Packet Classification Using Coarse-grained Tuple Spaces

Haoyu Song
Washington University
St. Louis, MO
hs1@arl.wustl.edu

Jonathan Turner
Washington University
St. Louis, MO
jon.turner@wustl.edu

Sarang Dharmapurikar
Washington University
St. Louis, MO
sarang@arl.wustl.edu

ABSTRACT

While the problem of high performance packet classification has received a great deal of attention in recent years, the research community has yet to develop algorithmic methods that can overcome the drawbacks of TCAM-based solutions. This paper introduces a hybrid approach, which partitions the filter set into subsets that are easy to search efficiently. The partitioning strategy groups filters that are close to one another in *tuple space* [10], which makes it possible to use information from single field lookups to limit the number of subsets that must be searched. We can trade-off running time against space consumption by adjusting the coarseness of the tuple space partition. We find that for two-dimensional filter sets, the method finds the best-matching filter with just four hash probes while limiting the memory space expansion factor to about 2. We also introduce a novel method for Longest Prefix Matching (LPM), which we use as a component of the overall packet classification algorithm. Our LPM method uses a small amount of on-chip memory to speedup the search of an off-chip data structure, but uses significantly less on-chip memory than earlier methods based on Bloom filters.

Categories and Subject Descriptors

C.2.6 [**Internetworking**]: Routers

General Terms

Algorithms, Design, Performance

Keywords

Packet Classification, Longest Prefix Matching

1. INTRODUCTION

Network routers, firewalls and intrusion prevention systems use packet classification as an enabling mechanism for several higher level functions, including access control, traffic engineering and quality-of-service mapping. Effective algorithmic solutions for packet classification remain elusive. Although TCAMs provide an effective, general solution for high performance systems, TCAMs are relatively expensive, have high power consumption and may not even be available in some system contexts. Hence, there is continuing interest in identifying algorithmic approaches that are more broadly applicable and can provide high throughput without excessive memory consumption.

Algorithmic methods for packet classification use one of more on-chip lookup engines (implemented either in hardware, or as software threads) accessing a data structure in off-chip memory. The key objective in this approach is to minimize the number of off-chip memory accesses needed to complete a lookup. While some classification methods are able to achieve fast lookup rates, this often comes at the expense of a large expansion in the memory required to represent the filter set (some methods consume several thousand bytes per filter). In this paper, we study an approach to packet classification that partitions the filter set into subsets that are easy to search efficiently. The partitioning strategy groups filters that are close to one another in *tuple space* [10], which makes it possible to use information from single field lookups to limit the number of subsets that must be searched. The method provides a straightforward way to trade-off running time against space consumption by adjusting the coarseness of the tuple space partition. To implement the one-dimensional lookups, we use a novel method for Longest Prefix Matching (LPM), which uses a small amount of on-chip memory to reduce the number of off-chip memory accesses needed. This approach improves on the technique first developed in [4], significantly reducing the amount of on-chip memory needed to obtain bounded worst-case performance.

Although our methods can be extended to handle general packet classification, we focus here on the special case of two-dimensional classification, in which each filter specifies a pair of prefixes. 2D packet classification is widely used in basic Access Control Lists (ACL) [3]. Moreover, EGT-PC [1] shows that it can also be used in general packet classification with some extensions. We show that our method is capable of high lookup rates without significantly expanding the memory required to represent the filters.

The remainder of this paper is organized as follows. We discuss the related work in Section 2. We describe the packet classification algorithm in Section 3. Our Longest Prefix Matching algorithm is described in Section 4. We discuss the tuple partition issues in Section 5 and evaluate the algorithm performance using different filter sets in Section 6. We conclude the paper in Section 7

2. RELATED WORK

Some previous work addresses the 2D packet classification problem. The AQT algorithm [2] applies the 2D cutting technique. Its performance highly depends on the filter set structure. The GOT [12] and EGT [1] algorithms are both trie-based, and have a worst-case lookup performance of $\alpha \times w$, where w is the longest prefix length and α is some constant factor. They traverse the prefix trees and jump between them to examine all possible matches. Another type of algorithm uses the tuple space search technique. A tuple is defined as a pair of unique prefix lengths (u, v). Filters belonging to the same tuple are stored in one hash table. The lookups are conducted by querying the hash tables. The storage and the lookup time depend on the number of tuples. The 2D tuple space search requires w^2 hash queries per lookup in the worst case. An enhanced version, called rectangle search, reduces the number to $2w-1$, at the cost of preprocessing and more storage [10]. However, this still requires up to 63 hash queries per lookup for IPv4 in the worst case. Another algorithm [14] that operates on conflict-free filter sets uses binary search to reduce the number of hash queries to $\log^2 w$, which is 25 for IPv4. Unfortunately, most 2D filter sets are not conflict-free. Despite the differences, the performance of the above algorithms depends on the prefix length w, which makes these algorithms less attractive for the IPv6 case.

Since our algorithm is directly derived from the tuple space search algorithm [10] and the crossproducting algorithm [12], we discuss these two algorithms in detail.

2.1 Tuple Space Search

Each filter in a 2D filter set is specified as a pair of prefixes. We define the lengths of these prefixes as a tuple, denoted as (i, j), where i is the length of the source IP address prefix and j is the length of the destination IP address prefix. Hence, filters can be grouped into different tuples. Filters in a tuple can be easily stored in a hash table with the i-bit prefix of source IP address and the j-bit prefix of destination IP address as the key. Figure 1 illustrates the idea, in which each grid represents a tuple. Although there are $33 \times 33 = 1089$ tuples in total, it is possible that some tuples contain no filter at all so we do not assign hash tables for these tuples. The number of nonempty tuples in some ACL filter sets is reported in Table 1.

Table 1: Number of Nonempty Tuples

Filter Set	# filters	# tuples
ACL1	426	31
ACL2	527	50
ACL3	1,588	89
ACL-syn	6,826	31

When each nonempty tuple is assigned a hash table, the lookup can simply query all the hash tables to find the best matching filter. However, we cannot afford to perform so many hash queries per lookup for high performance packet classification. A simple optimization, called tuple space pruning, can help reduce the number of hash tables queried per lookup. This method performs single field lookups first to determine a subset of tuples for which there are matching filters. For example, assume there is a tuple (i, j) and we perform the LPM on the source IP address of a packet, obtaining the lengths of all the matching prefixes for the packet. If none of these lengths equals i, we do not need to search the tuple (i, j) at all. If the cost of performing LPMs can be kept low, this optimization can help improve the average-case performance. However, the worst-case performance remains the same. Another optimization, called rectangle search, aims to improve the worst-case performance. It uses the property that more specific filters have higher priority than less specific filters. For example, in Figure 1, if we find a matching filter in the tuple $(20, 12)$, shown as the dark grid, then we do not need to search the hash tables in the region A because all tuples in this region are less specific tuples than the one matched. If any of these filters did have higher priority than the filter in tuple $(20, 12)$, we would never match that tuple, making it redundant. With this optimization, the worst-case number of hash table queries is just $2w - 1$ as opposed to w^2 in the original scheme.

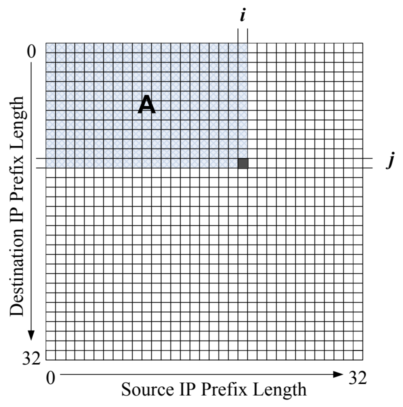

Figure 1: 2D Tuple Space Search

An alternate way to reduce the number of searches that must be performed is to group filters into *coarse-grained tuples* (CGT), rather than individual tuples. For example, we might group filters whose source address prefix lengths are in the range 21-24 and whose destination prefix lengths are in the range 9-16. If the number of CGTs is small, and we can search them quickly, we can obtain good worst-case performance. However, if we make the CGTs too large, the space required to represent the filters is likely to become excessively large. We can also use the tuple-space pruning technique to reduce the number of CGTs that must be searched in typical lookups. If the given packet's header does not have matching prefixes with lengths in the ranges defined by a given CGT, then we don't have to search that CGT.

Our approach can be used in combination with any method to search the CGTs. In this paper, we use cross-product tables to represent the filters in each CGT, since cross-product tables can be searched with a single hash lookup.

2.2 Crossproducting

The crossproducting algorithm is among the most straightforward ways to do packet classification. For each field, we assign each prefix on that field a unique ID. To classify a packet, the crossproducting algorithm performs LPMs on both fields first. The resulting IDs are combined to form an index, and then the index is used to retrieve the matching filter from a direct lookup table. If the source IP address field has m unique prefixes and the destination IP address field has n unique prefixes, the number of entries in the direct lookup table is $m \times n$. Although the lookup is very fast, the storage can be excessively large.

In the direct lookup table, not all the entries contain original filters. There are many "pseudo-filters" that come from the cross-products of original filters. These pseudo-filters must be stored and significantly expand the space used to represent the filter set. In Figure 2, A and C are nested prefixes in the source IP address field, as are B and D in the destination IP address field. Assume the prefix pair (A, B) is an original filter, $R1$. If the prefix pairs (A, D), (C, B), or (C, D) are not original filters, we need to add pseudo-filters for each to guarantee correct lookups. For example, if the single field lookups return the matching prefixes C and D, the results imply a match to the filter $R1$. Without the pseudo-filter (C, D), we will miss this match.

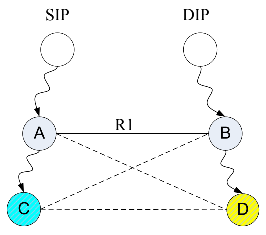

Figure 2: 2D Filter Expansion

The expanded filter set can be produced easily. Let s be any source prefix and d be any destination prefix. If there is an original filter (s_i, d_i) such that s_i is a prefix of s, and d_i is a prefix of d, then let (s_j, d_j) be the highest priority such prefix and include an pseudo-filter (s, d) that maps to (s_j, d_j). We evaluate the filter set expansion effect for some ACL filter sets. The expansion factors are shown in Table 2. The synthetic ACL filter set is expanded by more than 500 times.

Table 2: ACL Filter Set Expansion

Filter Set	Original filters	After Expansion	Table Entries
ACL1	426	19,885	29,294
ACL2	527	37,624	70,798
ACL3	1,588	222,396	408,157
ACL-syn	6,826	3,511,456	26,404,362

We can also see form the last column of Table 2 that 32% to 87% of entries in the direct lookup table actually do not contain any matching filter. Having the filters (p_{s1}, p_{d1}) and (p_{s2}, p_{d2}) does not necessarily mean we need to have two pseudo filters (p_{s1}, p_{d2}) and (p_{s2}, p_{d1}), because a match on (p_{s1}, p_{d2}) or (p_{s2}, p_{d1}) may not incur a match on any original filters. The empty table entries waste memory resources. This fact suggests that we use a hash table instead of a direct lookup table for better storage efficiency. By using FHT to implement the hash tables, we can achieve the similar lookup throughput as that of direct lookup tables.

3. COMBINING THE TWO ALGORITHMS

Using a hash table rather than a direct lookup table for the crossproducting algorithm can significantly reduce the storage requirement. However, if we keep all the filters in a single hash table, the expanded filter set due to the pseudo-filters may still be too large. To mitigate the filter expansion effect, we split the filters into several subsets and build a hash table for each of them. Now the pseudo-filters are only required for the cross-products of the filters in each subset. Because the number of nested prefixes on each field in each subset can be much smaller than that when all the filters are put together, the overall number of pseudo-filters can be significantly reduced. As a tradeoff, now we need to query multiple hash tables to find a matching filter.

Combining the above ideas with the CGT specification, our new algorithm allows a nice throughput-storage tradeoff and therefore is fast and scalable.

We defer the discussion of the methods used to group tuples to Section 5. Now assume we have partitioned the tuples into k groups. For each group, we store the filters in a hash table, adding pseudo-filters as needed to ensure that we can correctly identify the matching filter. Figure 3 illustrates a tuple partition. There are nine tuple sets from A to I in the figure. The tuple set I's specification is $([8, 16], [9, 21])$, for instance.

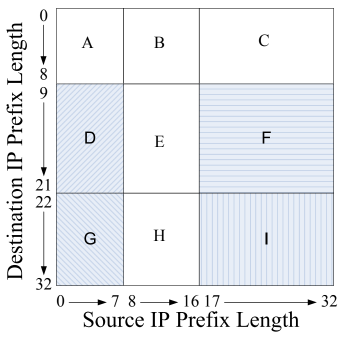

Figure 3: 2D Coarse-grained Tuple Space Partition

The lookup process is described as follows. We perform single field LPMs for both fields first. Since our coarse-grained tuples divide each IP address field into several segments, the LPMs need to return the longest matching prefix in each segment. We can easily adapt our new LPM algorithm discussed in Section 4 to support this. Through preprocessing, we embed such information in each SMT root so that the LPM performance is not affected. We use the results to determine the set of hash tables to query. Then we query these hash tables from more specific tuples to less specific tuples and terminate the search once the best matching filter is found. We use an example as shown in Figure 3 to illustrate the lookup algorithm. Suppose that for a given packet, the single field lookups show its source IP address has a matching prefix in segments $[0, 7]$ and $[17, 32]$, and its destination IP address has a matching prefix in segments $[9, 21]$ and $[22, 32]$. So the best matching filter can exist only in the hash tables for the tuple sets D, F, G, and I. Now the logical choice is to start the search from the tuple sets I, G, and F, because if we find matching filters belonging to these tuple sets, we do not need to search the tuple set D. If no match is found in these tables, we proceed to search the tuple set D.

With the above mentioned tuple partition, each tuple set is mapped to a grid in the 2D plane as shown in Figure 3. Choosing the number of segments for each field is a tradeoff of throughput and storage. At one extreme, when both fields have 33 segments, the algorithm regresses to a naive tuple space search algorithm. At another extreme, when both fields have only one segment, the algorithm regresses to a naive crossproducting algorithm.

To keep the LPM cost low, we use the LPM algorithm discussed in Section 4 to perform prefix matching on each field and the FHT data structure discussed in [8] to improve the hash query performance.

4. LONGEST PREFIX MATCHING

We can take advantage of on-chip memory to improve hash table lookup performance [8]. We consider applying this technique to further improve the LPM performance.

Using hash tables for LPM is not new. Dharmapurikar et. al. have presented a scheme to assign each unique prefix length a Bloom filter [4]. The queries to the Bloom filters are performed in parallel, and then the search for the longest matching prefix in an off-chip hash table starts from the longest length for which the corresponding Bloom filter reports a positive match. If no false positive is present, only one hash table query is needed to retrieve the best matching prefix.

However, LPM using Bloom filters has some disadvantages for IP lookups. Each distinct prefix length requires a Bloom filter, so the total number of Bloom filters can be large. Although the total number of items programmed in these Bloom filters is simply the number of prefixes in a table, the item distribution among these Bloom filters can be highly skewed. This makes engineering the system to best use the on-chip memory resource a challenging problem. In addition, a large number of Bloom filters leads to poor worst-case performance. If all the Bloom filters return a false positive, we need as many hash table queries as the number of Bloom filters for a packet lookup. Therefore, reducing the number of Bloom filters not only lowers the system complexity and but also improves the worst-case performance. To reduce the number of Bloom filters, the algorithm in [4] selects a few thresholds based on the prefix length distribution and expands the prefixes to their nearest thresholds. Now we need only one Bloom filter for each threshold. However, the prefix expansion increases the total number of prefixes in a route table a great deal, increasing the number of items that must be stored in both the on-chip memory and the off-chip hash table.

In addition to increasing the memory required, prefix expansion also significantly increases the incremental update cost. One single update might need a large number of memory operations on both the Bloom filter and the associated hash table. In an environment where the route table changes frequently, the update cost can become prohibitively large.

On the other hand, most successful IP lookup algorithms are essentially variations of the basic binary trie that allow for examining multiple bits per memory access [9]. Smart encoding techniques such as Tree Bitmap (TBM) [5] and Shape Shifting Tries (SST) [9] avoid the prefix expansion, improving storage efficiency and providing faster lookup throughput. However, searching in a trie always starts from the root, so the worst-case performance of these algorithms is proportional to the maximum trie depth [5] and is sensitive to the underlying trie structure [9].

Combining the hash table and trie data structures leads to a new LPM algorithm. It retains the memory efficiency of the trie-based algorithm and meanwhile allows the search to bypass intermediate trie nodes with the assistance of hash tables. The algorithm can be used in a high-performance IP lookup engine, especially for IPv6. It is suitable for hardware implementation and can sustain OC-192 and above line-speed processing by using only one commodity memory chip. The algorithm exhibits a nice tradeoff between throughput and storage, which allows system designers to decide the configurations based on the available on-chip and off-chip memory resource and the desired lookup throughput. In this paper we apply this LPM algorithm as a part of the packet classification algorithm.

4.1 Background

Some LPM algorithms take advantage of the trie data structure to support a pipelined architecture [6]. Ideally, pipelined lookups allow the completion of one packet lookup per clock cycle. Unfortunately, this technique has serious problems. It consumes too much memory bandwidth and the skewed storage requirement of the pipeline stages makes engineering the system difficult and inefficient. Another technique interleaves memory accesses from multiple parallel IP lookup engines [5, 9]. When these lookup engines share the same memory interface, they try to fully utilize the available memory bandwidth to gain a high aggregated lookup throughput. The bandwidth of a single SRAM chip today can be higher than 14 Gbps. Current VLSI technology makes it easy and low-cost to deploy multiple engines and synchronize their behavior. So the core problem here is to lower the bandwidth share of each engine. In other words, we should focus on reducing the number of off-chip memory accesses needed for a single packet lookup in order to achieve a higher overall lookup throughput.

The central piece of our LPM algorithm is a set of on-chip Bloom filters. As discussed in [8], Bloom filters have drawn significant attention in the networking research community recently due to their efficient use of memory. Reference [4] discusses using Bloom filters for IP lookups. Our work is built upon this algorithm and significantly improves it.

Sangireddy et. al. present an Elevator-Stair algorithm that combines hash tables and PATRICIA trees [7]. Hash tables are built on selected levels to indicate if there are longer prefixes starting from these levels. However, as the name of algorithm implies, the LPM starts from the tree root, searching the hash tables level by level to determine where to find the potential longest matching prefix. While this is akin to our algorithm, our algorithm supports directly jumping to the destination hash table, resulting in a faster search speed. Their algorithm uses the PARTRICIA tree for the second layer search. However, the PARTRICIA tree can only compress tree paths without any branch. Hence it is not as effective as other encoded multibit trie algorithms. Moreover, the algorithm does not use Bloom Filters to summarize the items in hash tables, so the algorithm has to physically access many of the off-chip hash tables.

4.2 LPM using Hash Table and Trie

The easiest way to organize data for IP lookup is to group the prefixes based on their lengths and store each group in a hash table. When lookups are performed in software, the binary search on these hash tables based on the prefix lengths is the best choice [13], resulting in the $O(\log W)$ lookup time performance, where W is the number of unique prefix lengths. When lookups are performed in hardware, however, we can take advantage of the embedded fast memory and the parallel processing capability of hardware to use the brute force method. As proposed in [4], an on-chip Bloom filter is used to summarize the items in each hash table. The lookup process probes all the Bloom filters simultaneously and uses the output of the Bloom filters to determine which hash table to query. In practice, the lookups can be very fast. Unfortunately, due to the possibility of false positive in Bloom filters, the worst-case lookup time performance is as poor as $O(W)$. When using the prefix expansion technique [11] to reduce W, i.e. the number of Bloom filters and hash tables, significantly more storage is required.

The high level idea of our algorithm is simple: with the reduced number of Bloom filters, instead of performing the prefix expansion, we encode the subtree between two length thresholds using the TBM or SST encoding technique. In a sense our new algorithm can be seen as a multi-bit trie algorithm with multilevel jump tables.

For example, assume we have a prefix table shown in Table 3. If we assign each unique length a Bloom filter, we need at least five Bloom filters. Future updates can drastically change the situation so more Bloom filters are expected. Here we get a sense of the difficulty of engineering such a system. Now we assume the table is just as it is. Six items are programmed in the Bloom filters and in the worst case we need five hash table queries to find the best matching prefix when all the Bloom filters show a false positive.

Table 3: Prefix Table

ID	Prefix
p0	*
p1	1*
p2	000*
p3	101*
p4	1000*
p5	10010*
p6	1001101*

Now we want to use the prefix expansion technique to reduce the number of Bloom filters to two. By carefully analyzing the prefix length distribution, we decide to set the two length thresholds to 4 and 7. The expanded table is shown in Table 4. The table size is doubled and 15 items need to be programmed in the two Bloom filters. Although the worst-case number of hash table queries is reduced to only 2, the storage required is significantly increased. Moreover, if now we need to remove the prefix $p1$, we need to remove five items from the on-chip Bloom filters and the off-chip hash tables. This is a high update cost.

Rather than expanding the prefix table, our algorithm seeks to encode the prefixes between the length thresholds using the trie data structure. As shown in Figure 4, the binary trie nodes are grouped into subtrees and the subtrees are encoded using either TBM or SST. Now in the first Bloom filter, we need to program only two items "10010" and "10011", and in the second Bloom filter, we need to program only one item "1001101". The root nodes of subtrees,

Table 4: Expanded Prefix Table

ID	Prefix	ID	Prefix
p0	*	p1	1101*
p2	0000*	p1	1110*
p2	0001*	p1	1111*
p4	1000*	p5	1001000*
p1	1001*	p5	1001001*
p3	1010*	p5	1001010*
p3	1011*	p5	1001011*
p1	1100*	p6	1001101*

associated with the items in the Bloom filters, are stored in the off-chip hash tables. In addition, the best matching prefix so far for each item is also stored along with the item in hash tables. For example, in the hash table entry associated with the item "10010", the best matching prefix so far is itself, "10010*". However, in the hash table entry associated with the item "10011*", the best matching prefix so far is $p1$ or "1*". Now there are only three items in the Bloom filters. Compared with the previous scheme, it is a huge saving.

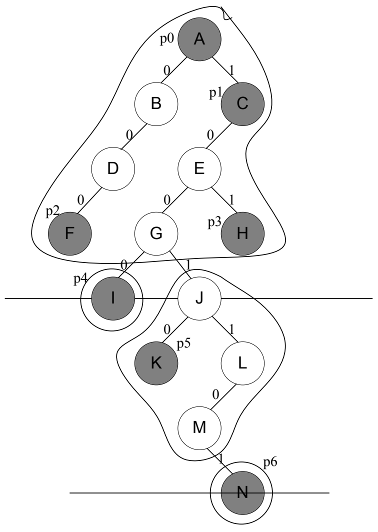

Figure 4: LPM Data Structure

There are some subtle points about this data structure. First, the items programmed in the Bloom filters may not be the prefixes in the Table. Rather, they are the prefixes of the paths that cross the length thresholds. See the path "1001101" in Figure 4 for an example. Second, if a long path cross multiple thresholds, then the prefixes of this path with different threshold lengths are programmed in multiple Bloom filters. We created a sort of dependency among the Bloom filters. This feature can help filter out certain false positive pattern. A match in a Bloom filter for a longer threshold can be a true match only if all the Bloom filters for the shorter thresholds show matches too. In other words, if any Bloom filter shows a mismatch, we know that the matches in Bloom filters for longer thresholds are definitely false positive, so we do not need to query their associated off-chip hash tables.

With this data structure, the LPM lookups are quite simple. Give an IP address, we extract the prefixes according to the length thresholds of the Bloom filters and then use these prefixes to query the Bloom filters in parallel. We then determine which hash table to query based on the Bloom filter outputs. If a Bloom filter and all the Bloom filters for shorter length thresholds report a match, we then query the hash table for this Bloom filter to verify the match. If it turns out to be a true match, then the best matching prefix is either the one stored in the hash table entry or a longer prefix in the subtree. So we traverse the subtree to search for a longer prefix. The best matching prefix is returned according to the search result. If the query to a hash table shows that a match in a Bloom filter is a false positive, then we go ahead to query the hash table for the Bloom filter with a shorter length threshold.

This architecture suggests that the worst-case lookup performance is determined by the number of Bloom filters and the cost to traverse a subtree. In the example shown in Figure 4, we can read a subtree in just one memory access, so in the worst case, a packet lookup needs two hash table queries to retrieve a valid multibit trie node and one extra memory access to retrieve the next hop per lookup. For this example, the worst-case performance is identical to that of the original method with prefix expansion. However, our algorithm uses much less memory and provides better support of incremental updates.

Now we describe the algorithm formally. The data structure construction algorithm starts from the binary prefix tree. The tree is partitioned into k segments at depth d_0, d_1, d_2, ... d_k, where $0 = d_0 < d_1 < d_2 < ... < d_k \leq w$. We then assign an on-chip Bloom filter B_i and an off-chip hash table H_i for each depth d_i when $i > 0$. For any path starting from the root with its length $\geq j$, there is a record in each Bloom filters B_r if $d_r \leq j$. In the Bloom filter B_i, the d_i-bit prefix of the path is programmed. A unique path prefix is only programmed in a Bloom filter once. Since the prefixes of a path with different lengths are present in a sequence of Bloom filters, we call it *Chained Path Bloom Filters* (CPBF). All the paths ended in a segment (d_i, d_{i+1}) forms a set of subtrees for which the roots are the tree nodes at the depth d_i. We use either Tree Bitmap or Shape Shifting Tries to encode these subtrees. Each such encoded subtree is called a *Segment Multibit Trie* (SMT). The stride or the node capacity is determined by the word size of the off-chip memory. All the path prefixes programmed in a Bloom filter B_i are also stored in the hash table H_i. Along with the path prefixes, the hash table stores the corresponding SMT root node and the length of the longest matching prefix of the root node.

The IP lookup process includes two steps. First, construct the Bloom Filter keys, query the Bloom Filters, and use the outputs to determine which Hash Table to search. Second, retrieve the best match so far and the SMT root from the Hash Table, traverse the SMT, and determine the best match.

In the first step, we use the prefixes of the IP address with length $d_1, d_2, ...d_k$ as keys to query the corresponding Bloom filters in parallel. We examine the match status from B_1 to B_k. If the first negative match is reported by B_j, then the length of the longest matching prefix must be shorter than d_j, even if some Bloom filters with index greater than j report a positive match. The dependency of the CPBF is able to filter out this kind of false positive without requiring any off-chip memory access. If $j = 1$, we know the best match exists in the SMT between depth 0 and depth d_1; otherwise, we query the hash table H_{j-1} to verify the match. If it turns out the match in B_{j-1} is a false positive, we then back to query the hash table with smaller index and so on. Finally, we can find exactly the segment which contains the best match. Once we find a true match in a hash table, in the second step, we retrieve the associated SMT root and traverse the SMT to find a longer matching prefix. The longest matching prefix in this SMT is returned as the best match. If the search fails, the stored best prefix is returned. The best matching prefix can then be used as a key to retrieve the associated information, such as the next hop for IP lookups.

CPBF provides a mechanism to fast jump the search to the target segment when doing LPM. The encoded SMT efficiently uses the memory and supports fast lookups. The combination of these two forms a more scalable and faster LPM algorithm.

4.3 Implementation Consideration

When designing the LPM system, we need to determine the number of Bloom filters as well as the set of length thresholds. Our algorithm is very flexible: each segment can have a different number of bits. We can optimize the segment partition to fit different applications. When the algorithm is used for IP lookups, we can engineer the design to gain the best throughput. For example, if we use TBM to encode the SMTs and one memory word can encode a node with a stride of s, we can partition the binary tree into $\lceil W/s \rceil$ segments, where W is the longest length of the prefixes in a table. So $\lfloor W/s \rfloor s$ Bloom filters are needed to cover path lengths s, $2s$, ... , $\lfloor W/s \rfloor s$. Since an SMT is encoded using a single memory word, one to $\lfloor W/s \rfloor$ off-chip memory accesses are needed to find the best matching prefix. Here we assume each hash table query requires just one off-chip memory access, which is reasonable with the use of FHT for implementing the hash tables.

If the on-chip memory resource becomes a concern, we can reduce the number of Bloom filters by letting each segment cover αs bits. Now only $\lfloor W/\alpha s \rfloor$ Bloom filters are needed and an SMT could have a depth of α. This means one to $\lfloor W/\alpha s \rfloor + \alpha$ memory accesses are required to find the best matching prefix. Our algorithm allows a tradeoff between throughput and storage. This is especially attractive for IPv6, where W is a large number. Assume the longest prefix length is 64 and the memory word size supports TBM nodes with a stride of 5, Figure 5 shows the number of Bloom filters required versus the worst-case number of memory accesses required for a packet lookup when we vary the value of α. The upper curve shows the absolute worst-case performance when Bloom filters can show false positives. If we assume there is no false positive from the Bloom filters, the worst-case performance is determined by the depth of SMTs, which is shown in the lower curve in Figure 5.

When no Bloom filter is used, the performance is simply that of the multibit trie algorithm. When just one Bloom filter is used, the performance improves almost two times. While more Bloom filters tend to worsen the absolute worst-case performance, the average-case performance and the usual performance are both improved substantially.

In [9] we take advantage of the tree sparsity to encode the subtrees using SST which can generally cover a larger stride per node. Similarly, we can expect a better performance if SST can also be used to encode SMTs. We use the following

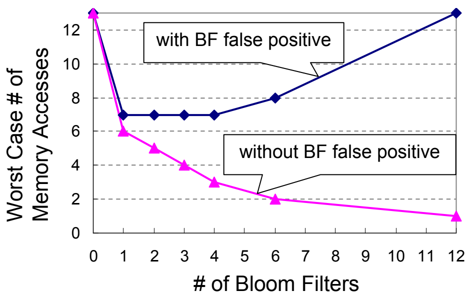

Figure 5: The Worst-Case Performance vs the Number of Bloom Filters

algorithm to partition the binary tree and construct the data structure dynamically.

Step 1 Traverse the binary tree and find the largest depth d_i such that every subtree rooted at depth d_i can be encoded as an SMT of depth k, for some specified k. The SMTs combine the TBM-type and SST-type nodes in order to minimize d_i. If $d_i > 0$, go to the step 2. Otherwise, the binary tree root is reached, so encode the subtree rooted at the binary tree root and halt.

Step 2 Build a Bloom filter for depth d_i. All the tree nodes at depth d_i have a record in the Bloom filter. Encode the subtrees and store them in the associated hash table. Store the best matching prefix so far for each item in the associated hash table.

Step 3 Prune the binary tree at depth d_i and repeat the previous steps on the remainder tree.

Figure 6 shows an example of such a tree partition when $k = 1$. Assume an SST node can encode five binary nodes and an TBM node can support a stride of 3. At the first iteration, we get the threshold at the depth 4. At the second iteration, we get the second threshold at the depth 1. The algorithm terminates at the third iteration.

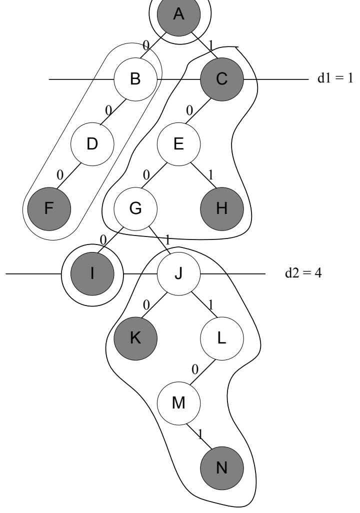

Figure 6: Tree Partition Using TBM and SST

Using this algorithm, the size of segments is not necessary to be the same, but adapts to the structure of the binary tree. Hence, we can reduce the required number of Bloom filters to the minimum. Note that we can also increase the desired SMT depth to further reduce the number of Bloom filters required at the cost of a little lower throughput.

4.4 LPM Performance

Assume there is no false positive from Bloom filters. Hence, traversing one SMT is the only cost incurred to find the longest matching prefix. In the worst case when all the k Bloom filters show a false positive, the extra cost is k hash queries. In this case, the performance is the same as the worst-case performance of the multi-bit trie algorithm.

Assume the best matching prefix is stored in H_i. Clearly, B_i should report a true match. We need to access only one SMT if and only if B_{i+1} does not show a false positive, so the probability is $(1-f)$, where f is the false positive rate of a Bloom Filter. Similarly, we need to access two SMTs if and only if B_{i+1} shows a false positive and B_{i+2} does not show a false positive, so the probability is $f(1-f)$. Therefore, the average number of STMs accessed per packet lookup is:

$$(1-f)+2f(1-f)+...+(k-i)f^{k-i-1}(1-f)+(k-i+1)f^{k-i}$$

f is typically a very small value, so we can let $1-f = 1$. Assume for the lookups, the best matching prefixes are evenly distributed in all the hash tables, then the average number of STMs accessed per packet lookup is:

$$1+\frac{2(k-1)}{k}f+\frac{3(k-2)}{k}f^2+...+\frac{2(k-1)}{k}f^{k-2}+f^{k-1}$$

A QDRII SRAM has an equivalent word size of 72 bits, which is sufficient to encode a TBM node with a stride of 5 or an SST covering a 16-node binary subtree [9][1]. We use the largest available BGP route table to demonstrate our algorithm's performance. There are about 200K prefixes in this table, and the lengths are distributed between 8 and 32. We evaluate our algorithm using only five Bloom filters for path lengths of 8, 13, 18, 23 and 28. This partition ensures that each SMT contains a single TBM node.

Table 5: BGP Table Results I

Depth	# SMTs	# Expanded Prefixes
8	128	22
13	2,648	405
18	23,689	47,155
23	71,913	286,769
28	10,333	1,319,789
32	—	89,640
Total	108,711	1,743,780

As shown in Table 5, the number of SMTs defines the number of items that must be stored in both the on-chip Bloom filters and the off-chip hash tables. If the original method was used with Bloom filters for lengths 8, 13, 18,

[1]When each SMT can be encoded using a single node, the node data structure does not need the EBM and the child pointer fields. Therefore, the node can support a larger stride or can cover more binary nodes. However, here we use the conservative numbers for evaluation

23, 28, and 32, the number of items stored corresponds to the number of expanded prefixes, shown in the right column. Thus, our method reduces the storage required by more than an order of magnitude.

We can reduce the number of Bloom filters further at the cost of one more memory access per lookup. The results are shown in Table 6. By reducing the number of Bloom filters, our algorithm needs fewer and fewer items in Bloom filters. However, the prefix expansion scheme needs more and more items. By eliminating two Bloom filters, now the prefix expansion scheme generates more than 50 expanded prefixes per original prefix on average.

Table 6: BGP Table Results II

Depth	# SMTs	# SMT nodes	# expanded prefixes
8	128	2,776	22
18	23,689	95,602	58,240
28	10,333	10,333	10,206,672
32	—	—	89,640
Total	34,150	108,711	10,354,574

Figure 7 shows the worst-case number of memory accesses per packet lookup versus the number of items stored per original prefix, for our algorithm and the prefix expansion scheme. Our algorithm provides a good worst-case performance with very low storage overhead. For the prefix expansion scheme to reach the similar worst-case performance, significantly more storage is required.

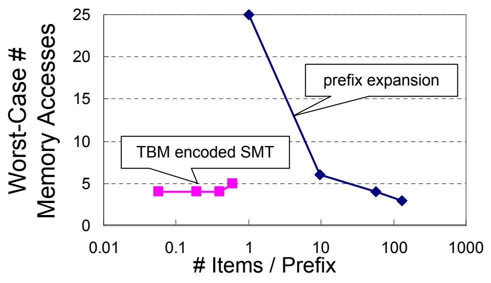

Figure 7: The Worst-Case Performance vs the Number of Items per Prefix

Finally, we investigate the effect of dynamically determining the segment sizes using SST and TBM together on the synthetic IPv6 table we used in [9]. There are 47 unique prefix lengths distributed from 18 to 64. If the naive Bloom filter scheme is applied, up to 47 Bloom filters are required and the worst-case performance is 47 hash table queries per packet lookup. If TBM is used to encode the SMTs and the memory word supports a stride of 5, then we can reduce the number of Bloom filters to just 10 and the worst-case performance now is 10 hash table queries per packet lookup. Table 7 summarizes the results. Because the binary trie is very sparse, the number of SMTs exceeds the number of expanded prefixes, so we can see the storage of our scheme is actually worse than that of the prefix expansion scheme. However, if we reduce the number of Bloom filters further, the number of SMTs will decrease and the number of expanded prefixes will increase.

By using SST and TBM together, an encoded SMT node can cover either 16 binary tree nodes or support a stride of 5. Letting each SMT contain a single node, we run the dynamic algorithm to determine the segment size and reduce the number of Bloom filters to 7. Now in the worst case, we need just seven hash queries per packet lookup to find the best matching prefix. Table 8 summarizes the results. The table also shows when the prefix expansion is used to support these thresholds, the table size is expanded to almost 100 times larger, while in our scheme, the items stored in Bloom filters are only 3.5 times more than the number of original prefixes. When a single QDRII SRAM chip is used, our algorithm can perform lookups for 200 million packets per second in the usual case and 25 million packets per second in the worst case. With a false positive rate of 0.008, our algorithm requires 12 Mb on-chip memory when FHT is used to implement the Bloom filters and the associated hash tables. On the other hand, the prefix expansion scheme requires 339 Mb on-chip memory.

Table 7: IPv6 BGP Table Results (TBM only)

Depth	# SMTs	# expanded prefixes
18	31,667	—
23	86,617	142
28	126,489	1,285
33	130,347	18,819
38	128,264	35,464
43	111,077	86,807
48	85,447	204,972
53	28,730	133,787
58	24,141	34,729
63	13,460	51,464
64	—	11,840
Total	766,239	579,309

Table 8: IPv6 BGP Table Results (SST and TBM)

Depth	# SMTs	# expanded prefixes
18	31,667	—
21	56,935	24
26	116,436	632
31	132,874	3,667
36	130,320	99,013
41	117,051	73,021
49	38,465	1,083,665
64	—	16,515,728
Total	623,748	17,775,750

We can see our new LPM algorithm has significant advantages over the previous algorithms for IP lookups. In our 2D packet classification algorithm, it allows us to significantly reduce the cost of single field lookups.

5. TUPLE PARTITION

If we visualize the tuple space in a 2D plane, the key for good performance of our packet classification algorithm is to come up with effective tuple partitions. There are two dimensions to approach this problem. First, if we are given the acceptable worst-case throughput, we seek to minimize the storage. Second, if we are given the storage we can use, we seek to maximize the throughput.

Now we consider the first approach. With the simple grid tuple definition, once the worst-case number of hash queries

Table 9: Filter Set Expansion for Different Configurations

Filter Set	# filters	1×1	1×2	2×1	2×2	2×4	4×2	3×3
ACL1	426	19,885	2,851	997	472	445	472	445
ACL2	527	37,674	9,317	8,978	1,225	922	899	596
ACL3	1,588	222,396	55,046	28,537	3,212	3,160	1,779	1,737
ACL-syn	6,826	3,511,456	384,353	34,579	9,666	7,992	9,666	7,992

is chosen, we need to determine how to assign the number of segments and the boundary of each segment on each field. The goal is to minimize the size of the expanded filter set. This optimization problem can be achieved through dynamic programming. In practice, we need also to consider the performance of LPM so it is better to have regular sized segments. Fortunately, we find that the regular sized segments can result in good performance.

Besides the grid-based partition, it turns out that we can have arbitrary tuple partitions. Figure 8 shows a simple example. There are only three tuple sets. The tuple set B can be represented as $([12, 23], [1, 25]) \cup ([1, 11], [9, 25])$, for instance. Note that we have removed the tuple sets $(0, 0)$, $(0, [1, 32])$, and $([1, 32], 0)$ from the tuple space, because the filters in them can be handled by the LPMs on both fields so that these filters do not need to be stored in any hash table. The lookup process for such partitions is almost the same with a minor difference. It is possible we find in a tuple we may have multiple prefix length combinations that we need to check. However, we only need to check the most specific combinations, thanks to the "pseudo filters". Hence, the number of hash table queries is still bounded by the number of tuples.

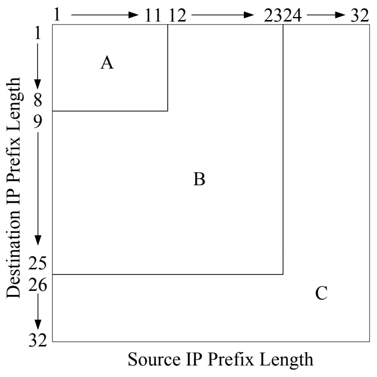

Figure 8: Another Tuple Partition Scheme

We can preset the tuple partition in favor of the LPM implementation. In this case, we have little control of the filter expansion but we can guarantee the worst-case performance. On the other hand, we can dynamically determine the tuple partition by constraining the filter set expansion so we can control the storage but not the worst-case performance. For example, we may want the overall filter set expansion factor to be no more than α. On way to achieve this is to partition the space incrementally. If the expansion ratio for a particular tuple set is too high, we divide it into smaller subsets.

While simple tuple partitions seem to work well on current filter sets, it makes sense to study better tuple partition algorithms for larger filter sets in the future. We believe there are many opportunities for future work in this direction.

6. EVALUATION

We first evaluate the grid tuple partition. We perform experiments on several ACL filter sets and summarize the results in Table 9 and Figure 9. In the first row of the table, $\alpha \times \beta$ means that the source IP address field has α equal sized segments and the destination IP address field has β equal sized segments. Therefore, the values of $\alpha \times \beta$ give the worst-case number of hash queries for a packet. In the figure, the dotted lines indicate the original size of the filter sets. We can see that at the cost of a very small number of hash queries, the size of the expanded filter set decreases quickly to approach its original size.

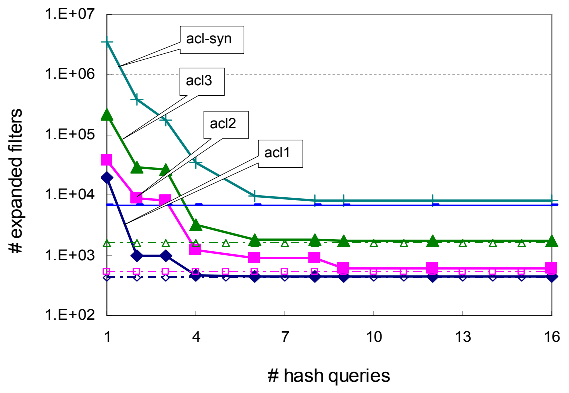

Figure 9: Filter Set Expansion vs. Number of Hash Queries

We use the LPM algorithm discussed in Section 4 to perform single field lookups. We assign Bloom filters at the same segment boundaries. Table 10 shows the total number of items that need to be programmed into the Bloom filters for LPM. The numbers are typically very small, which implies a small amount of on-chip memory usage.

We evaluate the algorithm performance when using irregular tuple partitions as shown in Figure 10. There are two to four tuple sets for different configurations. The tuple sets are equalled spaced on each axis. The simulation results are shown in Table 11. The size of the expanded filter sets is significantly reduced when a finer tuple partition is used. Sometimes when a finer tuple partition is used, the overall number of filters in the expanded filter set does not change. This is because some tuple sets do not contain any filter at all. Actually, for these evaluated filters sets, at most two tuple sets need to be searched even with the configuration (c) where there are four tuple sets, so in the worst case, a packet lookup requires only two hash table queries.

Finally, we dynamically determine the tuple partition by setting the filter set expansion factor to 2. For ACL2 and

Table 10: Number of Items in Bloom Filters for LPM

Filter Set	1 × 1	1 × 2	2 × 1	2 × 2	2 × 4	4 × 2	3 × 3
ACL1	0	61	3	64	184	72	120
ACL2	0	49	43	92	163	160	207
ACL3	0	59	57	116	368	310	385
ACL-syn	0	387	191	578	1,168	943	1,103

Table 11: # Filters for Different Tuple Configurations

Filter Set	# filters	Single Tuple	Config. (a)	Config. (b)	Config. (c)
ACL1	426	1,004	1,004	1,004	542
ACL2	527	3,027	3,027	2,604	1,914
ACL3	1,588	174,561	6,730	6,730	6,692
ACL-syn	6,826	185,799	185,799	185,799	12,306

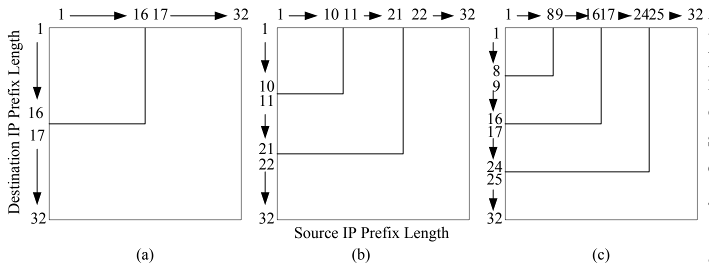

Figure 10: Experiments on Irregular Tuples

ACL3, we cannot achieve this goal with the approach shown in Figure 8. This means we need better tuple partition algorithms. For ACL1 and ACL-syn, we get the tuple partition shown in Figure 11. The figure also shows the number of filters before and after filter set expansion. The results show that two tuple sets are enough to satisfy our storage constraint.

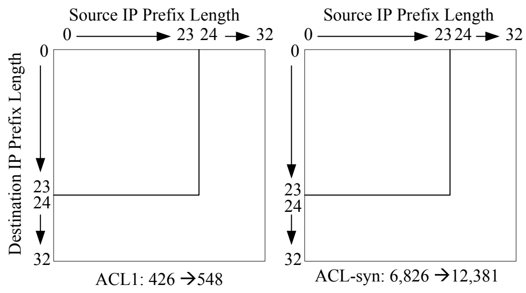

Figure 11: Dynamic Tuple Partition

7. CONCLUSION

In this paper we deal with a special case of the packet classification problem where each filter is specified as two prefixes. We present a novel packet classification algorithm derived from the tuple space search algorithm and the crossproducting algorithm. When implemented with our LPM and FHT techniques, the algorithm performs much better than previous algorithms for 2D IPv4 packet classification. It becomes even more attractive in IPv6 scenarios, where the previous algorithms suffer from the much longer prefixes. With a flexible tuple partition scheme, our algorithm exhibits an attractive tradeoff between storage and throughput, which allows the designer to control the system performance based on the available resource and the desired classification throughput. Our algorithm is fast yet yields small on-chip and off-chip memory consumption. Moreover, our LPM algorithm alone significantly improve previous algorithms and can be used for high performance IP lookups.

8. REFERENCES

[1] F. Baboescu, S. Singh, and G. Varghese. Packet classification for core routers: Is there an alternative to CAMs? In *IEEE INFOCOM*, 2003.

[2] M. Buddhikot, S. Suri, and M. Waldvogel. Space Decomposition Techniques for Fast Layer-4 Switching. In *Conference on Protocols for High Speed Networks*, 1999.

[3] E. Cohen and C. Lund. Packet Classification in Large ISPs: Design and Evaluation of Decision Tree Classifiers. In *ACM SIGMETRICS/Performance*, 2005.

[4] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest Prefix Matching using Bloom Filters. In *ACM SIGCOMM*, Aug. 2003.

[5] W. Eatherton, G. Varghese, and Z. Dittia. Tree Bitmap: hardware/software IP Lookups with Incremental Updates. *ACM SIGCOMM Computer Communication Review*, 2004.

[6] J. Hasan and T. N. Vijaykumar. Dynamic Pipelining: Making IP Lookup Truly Scalable. In *ACM SIGCOMM*, 2005.

[7] R. Sangireddy, N. Futamura, S. Aluru, and A. K. Somani. Scalable, Memory Efficient, High-Speed IP Lookup Algorithms. *IEEE/ACM Transactions on Networking*, 13, Aug. 2005.

[8] H. Song, S. Dharmarpurikar, J. Turner, and J. Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *ACM SIGCOMM*, 2005.

[9] H. Song, J. Turner, and J. Lockwood. Shape Shifting Tries for Faster IP Lookup. In *IEEE ICNP*, 2005.

[10] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *ACM SIGCOMM*, 1999.

[11] V. Srinivasan and G. Varghese. Fast Address Lookups using Controlled Prefix Expansion. *ACM Transaction on Computer Systems*, 17, Feb. 1999.

[12] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Layer Four Switching. In *ACM SIGCOMM*, 1998.

[13] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *ACM SIGCOMM*, 1997.

[14] P. Warkhede, S. Suri, and G. Varghese. Fast Packet Classification for Two-Dimensional Conflict-Free Filters. In *IEEE INFOCOM*, 2001.