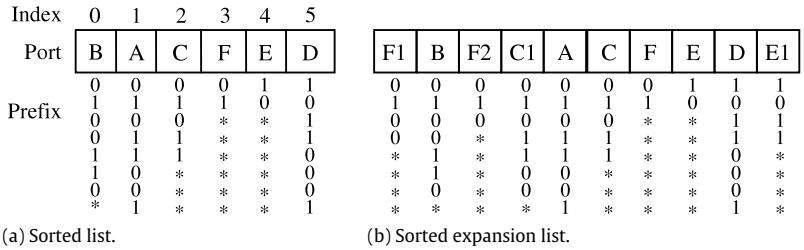
3.2. Binary range search

We use the endpoint definition for ranges proposed in [6] to construct the endpoint array for binary searches. The two integer endpoints of a range $[L, U]$ are defined as $L - 1$ and U when $L \neq 0$, but only one endpoint U is defined when $L = 0$. This definition is different from the traditional definition in which L and U are the endpoints for range $[L, U]$. The advantage of this new endpoint definition is that the number of endpoints will be much less than what is required in the traditional definition. The BRS algorithm (BRS_search) is simple and can be found in [3].

3.3. Hierarchical binary search for packet classification

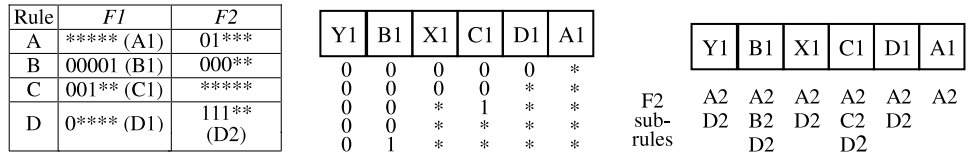
At this point, we illustrate how the BPS and BRS can be applied for multidimensional packet classification. Let a d-dimensional rule set **R** consist of n rules, $r_i = (F1_i, \ldots, Fd_i)$, where $i = 1, \ldots, n$, and the kth field (Fk_i) is a prefix or a range. We use $r = (F1, \ldots, Fd)$ when no confusion has incurred. The basic data structure proposed in this paper is a hierarchical expansion list of BPS and BRS, and is built as follows:

1. The $F1$ expansion list is constructed based on BPS for the prefix field or BRS for the range field by using the $F1$ field values of the rules.
2. For a rule $R = (F1, \ldots, Fd)$, the sub-rule $(F2, \ldots, Fd)$ is duplicated in the element *Elem* of $F1$ expansion list if the $F1$ field value of R covers the range associated with the prefix for BPS or the elementary interval for BRS in element *Elem*. This operation is called rule pushing. Each element of the $F1$ expansion list now contains a number of $(d - 1)$-dimensional sub-rules $(F2, \ldots, Fd)$.

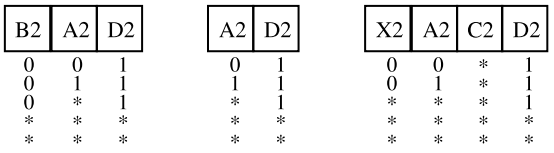


(a) Sorted list. (b) Sorted expansion list.

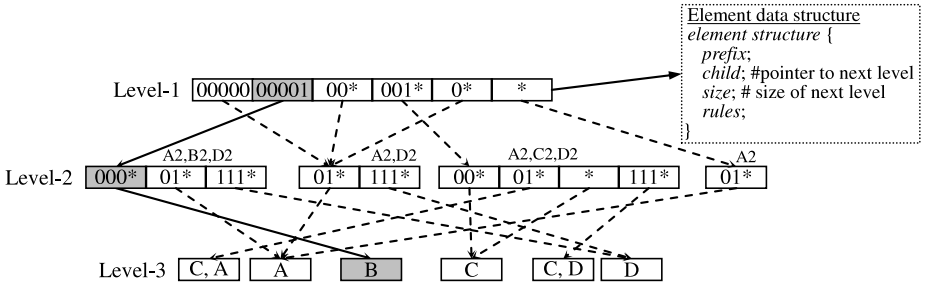
Fig. 1. Binary prefix search example.



(a) 2D rule table example. (b) F1 expansion list. (c) Rule pushing in F1.



(d) F2 expansion lists.



(e) Hierarchical expansion list, where the gray nodes denote the switch nodes for the packet containing headers [00 001, 00 000].

Fig. 2. 2D example.

3. Steps 1 and 2 are repeated for each dimension, but the rule-pushing operation in step 2 is not required for the last dimension.

The two-dimensional rule table example consisting of two prefix fields is considered (Fig. 2(a)). First, we build the F1 expansion list of the rule table (Fig. 2(b)). The F1 expansion list consists of six prefixes (elements), in which X1 and Y1 are auxiliary prefixes. Second, the rule-pushing operations are performed. In other words, the rule of an enclosing F1 prefix in an F1 expansion list is duplicated in the element whose associated prefix is covered by it. For example, F1 prefix A1 covers all the other five prefixes, B1, C1, D1, X1, and Y1. Thus, the sub-rule (A2) is duplicated in the five associated elements of the F1 expansion list (Fig. 2(c)). To this point, each constructed element of the expansion list contains the F2 field values. Finally, we need to construct all the F2 expansion lists (Fig. 2(d)). Given that F2 is the last dimension, no rule pushing is required. Fig. 2(e) shows the final two-level hierarchical expansion list and the detailed element data structure. The "rules" field of the element data structure records the matched rules at the element. The 'rules' fields of the elements in the bottom-level expansion lists are usually non-empty. If the element containing non-empty "rules" field is in the internal expansion list (i.e., not at bottom-level), then some of the common matched rules will be pushed up to save memory space, which is similar with HyperCuts [27]. If there is more than one rule in the "rules" field of an element, the best matched rule is pre-computed to speed up the search time.

The complete multidimensional packet classification HBS_search() algorithm that uses the binary prefix or range searches can be found in [3]. First, the F1 expansion list is searched to find an element (switch node) that contains a pointer for the next-level F2 expansion list built from the subset of rules that match the F1 header of the packet. Second, we search the next-level F2 expansion level to find the switch node in dimension 2. The search continues until it reaches the bottom-level expansion list to obtain the highest priority matched rule. We assume that the first two header fields of the incoming packet are [00 001, 00 000] and the packet classification search is performed on the two-level expansion list (Fig. 2(e)). Third, after searching the first-level expansion list with F1 value of 00 001, we find that the LPM is the second leftmost prefix 00 001 (switch node). Finally, by following the "child pointer" of the switch node, we locate the next-level expansion list and perform the binary search with F2 value of 00 000. The prefix 000* is found to be the match. Therefore, the matched rule is B.

Table 1
The H-cache scheme.

0	F1 tag	F2 tag	F3 tag	F4 tag	F5 tag	$action_0$
...	...					...
$2^m - 1$	F1 tag	F2 tag	F3 tag	F4 tag	F5 tag	$action_{2^m-1}$

4. Proposed cache schemes

4.1. Header cache (H-cache)

In the traditional header cache (H-cache), five-tuple header field values from the incoming packets and the classification result (matched rule ID or action) are cached. If all the header fields of the incoming packet exactly match one of the cached five-tuple header field values (i.e., a cache hit occurs), the cached classification result can be returned directly without searching the packet classification data structure under study. However, if a cache miss occurs, we have to search the rule data structure all over again. Table 1 shows a direct mapped cache table consisting of 2^m entries. To index the header fields of an incoming packet into one of the 2^m cache entries, we need a hashing function $H(F1, F2, F3, F4, F5)$ to perform a many-to-one mapping. Specifically, the five header field values of an incoming packet will be stored in the ith cache entry if $i = H(F1, F2, F3, F4, F5)\%2^m$, where % is the modulus operator. After locating the cache entry for an incoming packet, we have to compare the tags stored in the located cache entry with the packet's header fields one by one to ensure an exact match. A set-associative cache can be built similarly. In this paper, the CRC function supported by the Intel IXP2400 network processor is used to devise the needed hashing function, which will be discussed in a later section.

4.2. Header cache with hints (H-cache-hint)

We describe the basic header cache scheme with hints (H-cache-hint), which is proposed for the packet classification algorithm, by using a dimension-by-dimension hierarchical data structure (e.g., EGT [1], HiCuts [12], and HBS [3]). To ensure a precise description, we only show how the H-cache-hint scheme works for HBS. The idea of the basic H-cache-hint scheme comes from the partial hits, which are defined as cache misses when the first k header field values of the incoming packet match the first k tags and the $(k+1)$th header field value mismatches the $(k+1)$th tag of a cache entry. In H-cache, we have to perform a full search process to find a match after a cache miss. However, if we also cache the index of the switch node at the kth level expansion list, we can use the pointer for the $(k+1)$th level expansion list, which is stored in the kth level switch node, and directly jump to the $(k+1)$th level expansion list to complete the search. As a result, we can avoid searching the first k levels of the expansion lists and, therefore, reduce the cache miss penalty. In the n-dimensional rule sets, we need to cache the $n-1$ indices of switch nodes on the first $n-1$ levels of the expansion lists of HBS. Table 2 shows a direct mapped cache table consisting of 2^m entries that employ the basic H-cache-hint scheme. These cached indices of the switch nodes are the hints that can be used to reduce the search time. However, if the first header of the incoming packet mismatches the first field tag of all cache entries, we still do not gain anything from the hints in the cache. To solve this problem, we propose the modified H-cache-hint (MH-cache-hint) for HBS.

The modified H-cache-hint scheme employs the primary feature of HBS, in which the expansion lists in all the levels are sorted arrays. Consider the case when the first $k-1$ header fields of the incoming packet match the first $k-1$ tags, and the kth header (*Fk_header*) mismatches the kth tag (*Fk_tag*) of a cache entry. The search is performed in the kth level expansion list[1 ... *Fk_index*] if *Fk_header* $\leq$ *Fk_tag* or in the kth level expansion list[*Fk_index+1* ... *Fk_max*] if *Fk_header* $>$ *Fk_tag*, where *Fk_max* is the size of the kth level expansion list obtained from the element F_{k-1} *_index* of the $(k-1)$th level expansion list. Thus, we need to cache n indices of switch nodes for the n-dimensional rule set. Table 3 shows a direct mapped cache table consisting of 2^m entries that employ the MH-cache-hint scheme.

Table 2
The basic H-cache-hint scheme.

0	F1 tag	F2 tag	F3 tag	F4 tag	F5 tag	$action_0$
	F1 index	F2 index	F3 index	F4 index		
...	...					
2^m-1	F1 tag	F2 tag	F3 tag	F4 tag	F5 tag	$action_{2^m-1}$
	F1 index	F2 index	F3 index	F4 index		

4.3. Rule cache

In IP address lookups, previous studies [10,34,14] show that caching prefixes performs better than caching single addresses (i.e., header cache) because of the spatial locality of the IP destination addresses in the packet stream. Based on the same reasoning, we can cache prefixes for the IP address fields and ranges for the port fields in the multidimensional packet classification to obtain a better cache hit ratio. In this section, we propose a cache design called rule Cache (R-cache), in which HBS stores rules in the cache.

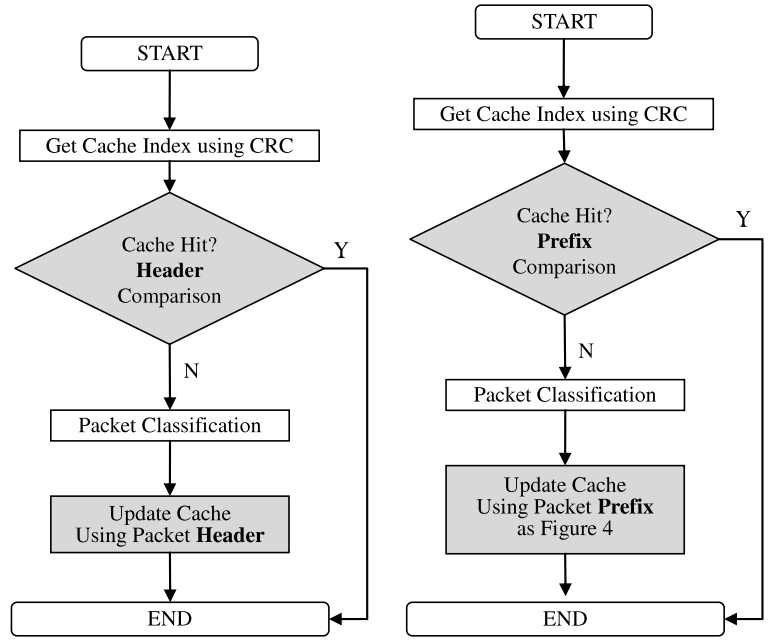
Fig. 3 shows the diagram of H-cache (a) and R-cache (b). The bold texts in the shaded boxes show the differences between H-cache and R-cache. The diagram of H-cache is similar to that of the traditional cache. In R-cache, the scheme needs to keep the source prefix and destination prefix. The address is matched against to the prefix to the matched cache entry. If a cache hit occurs, no additional packet classification operation is necessary. However, if a cache miss occurs, cache entry updating is needed after packet classification. Fig. 4 describes the details for computing the prefix to be written in the cache entry.

In HBS, five expansion lists are searched sequentially for each incoming packet. The results of searching these five expansion lists are two prefixes for prefix fields, two elementary intervals for range fields, and one singleton value for the protocol field. As all elementary intervals are disjoint from one another, caching the two elementary intervals found for range fields in HBS causes no problem, as does caching singleton values for the protocol field. However, caching the prefixes found may lead to incorrect results from cache hits because of the enclosure property of prefixes. Before we describe the details of finding the correct prefixes to be stored in the cache for prefix fields, we summarize the five-tuple rule that can be stored in the R-cache after searching the HBS. Let the five-tuple rule after searching the HBS be (*PFX1*, *PFX2*, *EI1*, *EI2*, *SGLTON*). The computed five-tuple rule that can be stored in the cache is (*C_PFX1*, *C_PFX2*, *EI1*, *EI2*, *SGLTON*), where *C_PFX1* and *C_PFX2* are computed from *PFX1* and *PFX2*, respectively, by the algorithm proposed below.

Let the LPM be X for the destination IP address IP_1 of an incoming packet. If a prefix Y is also contained in X and Y does not match IP_1, caching X for the current address IP_1 produces incorrect results for cache hits from subsequent packets containing the destination IP address IP_2 that matches Y. Consider the four prefixes in the first field of a rule table shown in Fig. 2(a) and the expansion list in Fig. 2(b). Two auxiliary prefixes 00 000 and 00* in the expansion list are generated. Assume that an incoming packet destined at the address 00 011 is followed by another packet destined at 00 001. After binary prefix search operations are performed for IP address 00 011, the LPM is 00*(X1). If we cache

Table 3
The MH-cache-hint scheme.

0	F1 tag	F2 tag	F3 tag	F4 tag	F5 tag	$action_0$
	F1 index	F2 index	F3 index	F4 index	F5 index	
	...					
$2^m - 1$	F1 tag	F2 tag	F3 tag	F4 tag	F5 tag	$action_{2^m-1}$
	F1 index	F2 index	F3 index	F4 index	F5 index	



(a) H-cache. (b) R-cache.

Fig. 3. Flow diagrams for H-cache and R-cache.

```
// List[] is an array storing the expansion list of sorted prefixes.
// D = d31...d0 is the target IP address.
// List[i].prefix is the longest matched prefix of D found by the binary prefix search (BPS)
// ⊇ is the enclosure operator, e.g., X ⊇ Y means prefix X encloses prefix Y
// The Longest Common Ancestor of two prefixes A = aW−1...a0 and B = bW−1...b0 is
// LCA(A, B) = cW−1...ci*, where ck = ak = bk and ck ≠ * for W − 1 ≥ k ≥ i and ai−1 ≠ bi−1.
Algorithm Compute_Cached_Prefix(Element List[], Index i, Address D)
{
01    LCA_left = LCA(D, List[i–1].prefix);
02    LCA_right = LCA(D, List[i+1].prefix);
03    if (List[i].prefix ⊇ List[i+1].prefix and List[i].prefix ⊇ List[i–1].prefix)
04          len = max(LCA_left.length, LCA_right.length);
05    else if (List[i].prefix ⊇ List[i–1].prefix)
06          len = LCA_left.length;
07    else if (List[i].prefix ⊇ List[i+1].prefix)
08          len = LCA_right.length;
09    else len = List[i].prefix.length − 1;
10          return d31...d31−len*;
}
```

Fig. 4. Compute the prefix to be cached after finishing the processing of a cache miss.

$X1$, subsequent packets destined at 00 001 obtain cache hits and incorrectly return $X1$ as the LPM. The correct LPM is 00 001 ($B1$). This problem can be solved easily by using the data structure of the binary trie as follows. In the binary trie, each node corresponds to a prefix. Specifically, a node in level n represents a prefix of length n. When performing the search operation in the binary trie, we traverse the binary trie until all 32 bits are exhausted (i.e., level 32 is reached) or a null child pointer of a node is encountered. Let X be the parent of the non-existing node pointed to by the null pointer encountered last. We can simply cache the prefix associated with X. For example, the cached prefix for address 00 011 should be prefix 0001^* in the five-bit address space although the LPM found for address 00 011 is 00^* (Fig. 2(b)).

Details on how to compute a proper prefix after the LPM is found in HBS are described below. Let $List[i].prefix$ in the expansion list be the LPM for an IP destination address $D = d_{31}.sd_0$. We have to examine the enclosure relationship with the left and right neighboring prefixes of $List[i].prefix$, such as $List[i-1].prefix$ and $List[i+1].prefix$, respectively. If $List[i].prefix$ does not enclose $List[i-1].prefix$ and $List[i+1].prefix$, $List[i].prefix$ can be stored in the cache directly. Otherwise, we have to compute the proper prefix to be stored in the cache. Fig. 4 shows the detailed

algorithm *Compute_Cached_Prefix()*. The first two lines compute LCA_left = LCA(D,*List*[$i-1$].*prefix*) and LCA_right = LCA(D, *List*[$i+1$].*prefix*), which are used later. In lines 3 and 4, if *List*[i].*prefix* encloses both *List*[$i-1$].*prefix* and *List*[$i+1$].*prefix*, we compute the maximum length (*len*) of LCA_left and LCA_right. Then, we use this maximum length to compute the prefix $d_{31}.sd^*_{31-\text{len}}$ of length *len* + 1 to be stored in the cache. If *List*[i].*prefix* only encloses *List*[$i-1$].*prefix*, we compute the length (*len*) of LCA_left, and if *List*[i].*prefix* only encloses *List*[$i+1$].*prefix*, we compute the length (*len*) of LCA_right. If line 9 is satisfied, we compute len = *List*[i].*prefix*.*length* − 1 to cache *List*[i].*prefix*. The computed cached prefix for HBS is the same as that in the binary trie.

4.4. Rule cache with hints (R-cache-hint and MR-cache-hint)

When cache misses occur, schemes similar to H-cache-hint and MH-cache-hint schemes (*R-cache-hint* and *MR-cache-hint*) can be developed to reduce search times by using hints. The cache data structure shown in Tables 2 and 3 can be utilized. When cache misses occur, the same search process used for H-cache-hint and MH-cache-hint can also be employed for the R-cache-hint and MR-cache-hint schemes.

5. Implementation issues in the IXP2400 network processor

5.1. IXP2400 hardware brief

The Intel IXP2400 network processor has an ARM compatible XScale core and eight MEs that can run in parallel or pipeline fashion for processing packets at a high-speed processing rate. Each ME has eight hardware threads that can execute concurrently [15,16,25].

Four kinds of memory units of different sizes and speeds can be accessed by IXP2400 MEs: *local memory*, *scratchpad*, *static random-access memory* (SRAM), and *dynamic random-access memory* (DRAM). Each ME has a local memory size of 640*32 bits, which is private and cannot be shared with other MEs. Scratchpad is the biggest on-chip memory (16 kB in IXP2400). It is usually used as a hardware ring to pass metadata information of packets between MEs. DRAM is used to store the payloads of incoming packets. SRAM is usually used to store the data structures of IP lookups and packet classification. Local memory is the smallest and fastest memory unit. The speed and size of SRAM are between those of the scratchpad and DRAM.

5.2. Resource allocation

Given that IXP2400 has eight MEs, we allocate one ME to receive packets (*receiving ME*) and another to transmit packets (*transmitting ME*). The remaining six MEs (*processing MEs*) can be used to implement the HBS scheme with the proposed cache designs. In general, as the number of processing MEs increases, the forwarding rate of HBS also increases. However, existing packet classification algorithms cannot reach the maximum line speed of IXP2400 because a large number of memory accesses are needed for each search operation.

Among the four memory types in IXP2400, SRAM is selected to store the data structure of the HBS scheme. Our IXP2400 Development Board Radisys ENP-2611 has two channels of SRAMs, each of which is 4 MB. In the later version of ENP-2611, each SRAM is 8 MB, which is sufficient to store the data structure of the rule table. As for the proposed cache design, we abandoned the idea of using local memory to implement caches because local memory has three disadvantages. First, local memory can only store 160 16-byte cache entries because its size is only 640*32 bits. Second, local memory is normally used as an alternative storage called *spilling region* [17] for variables that cannot be allocated to general purpose or transfer registers. Therefore, available local memory that can be used for caches becomes even smaller. Moreover, accessing the part of local memory that implements caches is interfered with by operations on variables stored in spill regions. Third, local memory is private to each ME, and thus, data stored in the local memory of one ME cannot be shared by another ME.

5.3. Data structure design of evaluated packet classifier

As for the implementation issues for packet classification in a real environment such as the Intel IXP2400 network processor, we have to consider the limitation of system resources and the characteristics of real-world rule sets. In practice, a two-level hierarchical expansion list is sufficient because only a small number of three-dimensional sub-rules are pointed to by each element of the second-level expansion list. As a result, a simple linear list sorted by priority can be used for the last three dimensions. In this paper, the two prefix fields (i.e., source and destination IP fields) of the rules are used to construct the two-level hierarchical expansion list. Using the two prefix fields is more efficient than using the two range fields because the numbers of distinct field values in the source and destination IP address fields are much larger than those in the source and destination port fields, respectively. For example, in ACL, Firewall, and IPC tables generated by ClassBench, the number of distinct field values in F3, F4, or F5 is fixed at one value (wildcard) or at a small range of values.

Fig. 5(a) shows the implementation details of the example in Fig. 2. Fig. 5(b) and (c) show the detailed data structures for the level-1 and level-2 expansion lists and the level-3 sub-rule list. All level-2 expansion lists are combined into a single list called *level-2 combined list*, from which some redundant expansion lists are removed. For example, the second leftmost expansion list of 01* and 111* is the sub-list of the leftmost expansion list and can thus be removed. Similarly, level-3 sub-rule lists can be combined into a single list called *level-3 combined list*, from which redundancy is also removed. As the original BPS allows the wildcard field (i.e., *), we need 33 different lengths (i.e., 0–32) that require six bits to represent them. In our implementation, we prohibit wildcards by expanding them to 0*, 1*, or both. Thus, five bits are sufficient. For example, the wildcard field value in the level-1 and level-2 expansion lists in Fig. 2 can be changed to 1* (Fig. 5(a)). Figs. 5(b) and 4(c) illustrate the detailed element data structures of the level-1 list, level-2 combined list, and level-3 combined list. The *BaseIndex* field in the *element12* data structure is set to 22 bits, which can support the level-3 sub-rule array of at most 2^{22} elements for rule tables much larger than the ones used. In element12, only the first three fields are needed for HBS without cache and HBS with H-cache and MH-cache-hint. As the total size of these first three fields exceeds 64 bits and the minimum memory unit of SRAM in IXP2400 is 4 bytes, we use 12 bytes for element 12. The unused space can then be used for the proposed cache schemes, R-cache and MR-cache-hint, which include the fields CoverLeft, CoverRight, LLength, and RLength. When we compute the prefixes to be cached (e.g., List[i].prefix), we have to figure out if List[i].prefix encloses its left (List[$i-1$].prefix) or right neighboring prefix (List[$i+1$].prefix) (Fig. 4). Therefore, the two fields, CoverLeft and CoverRight, are designed for these two conditions. Moreover, LLength and RLength represent the prefix lengths of List[$i-1$].prefix and List[$i+1$].prefix, respectively. With CoverLeft and CoverRight pre-built in the search process, the enclosure operations in Fig. 4 are not needed.

5.4. Data structure design of proposed caches

As the cache is searched by software, searching a set of cache entries will be very slow if the cache is implemented as a set-

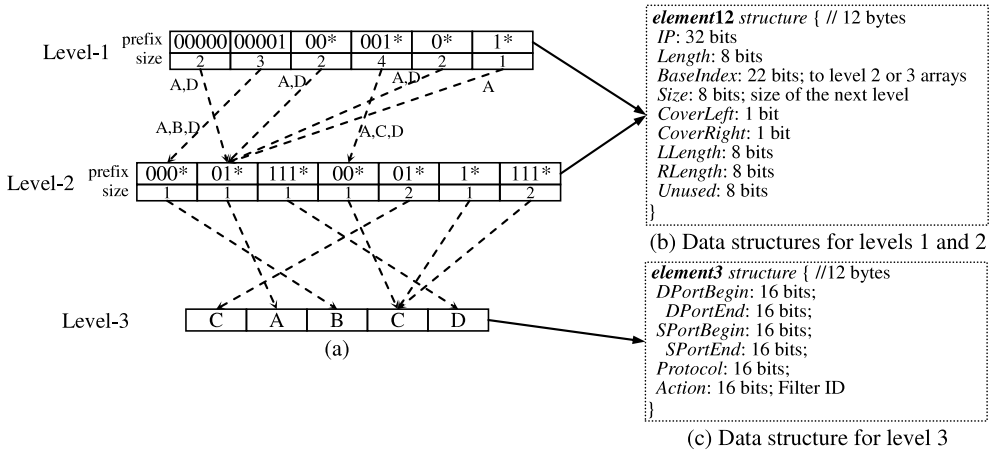


Fig. 5. HBS data structures implemented in SRAM of the IXP2400 network processor.

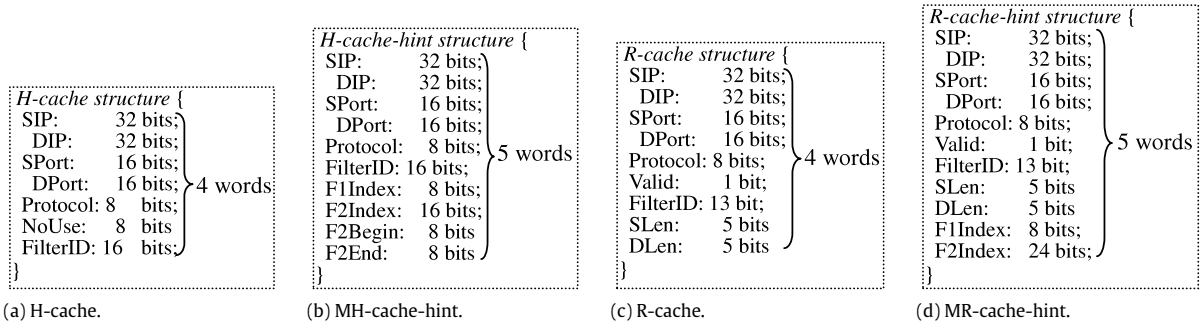


Fig. 6. Data structures for header and rule caches.

associative cache. Thus, we implement the cache as a direct mapped cache in our experiments. As a result, different cache policies [21] are not studied in the paper.

Fig. 6 shows detailed cache data structures for the proposed cache schemes: H-cache, MH-cache-hint, R-cache, and MR-cache-hint. Each H-cache entry consists of 104 bits for the five-dimensional header fields of packets plus 16 bits for rule ID (Fig. 6(a)). MH-cache-hint in Fig. 6(b) contains the fields *F1Index* and *F2Index*, which are the switch node indices of the level-1 and level-2 expansion lists, respectively. The fields *F2Begin* and *F2end* are the indices in the level-2 expansion list. With these two fields, one memory access to the switch node is saved. However, if the two prefix fields stored in cache match the destination and source IP addresses of the packet, but the remaining three fields in cache do not match the destination and source ports and protocol of the packet, we have to search the level-3 sub-rule array by obtaining information from the switch node in the level-2 expansion list.

In this paper, R-cache is designed to contain two prefix fields (SIP/SLen and DIP/DLen) and three exact values in the other three fields (Fig. 6(c)) because we implement the hierarchical expansion list as a two-level structure. Prefixes to be cached are computed based on the algorithm in Fig. 4. Fig. 6(d) shows the similar data structures of MR-cache-hint and MH-cache hint. The number of bits reserved for all fields in the cache structures is sufficient for the rule table of 5000 rules used.

6. Performance evaluation

In our experiments, the code is developed based on the "receive–process–transmit" programming model [18]. *Receiving ME* is dedicated to receiving packets, and *transmitting ME* is dedicated to transmitting packets. The remaining six MEs, called *processing MEs*, are used for the main processing tasks. Processing MEs obtain packets from the receiving ME and send processed packets to the transmitting ME. All codes in processing MEs are written in Micro C, whereas codes in the receiving and transmitting MEs are written in microcode, which is taken from the *Static Forward Project* provided in ENP SDK 3.5 R4 [26], the software development kit for the Radisys ENP-2611 IXP2400 network processor development board [25]. The Workbench development tool of IXA SDK 3.51 [17] is used to conduct simulations, and the frequency of both XScale and ME is set to 600 MHz in all tests. Workbench is a cycle-accurate simulator. Obtained results are not affected much by differences between system architectures, types of CPU, memory sizes, or system loads. Therefore, the confidence interval of our results is close to 100% by repeating each test 10 times.

The data structure of the packet classification scheme in the simulation is first built by using C codes running in a standard personal computer. This pre-built data structure is then loaded into SRAM by Workbench startup scripts and are not modified during the simulation. However, data structures for proposed caches are maintained dynamically by processing MEs. The packet classification data structure is stored in channel 1 SRAM while the cache data structures are stored in the channel 0 SRAM of the IXP2400 network processor.

ClassBench [33] is used to generate rule tables for the experiments. The rule table with n five-dimensional rules is denoted by *Tn*, where $n = 1000$–5000. We only show the results for the tables of 5000 rules (i.e. T5000) with "firewall" setting because the performance results of other settings, ACL and IPC, are similar. Two packet trace files (i.e., T5000H and T5000L) of 50,000 packets are also generated, where suffixes *H* and *L* indicate high and low temporal locality, respectively. The parameters used to generate these traces are a three-tuple of (*Pareto parameter a*, *Pareto parameter b*, *scale*), which is set to (1, 1, 10) and (1, 0.1, 10) for high and low

Table 4
Forwarding rates in MPPS.

Traces	Scheme	1ME	2ME	3ME	4ME	5ME	6ME
T5000H	EGT	0.44	0.72	0.78	0.79	0.79	0.79
	HiCuts	1.00	1.98	2.95	3.85	4.32	4.35
	HyperCuts	1.16	2.32	3.46	4.47	4.67	4.68
	HBS	1.51	2.97	4.14	4.64	4.73	4.74
T5000L	EGT	0.41	0.72	0.81	0.82	0.82	0.82
	HiCuts	0.98	1.95	2.91	3.82	4.26	4.29
	HyperCuts	1.15	2.29	3.42	4.45	4.64	4.64
	HBS	1.48	2.92	4.16	4.80	4.95	4.97
T5000HN	EGT	0.44	0.73	0.78	0.79	0.79	0.79
	HiCuts	0.99	1.98	2.95	3.85	4.32	4.35
	HyperCuts	1.16	2.32	3.46	4.47	4.67	4.68
	HBS	1.50	2.97	4.15	4.63	4.73	4.74
T5000LN	EGT	0.41	0.72	0.81	0.82	0.82	0.82
	HiCuts	0.98	1.95	2.91	3.82	4.27	4.29
	HyperCuts	1.15	2.29	3.42	4.45	4.64	4.64
	HBS	1.48	2.92	4.15	4.80	4.95	4.97

Table 5
Average number of memory accesses and memory requirements for rule table T5000.

	HBS	EGT	HiCuts	HyperCuts
T5000H	21.5	107.3	30.0	19.3
T5000L	21.2	115.7	30.7	19.4
T5000HN	21.5	117.7	30.0	19.3
T5000LN	21.2	115.7	30.7	19.5
Memory(KB)	4986	84	4043	4449

locality, respectively. High locality means a high probability of repeated packets containing the same five header fields for packet classification. Trace files also associate each packet with the original rule generating the packet. To test the rule cache, two additional packet traces, *T5000HN* and *T5000LN*, are generated from T5000H and T5000L, respectively, with the following constraint: the least significant (32-*len*) bits of the destination and source IP address field values of each packet are randomly complemented if the associated prefix length is *len*. The suffix *N* implies LAN-based traffic where repeated packets come from the same source subnet or go to the same destination subnet. LAN-based traffic also implies the spatial locality of packets. All packets in the experiments are assumed to be the smallest Ethernet packets of 64 bytes.

Table 4 shows the forwarding rates in million packets per second (MPPS) for HBS, EGT, HiCuts, and HyperCuts. The variations of different trace files have no impact on the forwarding rates of all tested schemes because no cache takes advantage of packet temporal locality. The forwarding rate achieved by HBS increases faster than that of other schemes when more processing MEs are used. In general, HBS performs the best and EGT performs the worst. To further find the reasons for the forwarding rate difference among these schemes, we analyze the average numbers of SRAM memory accesses in Table 5, which are computed from one of the processing MEs. None of these four schemes can achieve the maximum forwarding rate (i.e., line speed) of 6.46 MPPS for IXP2400. Therefore, the proposed cache schemes are used to achieve the line speed of 6.46 MPPS for IXP2400.

Before we demonstrate the performance results of the proposed cache schemes, we also give the memory requirements of the tested packet classification schemes in Table 5. The memory requirements of HBS, EGT, HiCuts, and HyperCuts for rule table T5000 are 4986, 84, 4043, and 4449 kB, respectively. EGT requires the least amount of memory, but its forwarding rate is the worst. HBS, HiCuts, and HyperCuts require around 4–5 MB of memory; HBS needs slightly more memory because of the rule pushing and auxiliary nodes. Compared with the bitmap-RFC (a compressed RFC-like scheme) proposed in [22], which needs 43.4 MB for 5700 rules, the memory requirements for HBS, HiCuts, and HyperCuts are much better.

The CRC units supported in the IXP2400 network processor are used to implement the hash function for mapping the five header field values of an incoming packet to the correct cache entry in H-cache and R-cache. In IXP2400, two hashing function units are available: SHaC and CRC units. Only one SHaC function unit is shared by all MEs. Therefore, calling SHaC unit by all processing MEs for all packets results in heavy workload on the SHaC unit and, thus, longer processing time. On the contrary, a CRC unit exists in each ME. The CRC unit should be used to reduce the overhead of hashing operations in cache.

Two classes of hashing keys, $5D - k$ and *SA-k*, are used in our experiments. The hashing key in $5D - k$ is $k + k + 16 + 16 + 8$ bits extracted from the most significant k bits from source and destination IP field values, 16 bits from source and destination port field values, and 8 bits from the protocol field value of the header of the incoming packet. In *SA-k*, only the most significant k bits from the source IP field value is used as the hashing key. By testing numerous settings of various k values, the settings with the best search performance for cache schemes are 5D-32, SA-32, 5D-24, and SA-32 for the H-cache, MH-cache-hint, R-cache, and MR-cache-hint schemes, respectively.

Hereafter, we focus only on the performance evaluation of HBS integrated with the proposed caches because HBS performs better than EGT, HiCuts, and HyperCuts. The hit ratios of all proposed cache schemes are shown in Table 6. High temporal locality traces (T5000H and T5000HN) have higher hit ratios than low temporal locality traces for any cache scheme. For traces with network-based traffic, such as T5000HN and T5000LN, the hit ratios of rule caches are much larger than those of header caches. Even for trace T5000L, R-cache is a little better than H-cache. Consider the impact of hint-based caches on hit ratios. For traces T5000H and T5000L, the hit ratios of hint-based caches are worse than those of non-hint-based caches by around 3%–4%. For traces T5000HN and T5000LN, the hit ratios of MR-cache-hint are also worse than those of R-cache by 3%–6%. However, the hit ratios of MH-cache-hint are much worse than those of H-cache. These hit ratio differences result from the different hash keys used by the different cache schemes. If the same hash keys are used, the hit ratios of hint-based caches are not different from those of non-hint-based caches. Although SA-32 uses a different hashing key, for the proposed hint-based caches, it incurs various amounts of loss in term of hit ratios. Therefore, the overall search throughput of hint-based caches is improved because a significant amount of miss ratios can be utilized to reduce search times.

Table 7 summarizes the origins of cache misses. Column F1Miss indicates the cache miss ratios that occur when the cached F1 tag does not match header field F1 of the incoming packet. Similarly, column F1Hit–F2Miss indicates the cache miss ratios that occur when the cached F1 tag matches header field F1 and the cached F2 tag does not match header field F2 of the incoming packet. Column F1F2Hit–F3F4F5Miss indicates the cache miss ratios that occur when the cached F1 and F2 tags match header fields F1 and F2 and the cached F3, F4, or F5 tag does not match header field F3, F4, or F5 of the incoming packet. As the hints stored along with cache entries are the indices of cached F1 and F2 tags on the level-1 and level-2 expansion lists for previously searched packet header fields, search spaces on the level-1 and level-2 expansion lists can be reduced. Search space reduction for F1F2Hit–F3F4F5Miss cases are the most beneficial because only a small sub-range of the level-3 expansion list needs to be searched. F1Hit–F2Miss is the most frequent case for both MH-cache-hint and MR-cache-hint (Table 7).

Table 8 shows the forwarding rates of HBS enhanced with the proposed caches of 128 entries. We highlight the cases when the

Table 6
Hit ratios of HBS with the proposed cache schemes.

Traces	Cache size	H-cache (%)	MH-cache-hint (%)	R-cache (%)	MR-cache-hint (%)
T5000H (5D-32)	128	75.61	72.82	75.06	72.57
	1024	76.50	73.33	76.75	73.29
T5000L (SA-32)	128	44.30	43.57	44.57	43.57
	1024	47.18	44.07	48.60	44.19
T5000HN (5D-24)	128	66.76	39.36	75.26	69.62
	1024	69.03	40.22	76.80	70.64
T5000LN (SA-32)	128	39.59	22.53	44.39	41.69
	1024	42.30	22.92	48.65	42.53

Table 7
Cache miss ratios for HBS with MH-cache-hint and MH-cache-hint.

	Traces	Cache size	F1Miss (%)	F1Hit–F2Miss (%)	F1F2Hit–F3F4F5Miss (%)	5DHit (%)
MH-cache-hint	T5000H	128	7.64	17.59	1.48	72.82
		1024	0.36	23.91	1.73	73.33
	T5000L	128	14.89	38.35	2.97	43.57
		1024	0.84	51.29	3.51	44.07
	T5000HN	128	11.79	47.12	1.34	39.36
		1024	1.11	56.19	1.69	40.22
	T5000LN	128	22.65	52.06	2.48	22.53
		1024	2.49	70.95	3.10	22.92
MR-cache-hint	T5000H	128	7.36	19.61	1.78	72.57
		1024	0.38	25.68	2.16	73.29
	T5000L	128	14.31	41.91	3.80	43.57
		1024	0.83	54.69	4.72	44.19
	T5000HN	128	11.31	18.67	1.79	69.62
		1024	1.12	27.47	2.30	70.64
	T5000LN	128	21.79	36.24	3.45	41.69
		1024	2.45	54.50	4.65	42.53

Table 8
Forwarding rates of proposed cache schemes in MPPS (percentage of max speed).

Cache size = 128 entries		1ME	2ME	3ME	4ME	5ME	6ME
T5000H	H-cache	3.19(49%)	5.18(80%)	6.46 (100%)			
	MH-cache-hint	3.71(57%)	6.15(95%)				
	R-cache	3.05(47%)	5.06(78%)				
	MR-cache-hint	3.48(54%)	5.80(90%)				
T5000L	H-cache	1.83(28%)	3.46(54%)	4.87(75%)	5.86(91%)	6.18(96%)	6.24(97%)
	MH-cache-hint	2.32(36%)	4.36(67%)	5.95(92%)	6.46(100%)		
	R-cache	1.77(27%)	3.34(52%)	4.77(74%)	5.81(90%)	6.17(96%)	6.26(97%)
	MR-cache-hint	2.17(34%)	4.09(63%)	5.74(89%)	6.46(100%)		
T5000HN	H-cache	2.67(41%)	4.71(73%)	6.30(98%)	6.46(100%)		
	MH-cache-hint	2.49(39%)	4.60(71%)	6.31(98%)			
	R-cache	3.05(47%)	5.07(78%)				
	MR-cache-hint	3.20(50%)	5.50(85%)				
T5000LN	H-cache	1.74(27%)	3.36(52%)	4.80(74%)	5.83(90%)	6.11(95%)	6.18(96%)
	MH-cache-hint	1.97(34%)	3.80(59%)	5.39(83%)	6.24(96%)	6.40(99%)	6.42(99%)
	R-cache	1.77(27%)	3.35(52%)	4.77(74%)	5.83(90%)	6.17(96%)	6.26(97%)
	MR-cache-hint	2.05(32%)	3.91(61%)	5.52(85%)	6.33(98%)	6.46(100%)	

maximum forwarding rates of 6.46 MPPS supported by IXP2400 reach 6.46 (100%), and use gray cells to indicate when 90%–99% of the maximum forwarding rates can be achieved. Hint-based caches perform better than those without hints for all traces, except that H-cache is better than MH-cache-hint for trace T5000HN with one or two processing MEs. For high-locality packet traces T5000H and T5000HN, three MEs are enough for all caches to achieve maximum forwarding rate, except for the H-cache and MH-cache-hint running for trace T5000HN. Furthermore, with the low-locality packet trace T5000L, both MH-cache-hint and MR-cache-hint can achieve the maximum forwarding rate with four processing MEs. For traces T5000H and T5000L, MH-cache-hint performs better than MR-cache-hint when no more than three MEs are used. When five MEs are utilized, only MR-cache-hint can achieve the maximum forwarding rate for all traces. Table 9 shows the average number of memory accesses, which basically follow trends similar to those of the forwarding rates (Table 8).

To demonstrate the performance advantages of the proposed cache designs over the original HBS, we use bar charts to demonstrate the speedups of HBS enhanced with the proposed cache designs relative to the original HBS running on one ME (Fig. 7). The performance improvement of the original HBS from using three to five MEs is insignificant, which implies that the performance of the original HBS is limited by the data structure itself, not by the number of MEs used. On the contrary, the performance improvements of the HBS enhanced by the MH-cache-hint and MR-cache-hint schemes are limited by the line speed of the IXP2400. If the line speed of IXP2400 increases, the

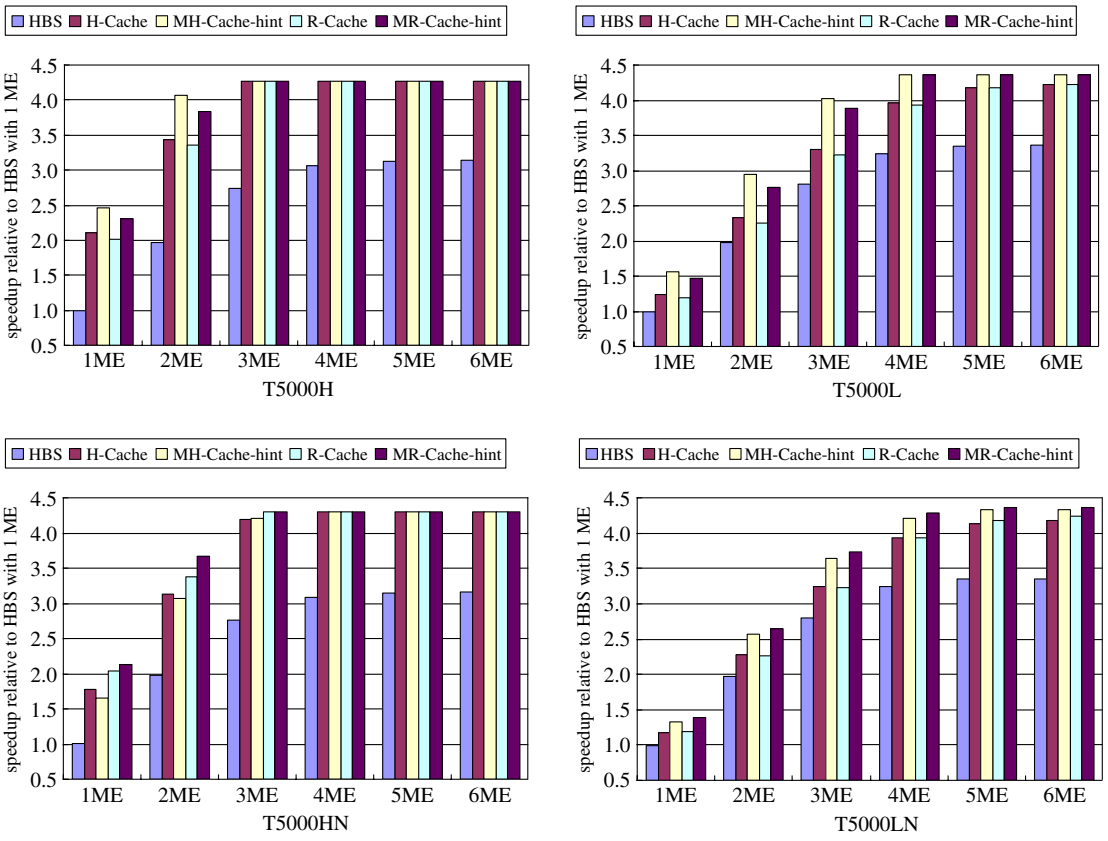


Fig. 7. Speedups of original HBS and HBS enhanced with the proposed cache schemes relative to HBS with 1 ME. (Cache size = 128 entries.)

Table 9
Average number of memory accesses per search.

Traces	# of entries	HBS	H-cache	MH-cache-hint	R-cache	MR-cache-hint
T5000H	128	21.46	6.56	5.35	6.68	5.49
	1024	21.46	6.35	4.66	6.28	4.86
T5000L	128	21.22	13.65	9.70	13.59	9.95
	1024	21.22	12.93	8.39	12.64	8.76
T5000HN	128	21.46	8.46	9.68	6.63	6.15
	1024	21.46	7.93	8.66	6.27	5.21
T5000LN	128	21.24	14.64	12.76	13.64	10.81
	1024	21.24	13.97	10.93	12.63	9.13

speedups of the MH-cache-hint and MR-cache-hint schemes are also expected to increase.

As rule tables may also be updated from time to time, subsequent experiments are designed to evaluate the performance of the proposed hint-based caches when updates are considered. Update frequency is assumed to be 100 or 1000 updates per second.

We use the *Foreign Model simulation extensions* of Workbench [17] to implement cache invalidation, which invalidates all cache entries every centisecond or millisecond. We evaluate only the proposed hint-based caches and show the results in Table 10. Forwarding rate reductions in percentage caused by update operations are shown in parentheses. As a result, forwarding rates are reduced by at most 5.5% when the impact of updates is considered. Similar to the case of no update, MR-cache-hint needs only five MEs to achieve the maximum forwarding rate provided by IXP2400 for all traces.

In pages 6 and 7, we survey some previous work [5,7,10,4], of which only [5,4] investigate software-based caches because they do not need additional hardware logic. In [5,4], the authors implemented the proposed schemes on Intel IXP1200. Table 11 shows the throughput obtained from [5]. The average throughput of [4] is 803 Mb/s. However, because [5,4] lack cache size information, a fair comparison with them is difficult to make.

7. Future work

We use IXP2400 in the experiments because we have an IXP2400-based development board ENP-2611 made by Radisys in our laboratory. Thus, we shall use IXP2400 for real implementation. Aside from IXP2400, the second generation of the Intel IXP network processor family also includes IXP2800, IXP2805, IXP2850, and IXP2855 [18]. Only IXP2850 and IXP2855 have enabled cryptography functions, which are not used for packet classification. All IXP28XX series network processors contain more MEs (i.e., 16) than IXP2400 (i.e., 8). More MEs can bring higher throughput, but also lead to higher system costs. On the other hand, Netronome produces the upgraded version of IXP-2XXX compatible network processors, namely, NFP-3216, NFP-3240, and NFP-6XXX [23], which have more MEs (NFP-3240 has 40 MEs). The syntax used for NFP network processors are not 100% compatible with IXP2XXX processors. Function units are not exactly the same

Table 10
Forwarding rates when updates are considered in caches of 128 entries.

		Update (s)	1ME	2ME	3ME	4ME	5ME	6ME
T5000H	MH	0	3.71	6.15	6.46			
	Cache	100	3.56(−4.0%)	5.96(−3.1%)				
	Hint	1000	3.52(−5.1%)	5.95(−3.3%)				
	MR	0	3.48	5.80				
	Cache	100	3.33(−4.3%)	5.59(−3.6%)				
	Hint	1000	3.29(−5.5%)	5.57(−4.0%)				
T5000L	MH	0	2.32	4.36	5.95	6.46		
	Cache	100	2.26(−2.6%)	4.28(−1.8%)	5.93(−0.3%)			
	Hint	1000	2.25(−3.0%)	4.27(−2.1%)	5.92(−0.5%)			
	MR	0	2.17	4.09	5.74			
	Cache	100	2.12(−2.3%)	4.05(−1.0%)	5.68(−1.0%)			
	Hint	1000	2.09(−3.7%)	4.02(−1.7%)	5.67(−1.2%)	6.41(−0.8%)		
T5000HN	MH	0	2.49	4.60	6.31	6.46		
	Cache	100	2.42(−2.8%)	4.51(−2.0%)	6.19(−1.9%)			
	Hint	1000	2.41(−3.2%)	4.49(−2.4%)	6.19(−1.9%)			
	MR	0	3.20	5.50				
	Cache	100	3.09(−3.4%)	5.34(−2.9%)				
	Hint	1000	3.06(−4.4%)	5.32(−3.3%)				
T5000LN	MH	0	1.97	3.80	5.39	6.24	6.40	6.42
	Cache	100	1.93(−2.0%)	3.76(−1.1%)	5.36(−0.6%)	6.20(−0.6%)	6.37(−0.5%)	6.42(−0.0%)
	Hint	1000	1.92(−2.5%)	3.75(−1.3%)	5.35(−0.7%)	6.19(−0.8%)	6.37(−0.5%)	6.40(−0.1%)
	MR	0	2.05	3.91	5.52	6.33	6.46	
	Cache	100	2.00(−2.4%)	3.84(−1.8%)	5.47(−0.9%)	6.31(−0.3%)		
	Hint	1000	1.99(−2.9%)	3.83(−2.0%)	5.46(−1.1%)	6.30(−0.5%)		

Table 11
Throughput of scheme proposed in [10] (based on IXP1200).

Number of hash levels	All-miss cache throughput (Mb/s)
0	990
1	868
2	729
3	679
4	652
5	498

as those of IXP2400 (e.g., MEs control store, memory controller, and media switch fabric). The performance results conducted on these advanced network processors are more attractive. Therefore, we intend to obtain boards implemented with these advanced network processors in the near future and to conduct further study on them. The ultimate goal is to implement a real high-performance router using such boards.

Nevertheless, instructions for mapping our schemes onto these processors are still provided. For IXP28XX-based processors:

1. As IXP28XX processors have more MEs (i.e., 16) than IXP2400 (i.e., 8), we can allocate more MEs for packet processing, which somehow lead to higher throughput.
2. The operation frequency of IXP2800 (1.4 GHz) is faster than that of IXP2400 (600 MHz). Thus, the time needed for non-memory-access operations (i.e., register operations) becomes shorter.
3. IXP28XX processors have four SRAM channels instead of the two in IXP2400, which indicates more SRAM space available to programmers. In fact, we can distribute the needed data structure to all four SRAM channels. For example, the data structure of the cache can be stored in SRAM channel 0, that of the first dimension of HBS in SRAM channel 1, that of the second dimension of HBS in SRAM channel 2, and the remaining data structure in channel 3. This distribution speeds up packet processing because it reduces the time needed to obtain data from SRAM.

For NFP-3XXX- and NFP-6XXX-based processors:

1. As NFP series processors have even more MEs, we can allocate more MEs for packet processing as well.
2. NFP series processors have larger control stores (16 K) than IXP2400 (4 K); thus, we can embed some shared data structure into codes to utilize unused space (i.e., by hardcoding). For example, we can embed the first-dimension data structure of HBS into the control store, thereby reducing the time for packet processing. Thus, memory access to SRAM for the first-dimension data structure is no longer necessary.
3. IXP2400 has per-ME content-addressable memory, but its size is just 16 entries and it is not shared by all MEs. Therefore, IXP2400 is not suitable for our proposed cache schemes. However, NFP processors support additional ternary content-addressable memory (TCAM; shareable). We can store the proposed cache in the TCAM to replace the direct mapping scheme we use for better cache hit ratio.

8. Conclusion

In this paper, we evaluate some well-known packet classification schemes such as EGT [1], HiCuts [12], HyperCuts [27], and HBS [3] by implementing them on an Intel IXP2400 network processor. HBS outperforms EGT, HiCuts, and HyperCuts. However, none of these schemes can achieve the maximum line speed supported by IXP2400. Therefore, we propose the use of header and prefix caches to improve the performance of HBS. Furthermore, these two cache schemes are enhanced with hints stored in cache entries to reduce the cache miss penalty. The proposed cache schemes are suitable for any dimension-by-dimension hierarchical packet classification scheme. HBS enhanced with any of the four proposed cache schemes can achieve the line speed of IXP2400. In particular, HBS enhanced with hint-based caches outperforms HBS enhanced with cache schemes without hints.

References

[1] F. Baboescu, S. Singh, G. Varghese, Packet classification for core routers: is there an alternative to CAMs? in: Proc. IEEE INFOCOM 2003, vol. 1, 2003, pp. 53–63.
[2] Y.-K. Chang, Fast binary and multiway prefix searches for packet forwarding, Computer Networks 51 (3) (2007) 588–605.
[3] Y.-K. Chang, Efficient multidimensional packet classification with fast updates, IEEE Transactions on Computers 58 (4) (2009) 463–479.

[4] F. Chang, W.-C. Feng, W.-Chi Feng, K. Li, Efficient packet classification of digest caches, in: Proc. the Third Workshop on Network Processors & Applications, NP3, 2004.
[5] F. Chang, K. Li, W.-C. Feng, Approximate caches for packet classification, in: Proc. IEEE INFOCOM 2004, vol. 4, 7–11 March 2004, pp. 2196–2207.
[6] Y.-K. Chang, Y.-C. Lin, Dynamic segment trees for ranges and prefixes, IEEE Transactions on Computers 56 (6) (2007) 769–784.
[7] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, Wire speed packet classification without TCAMs: a few more registers (and a bit of logic) are enough, ACM SIGMETRICS Performance Evaluation Review 35 (1) (2007) 253–264.
[8] A. Feldmann, S. Muthukrishnan, Tradeoffs for packet classification, in: Proc. IEEE INFOCOM 2000, vol. 3, 26–30 March 2000, pp. 1193–1202.
[9] S. Giordano, G. Procissi, F. Rossi, F. Vitucci, Design of a multi-dimensional packet classifier for network processors, in: Proc. IEEE ICC 2006, vol. 2, 2006, pp. 503–508.
[10] K. Gopalan, T. Chiueh, Improving route lookup performance using network processor cache, in: Proc. the 2002 ACM/IEEE Conference on Supercomputing, 2002, pp. 1–10.
[11] P. Gupta, N. McKeown, Packet classification on multiple fields, ACM SIGCOMM Computer Communication Review 29 (4) (1999) 147–160.
[12] P. Gupta, N. McKeown, Classifying packets with hierarchical intelligent cuttings, IEEE Micro 20 (1) (2000) 34–41.
[13] P. Gupta, N. Mckeown, Algorithms for packet classification, IEEE Network 15 (2) (2001) 24–32.
[14] Zhuo Huang, Gang Liu, Jih-Kwon Peir, Greedy prefix cache for IP routing lookups, in: Proc. the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks, 2009, pp. 92–97.
[15] Intel Corporation, Intel® IXP2400 network processor hardware reference manual, November 2003.
[16] Intel Corporation, Intel® IXP2400/IXP2800 network processors microengine C language support reference manual, November 2003.
[17] Intel Corporation, Intel® IXP2400/IXP2800 network processors development tools user's guide, March 2004.
[18] E.J. Johnson, A.R. Kunze, IXP2400/2800 Programming: The Complete Microengine Coding Guide, Intel Press, 2003.
[19] T.V. Lakshman, D. Stiliadis, High-speed policy-based packet forwarding using efficient multi-dimensional range matching, ACM SIGCOMM Computer Communication Review 28 (4) (1998) 203–214.
[20] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Transactions on Networking 7 (3) (1999) 324–334.
[21] G. Liao, H. Yu, L. Bhuyan, A new IP lookup cache for high performance IP routers, in: Proc. the 47th Design Automation Conference, DAC'10, 2010, pp. 338–343.
[22] D. Liu, Z. Chen, B. Hua, N. Yu, X. Tang, High-performance packet classification algorithm for multithreaded IXP network processor, ACM Transactions on Embedded Computing Systems 7 (2) (2008).
[23] Network flow processor, NFP-3240 and NFP-6XXX. http://www.netronome.com.
[24] Y. Qi, L. Xu, B. Yang, Y. Xue, J. Li, Packet classification algorithms: from theory to practice, in: Proc. IEEE INFOCOM, 2009, pp. 648–656.
[25] RadiSys Corporation, ENP-2611 hardware reference, August 2003.
[26] RadiSys Corporation, ENP software development kit programmer's guide, April 2004.
[27] S. Singh, F. Baboescu, G. Varghese, J. Wang, Packet classification using multidimensional cutting, in: Proc. ACM SIGCOMM, August 2003, pp. 25–29.
[28] D. Srinivasan, W.-C. Feng, Performance analysis of multi-dimensional packet classification on programmable network processors, Computer Communications 28 (15) (2005) 1752–1760.
[29] V. Srinivasan, S. Suri, G. Varghese, Packet classification using tuple space search, in: Proc. ACM SIGCOMM, 1999, pp. 135–146.
[30] V. Srinivasan, G. Varghese, S. Suri, M. Waldvogel, Fast and scalable layer four switching, ACM SIGCOMM Computer Communication Review 28 (4) (1998) 191–202.
[31] C.-F. Su, High-speed packet classification using segment tree, in: Proc. IEEE GLOBECOM, vol. 1, 2000, pp. 582–586.
[32] D.E. Taylor, Survey and taxonomy of packet classification techniques, ACM Computing Surveys 37 (3) (2005) 238–275.
[33] D.E. Taylor, J.S. Turner, ClassBench: a packet classification benchmark, IEEE/ACM Transactions on Networking 15 (3) (2007) 499–511.
[34] Nian-Feng Tzeng, Routing table partitioning for speedy packet lookups in scalable routers, IEEE Transactions on Parallel and Distributed Systems 17 (5) (2006) 481–494.
[35] B. Xu, D. Jiang, J. Li, HSM: a fast packet classification algorithm, in: Proc. IEEE AINA 2005, vol. 1, 2005, pp. 987–992.
[36] J. Xu, M. Singhal, J. Degroat, A novel cache architecture to support layer-four packet classification at memory access speeds, in: Proc. IEEE INFOCOM, 2000, pp. 1445–1454.

Yeim-Kuan Chang received Ph.D. degree in Computer Science from Texas A&M University, College Station, in 1995. He is currently a Professor in the Department of Computer Science and Information Engineering, National Cheng Kung University, Taiwan. His research interests include Internet router design, computer architecture, and multiprocessor systems.

Fang-Chen Kuo received the M.S. degree in Computer Science and Information Engineering from National Cheng Kung University, Taiwan, Republic of China, in 2006. He is currently working toward the Ph.D. degree in Computer Science and Information Engineering at National Cheng Kung University, Taiwan, Republic of China. His current research interests include high-speed networks and high-performance Internet router design.