Washington University in St. Louis

School of Engineering & Applied Science

Fast Packet Classification Using Bloom Filters

Sarang Dharmapurikar
Haoyu Song
Jonathan Turner
John W. Lockwood

WUCS-2006-27

May 12, 2006

Department of Computer Science
Applied Research Lab
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130

Abstract

While the problem of general packet classification has received a great deal of attention from researchers over the last ten years, there is still no really satisfactory solution. Ternary Content Addressable Memory (TCAM), although widely used in practice, is both expensive and consumes a lot of power. Algorithmic solutions, which rely on commodity memory chips, are relatively inexpensive and power-efficient, but have not been able to match the generality and performance of TCAMs.

In this paper we propose a new approach to packet classification, which combines architectural and algorithmic techniques. Our starting point is the well-known crossproducting algorithm, which is fast but has significant memory overhead due to the extra rules needed to represent the crossproducts. We show how to modify the crossproduct method in a way that drastically reduces the memory required, without compromising on performance. We avoid unnecessary accesses to off-chip memory by filtering off-chip accesses using on-chip Bloom filters. For packets that match p rules in a rule set, our algorithm requires just $4 + p + \epsilon$ independent memory accesses on average, to return all matching rules, where $\epsilon \ll 1$ is a small constant that depends on the false positive rate of the Bloom filters. Each memory access is just 256 bits, making it practical to classify small packets at OC-192 link rates using two commodity SRAM chips. For rule set sizes ranging from a few hundred to several thousand filters, the average rule set expansion factor attributable to the algorithm is just 1.2. The memory consumption per rule is 36 bytes in the average case.

1 Introduction

The general packet classification problem has received a great deal of attention over the last decade. The ability to classify packets into flows based on their packet headers is important for QoS, security, virtual private networks (VPN) and packet filtering applications. Conceptually, a packet classification system must compare each packet header received on a link against a large set of rules, and return the identity of the highest priority rule in the set that matches the packet header (or in some cases, all matching rules). Each rule can match a large number of packet headers, since the rule specification supports address prefixes, wild cards and port number ranges. Much of the research to date has concentrated on algorithmic techniques which use hardware or software lookup engines, which access data structures stored in commodity memory. However none of the algorithms developed to date have been able to displace TCAMs, in practical applications.

TCAMs offer consistently high performance, which is largely independent of the characteristics of the rule set, but they are relatively expensive and use large amounts of power. A TCAM requires a deterministic time for each lookup, and recent devices can classify more than 100 million packets per second. Although TCAMs are a favorite choice of network equipment vendors, alternative solutions are still being sought, primarily due to the high cost of the TCAM devices and their high power consumption. The cost per bit of a high performance TCAM is about 15 times larger than a comparable SRAM [2], [1] and they consume more than 50 times as much power, per access [16],[13]. This gap between SRAM and TCAM cost and power consumption makes it worthwhile to continue to explore better algorithmic solutions.

In this paper we introduce an algorithmic solution which is both fast and highly memory efficient. It is based on the well-known "Crossproducting Algorithm" [11]. The crossproducting algorithm decomposes the packet classification problem into a set of single field lookup problems and combines the results to form a key to retrieve the best matched rule from a direct-lookup table. From the throughput perspective, the single field lookups are the only real performance bottleneck. However, the major problem with this algorithm is its prohibitively high memory consumption due to the large number of additional "crossproduct rules" that must be added to the rule set. Even small rule sets can require impractically large amounts of memory.

Leveraging recent advances in algorithms and architectures, we introduce some new ideas to address these problems. In particular, our Multi-Subset Crossproducting Algorithm significantly reduces this memory overhead while preserving the overall speed of the algorithm. First of all, we perform the single field lookup by longest prefix matching (LPM) on each field, using the fast and memory efficient Bloom filter based algorithm introduced in the Chapter [4]. Using this algorithm, on an average, only one off-chip memory access is needed for each single field lookup. Therefore, with very high probability, the longest prefix matching can be performed on the source and destination addresses and the source and destination ports in just four memory accesses.

To reduce memory consumption, we divide the rules into multiple subsets and then construct a crossproduct table for each subset. This reduces the overall crossproduct overhead drastically. In addition, instead of a direct lookup table, a hash table is used to further reduce the size of the crossproduct lookup table. Since the rules are divided into multiple subsets, we need to perform a lookup in each subset. However, we can use Bloom filters to avoid lookups in subsets that contain no matching rules, making it possible to sustain high throughput. In particular, we show that the highest priority matching rule can be found using only p more memory accesses, where p is the number of rules a packet can match. In summary, we demonstrate a method, based on Bloom filters and hash tables, that can classify a packet in $4 + p + \epsilon$ memory accesses where ϵ is a small constant $\ll 1$ determined by the false positive probability of the Bloom filters. With two memory chips, one for the LPM tables and the other for rule tables, the LPM phase of 4 memory accesses and rule lookup phase of p memory accesses can be pipelined. With pipelining, the memory accesses per packet can be reduced to $max\{4, p\}$. We also show how a special case of our algorithm is in fact a highly optimized variant of the well-known Tuple Space Search algorithm proposed by Srinivasan et. al. [10] We also discuss the underlying architectural issues in realizing this method in hardware.

We leverage some of the existing work on Bloom filters and hardware implementation to design our packet classification system. Our results show that our architecture can handle large rule sets, containing hundreds of thousands of rules, efficiently with an average memory consumption of 30 to 36 bytes per rule.

The rest of the chapter is organized as follows. In the next section we discuss the related work. We describe the naive crossproducting algorithm in more details in Section 3. In Section 4, we discuss our Multi-Subset Crossproducting Algorithm. In Section 5 we describe our heuristics for intelligent partitioning of the rules into subsets to reduce the overall crossproducts. Finally, in Section 6 we discuss the architectural issues in implementing our algorithm in hardware. Section 7 concludes the chapter.

2 Related Work

There is a vast body of literature on packet classification. An excellent survey and taxonomy of the existing packet classification algorithms and architectures can be found in [13]. Here, we discuss only the algorithms that are closely related to our work.

Algorithms that can provide deterministic lookup throughput is akin to the basic crossproducting algorithm [11]. The basic idea of the crossproducting algorithm is to perform a lookup on each field first and then combine the results to form a key to index a crossproduct table. The best-matched rule can be retrieved from the crossproduct table in only one memory access. The single field lookup can be performed by direct table lookup as in the RFC algorithm [6], or by using any range searching, or LPM algorithm. The BV [7] and ABV [3] algorithms use bit vector intersections to replace the crossproduct table lookup. However, the width of a bit vector equals to the number of rules and each unique value on each field needs to store such a bit vector. Hence, the storage requirement is significant, which limits its scalability.

Using a similar reduction tree, the DCFL [12] algorithm uses hash tables rather than direct lookup tables to implement the crossproduct tables at each tree level. However, depending on the lookup results from the previous level, each hash table needs to be queried multiple times and multiple results are retrieved. For example, at the first level, if a packet matches m nested source IP address prefixes and n nested destination IP address prefixes, we need $m \times n$ hash queries to the hash table with the keys that combine these two fields and the lookups typically result in multiple valid outputs that require further lookups. For a multi-dimensional packet classification, this incurs a large performance penalty.

TCAMs are widely used for packet classification. The latest TCAM devices also include the banking mechanism to reduce the power consumption by selectively turning off the unused banks. Traditionally, TCAM devices needed to expand the range values into prefixes for storing a rule with range specifications. The recently introduced algorithm, DIRPE [8], uses a clever technique to encode ranges differently which results in less overall rule expansion compared to the traditional method. The authors also recognized that in modern security applications, it is not sufficient to stop the matching process after the first match is found but all the matching rules for a packet must be reported. They devised a multi-match scheme with TCAMs which involves multiple TCAM accesses.

Yu et. al. described a different algorithm for multi-match packet classification based on geometric intersection of rules [15]. A packet can match multiple rules because the rules overlap. However, if the rules are broken into smaller sub-rules such that all the rules are mutually exclusive then the packet can match only one rule at a time. This overlap-free rule set is obtained through geometric intersection. Unfortunately, the rule set expansion due to the newly introduced rules by the intersection can be very large. In [16], they describe a modified algorithm called SSA which reduces the overall expansion. They observe that if the rules are partitioned into multiple subsets in order to reduce the overlap then the resulting expansion will be small. At the same time one would need to probe each subset independently to search a matching rule. In a way, our algorithm is similar to SSA in that we also try to reduce the overlap between the rules by partitioning them into multiple subsets and thus reduce the overall expansion. However, while SSA only

cares about an overlap in all the dimensions, our algorithm considers the overlap in any dimension for the purpose of partitioning. Hence the partitioning technique are different. Moreover, SSA is a TCAM based algorithm whereas ours is memory based. Finally, SSA requires a probe of all the subsets formed, one by one, requiring as many TCAM accesses as there are subsets. Our algorithm needs only p memory accesses, just as many as the number of matching rules per packet.

3 Naive Crossproducting Algorithm

The naive crossproducting algorithm works as follows. Let a five-tuple rule be specified as $r = [v_1, v_2, v_3, v_4, v_5]$ where each v_i is a prefix of field i. Let $V_i = \cup v_i$ i.e. V_i is a set of all the distinct prefixes of the field i present in the rule set. The crossproducting algorithm creates all the possible rules of the form $r' = [v'_1, v'_2, v'_3, v'_4, v'_5]$ where $v'_i \in V_i$. In other words, the algorithm simply produces the crossproduct set $V_1 \times V_2 \times V_3 \times V_4 \times V_5$. Given a five-tuple of the packet header, a matching rule can be searched as follows. First we perform an independent search on each field and find the most specific prefix i.e. the longest matching prefix . After having obtained these longest matching prefixes for each field u_i, we create a unique key $u = [u_1, u_2, u_3, u_4, u_5]$ and use it to directly index the crossproduct rule table. Each rule in the crossproduct table is either the original rule or an artificial rule generated in the process of crossproducting. Moreover, each extra rule either corresponds to an original rule or does not correspond to anything. Hence, upon a match, we either get the ID of an original rule or we don't get any ID implying there was no match. Thus, when there is a match, the correct matching rule can always be found. This can be illustrated with the example shown in Figure 1. Here, we show only a two dimensional rule set where each field is four bits wide for the purpose of illustration. The original rule set is shown in Figure 1(A). Figure 1(C) shows the crossproduct table for this rule set. Figure 1(B) shows the representation of the rules using a trie.

In this rule set, the first field contains 4 unique prefixes {1*, 00*, 01*, 101*} which can be labeled as $\{1, 2, 3, 4\}$ respectively. Likewise, the second field contains 4 unique prefixes {*, 00*, 11*, 100*}, which can also labeled as $\{1, 2, 3, 4\}$ respectively. A straightforward crossproduct table will contain $4 \times 4 = 16$ entries. Among the these 16 entries are the six original rules (white colored rows) and the remaining are generated due to the crossproduct. There are crossproduct rules that correspond to an original rule, i.e. a match of these crossproduct rule implies a match for one or more of the original rules. We call these rules "pseudo-rules" (blue colored rows). Take for instance the rule $p_7 = [101*, 00*]$. If there is a match for this rule then it implies a match for original rules r_1 and r_2 since p_7 is more specific to both r_1 and r_2. There are also some entries which do not map to any original rule, e.g. [01*, 1*], which we call "empty rules" (green colored rows). To illustrate the rule matching process, assume that we get a packet with header value [1011, 0011]. We perform the longest prefix matching on each of these fields and find that these prefixes are 101 with label 4 and 00 with label 2. The entry at the location $4 \times 2 = 8$ in the table can be look up for a match. Since there is a matching rule p_7, we can declare a match for the original matching rules r_1 and r_2.

This algorithm has two problems: 1) A large number of empty rules 2) A very large number of pseudo-rules. The first problem can be mitigated by using a hash table instead of direct lookup table. The crossproduct table maintains all the possible entries generated from the crossproduct so that it can be directly indexed. Since there are several empty rules, the sparsity can be utilized to compress the table further by using a hash table. This is a trivial modification to the crossproduct table. Henceforth we assume that the crossproduct table contains only the rules that correspond to at least one of the original rules, i.e. we have only pseudo-rules and the original rules but no empty rules. A trie based representation of the pseudo-rules along with the original rules is shown in Figure 1(D). We will use this trie based representation to illustrate our algorithm further. We build a trie for each field. We mark the nodes corresponding to the prefixes involved in the rules. A connection between the marked nodes of each field represents a rule.

With this representation, it is easy to see that after the empty rules are removed from the crossproduct

table, the remaining pseudo-rules are the rules such that if we keep following the *marked* ancestors of the nodes of each field then there is at least one combination of marked ancestors that represents one of the original rules. In fact, now we can create the crossproduct rule set with an alternative and more efficient procedure described below. First, we introduce the following notations.

- Let R denote the set of original rules and C the set of rules after crossproduct.
- Let $u \prec v$ denote that u is a prefix of v. (Note that $v \prec v$ always holds). Since each prefix corresponds to a marked node in the trie, we will use the terms prefix and marked node interchangeably. Hence, $u \prec v$ also denotes that the node u is the marked ancestor of marked node v.
- Let r denote a rule. Let $r.v_i$ denote the prefix of field i in the rule. Let $r.Id$ denote the set of rule IDs associated with this rule.
- Let T_i denote the trie built from the prefixes of field i.

The pseudo-code for the crossproduct algorithm is described below:

BuildCrossproductTable(S)
1. **for each** $r \in R$
2.   **for each** field i
3.   InsertInTrie($r.v_i, T_i$)

4. **for each** $r \in R$
5.   **for each** field i
6. $V_i \leftarrow V_i \cup r.v_i \cup GetAllMarkedDescendants(r.v_i, T_i)$
7. **for each** node $v_1 \in V_1$
8.   **for each** node $v_2 \in V_2$
9.   **for each** node $v_3 \in V_3$
10.   **for each** node $v_k \in V_k$
11. $c.v_1 \leftarrow v_1$, $c.v_2 \leftarrow v_2$,... $c.v_k \leftarrow v_k$
12.   if $c \in C$
13. $c.Id \leftarrow c.Id \cup r.Id$
14.   else
15. $c.Id \leftarrow r.Id$
16. $C \leftarrow C \cup c$

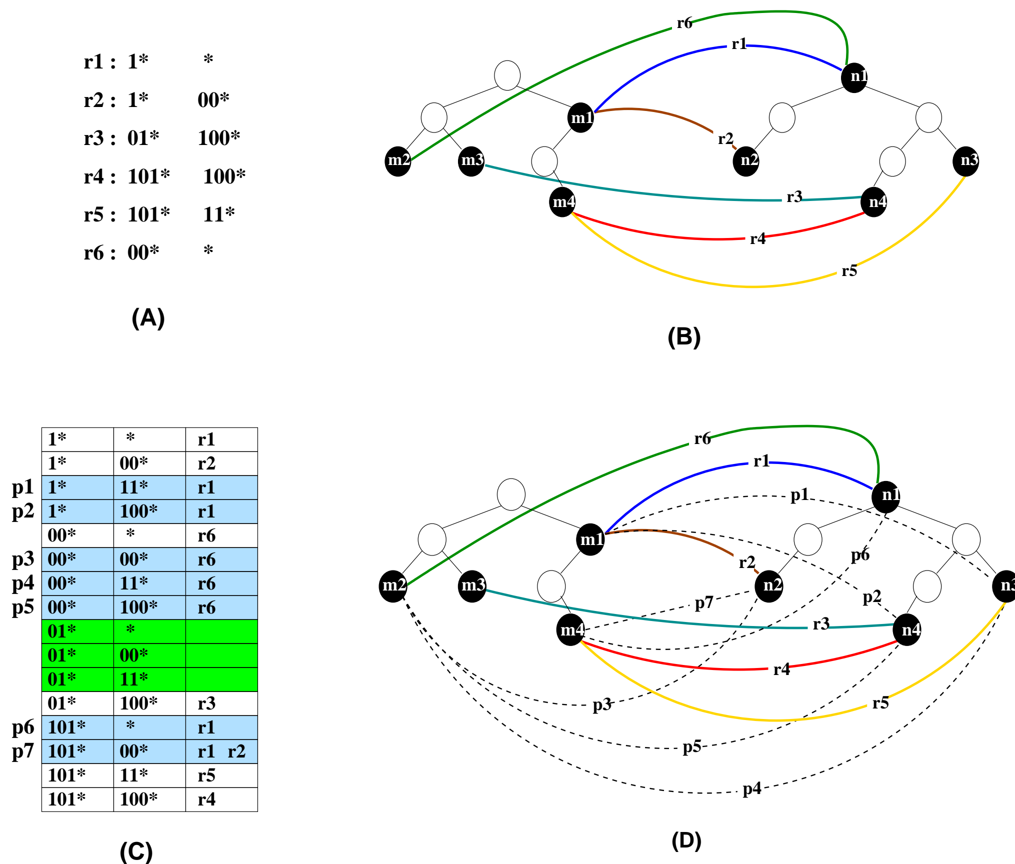
Thus, to build a crossproduct table, we first build a trie for each field with the prefixes of that field in all the rules (line 1-3). Then we pick rules one by one and for each we locate the node corresponding to each field prefix in the corresponding trie and get the set of all the corresponding descendants (line 5-6) including the node under consideration. A set of such descendent's for field i, including the given node itself, is denoted by V_i. Then we take the crossproduct of these sets and insert the resulting rules into the crossproduct set. Note that this crossproduct set will also include the original rule since we are also including the nodes of the original rule in the crossproduct. Each of the crossproduct rules points to the original rule under consideration for which the crossproduct is being generated. While doing so, we see if the rule is already inserted into the table while considering any other original rule. If it is then we just need to append the ID of the original rule under consideration to the set of rule IDs associated with this pseudo-rule (line 12-13). Thus, a match for this pseudo-rule will mean a match for all the rule IDs of the original rules associated with it. If the rule is not present in the table then it is added and the associated rule ID is set to the original rule ID (line 14-16). The resulting rule set C consists of both the original rules and the crossproduct rules.

ClassifyPacket(P)
1. **for each** field i
2. $\quad r.v_i \leftarrow LPM(P.f_i)$
3. $\{match, \{Id\}\} \leftarrow HashTableLookup(r)$

The packet classification process is simple:

As the algorithm describes, we first execute LPM on each field value f_i of packet P and assign the longest matching prefix to a rule r. Then we look up this rule in the hash table. The result of this lookup indicates if the rule matched or not and also outputs a set of matching rule IDs associated with a matching rule.

It is evident that the crossproduct algorithm is efficient in terms of memory accesses: the memory accesses are required for only LPM on each field and the final hash table lookup to search the rule in the crossproduct table. For 5-tuple classification, we don't need to perform the LPM for the port field; it can be a direct lookup in a small on-chip table. Moreover, if we use the Bloom filter based LPM technique described in the previous chapter, we would need approximately one memory access per LPM. Therefore, the entire classification process takes five memory accesses with very high probability to classify a packet.



	1*	*	r1
	1*	00*	r2
p1	1*	11*	r1
p2	1*	100*	r1
	00*	*	r6
p3	00*	00*	r6
p4	00*	11*	r6
p5	00*	100*	r6
	01*	*	
	01*	00*	
	01*	11*	
	01*	100*	r3
p6	101*	*	r1
p7	101*	00*	r1 r2
	101*	11*	r5
	101*	100*	r4

Figure 1: **Illustration of basic ideas. (A) Rule set (B) Rule representation using trie (C) Crossproduct table (D) Representation of original rules and pseudo-rules using trie**

However, the overhead of pseudo-rules can be very large. If each field has 100 unique values in the rule set (ignoring the protocol field) then the expanded rule set can be potentially as large as 100^4 making it impractical to scale for larger rule sets.

In order to get a sense of the amount of expansion the naive crossproducting algorithm can cause, we experimented with several real life rule sets as well as synthetic rule sets which preserve the structure of the real rule sets. We used the synthetic rule set generator ClassBench [14]. The real life rule sets obtained from access control lists (ACL), firewalls (FW) and IP chains (IPC) were used as seeds to generate larger rule sets with approximately ten thousand rules (all the rule sets with name ending in 's' in Table 1). Note that our algorithm needs the ranges to be expanded into prefixes. Due to this expansion, the size of the rule set increases. The reported number of rules in each set is the number after the range to prefix expansion. The number of rules in each set and the expansion factor, δ after the naive crossproduct is shown in Table 1. As the table shows, the expansion factor can be very large. The smallest expansion was observed to be 200 times the original rule set size and the largest was 5.7×10^6 times! Clearly, the naive crossproducting algorithm is impractical for large rule sets.

So how can we reduce the overhead of the pseudo-rules and also preserve the fast speed of the algorithm? We present our Multi-subset Crossproducting Algorithm that achieves this objective.

4 Multi-subset Crossproducting Algorithm

In the naive scheme we require just one hash table access to get the list of matching rules. However, if we allow ourselves to use multiple hash table accesses then we can split the rule set into multiple smaller subsets and take the crossproduct within each of them. With this arrangement, the total number of pseudo-rules can be reduced significantly compared to the naive scheme. This is illustrated in Figure 2. We divide the rule set into three subsets. Within each subset, we take a crossproduct, retaining only the rules that correspond to one of the original rules within that subset. This results in inserting pseudo-rules p_7 in subset 1 (G_1) and p_2 in subset 2 (G_2). All the other pseudo-rules vanish and the overhead is significantly reduced. Why does the number of pseudo-rules reduce drastically? This is because the crossproduct is inherently multiplicative in nature. When the number of overlapping prefixes of a field i get reduced by a factor of x_i due to partitioning, the resulting reduction in the crossproduct rules is of the order Πx_i and hence large.

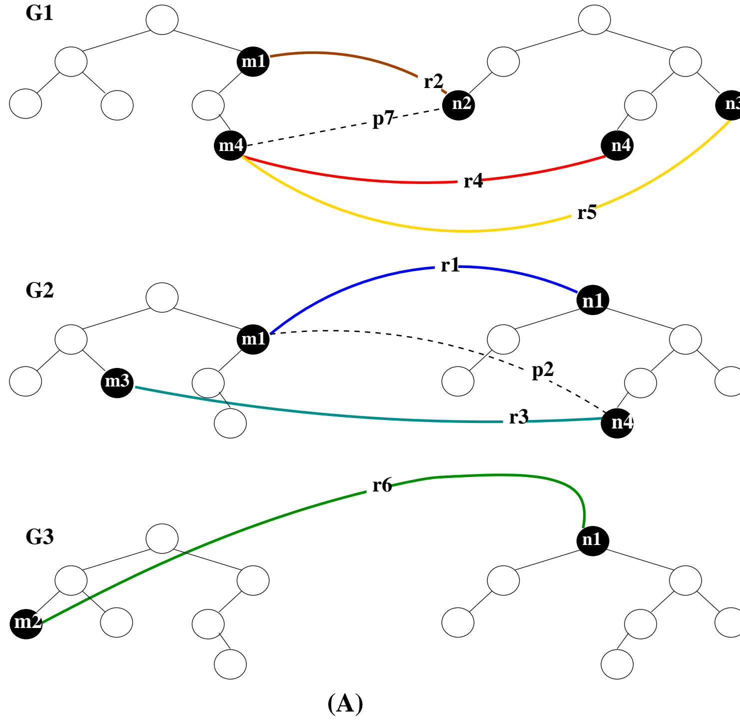
After having reduced the crossproduct memory overhead, an independent hash table can be maintained for each rule subset and an independent rule lookup can be performed in each. The splitting introduces two extra memory access overheads: 1) The entire LPM process on all the fields needs to be repeated for each subset 2) a separate hash table access per subset is needed to lookup the final rule. We now describe how to avoid the first overhead and reduce the second overhead.

With reference to our example in Figure 2, due to the partitioning of rules into subsets G_1, G_2 and G_3, the sets of valid prefixes of the first field are $\{m_1, m_4\}$ for G_1, $\{m_1, m_3\}$ for G_2 and $\{m_2\}$ for G_3. Hence, the longest prefix for one subset might not be the longest prefix in other subset requiring a separate LPM for each subset.

However, this can be easily avoided by modifying the LPM data structure. For each field, we maintain only one global data structure which contains the unique prefixes of that field from all the subsets. When we perform the LPM on a field, the matching prefix is the longest one across all the subsets. Therefore, the longest prefix for individual subsets is either the prefix that matches or its sub-prefix. With each prefix in the LPM table, we can maintain a list of sub-prefixes, one for each subset, which is the longest prefix for that subset.

Conceptually, the LPM table for field i consists of entries where each entry t_i consists of a prefix $t_i.v$ which is the lookup key portion of that entry and the associated information consists of g entries, $t_i.u[1] \ldots t_i.u[g]$ where g is the number of subsets formed. Each $t_i.u[j]$ is either *NULL* or has a value such that $t_i.u[j]$ is the longest matching prefix of field i in subset j which obeys $t_i.u[j] \prec t_i.v$. If $t_i.u[j] == NULL$ then there isn't any prefix of $t_i.v$ which is the longest prefix in that subset.

After a global LPM on the field, we have all the information we need about the matching prefixes in



	G1	G2	G3
1*	**1**	**1**	–
00*	–	–	**2**
01*	–	**2**	–
101*	**3**	**1**	–

LPM Table for field 1

	G1	G2	G3
*****	–	**0**	**0**
00*	**2**	**0**	**0**
11*	**2**	**0**	**0**
100*	**3**	**3**	**0**

LPM Table for field 2

(B)

Figure 2: **Dividing rules in separate subsets to reduce overlap. The corresponding LPM tables.**

individual subsets. Secondly, since $t_i.u[j]$ is a prefix of $t_i.v$, we do not need to maintain the complete prefix $t_i.u[j]$ but just the length of the prefix $t_i.u[j]$. The prefix $t_i.u[j]$ can always be obtained by considering the correct number of bits of $t_i.v$.

The LPM table for the example shown in Figure 2(A) is shown in Figure 2(B). In this example, since we have three subsets, with each prefix we have three entries each corresponding to a subset. For instance, the table for field 1 tells us that if the longest matching prefix on this field in the packet is 101 then there is a sub prefix of `101` of length 3 (which is `101`=m_4 itself) that is the longest prefix in G_1, there is a sub prefix of length 1 (which is `1` =m_1) that is the longest prefix in G_2 and there is no sub prefix (indicated by —) of `101` that is the longest prefix in G_3.

Likewise, the table for field 2 says that if the longest matching prefix for this field in the packet header is `100` then there is a sub prefix of `100` of length 3 (which is `100`=n_4) that is the longest prefix in G_1, there is a sub prefix of length 3 (hence again `100`=n_4) that is the longest prefix in G_2 and finally there is a sub prefix of length 0 (hence $* = n_1$) that is the longest prefix in G_3. Thus, after finding the longest prefix of a field, we can read the list of longest prefixes for all the subsets and use it to probe the hash tables. For

example if `101` is the longest matching prefix for field 1 and `100` for the field 2 then we will probe the G_1 rule hash table with the key $\langle$`101, 100`$\rangle$, the G_2 rule hash table with the key$\langle$`1, 100`$\rangle$ and we don't need to probe the G_3 hash table.

The classification algorithm is described below.

ClassifyPacket(P)
1. **for each** field i
2. $t_i \leftarrow LPM(P.f_i)$
3. **for each** subset j
4. **for each** field i
5. **if**$(t_i.u[j] \neq NULL)$ $r.v_i = t_i.u[j]$
6. **else** break
7. $\{match, \{Id\}\} \leftarrow HashTableLookup_j(r)$

Thus, even after splitting the rule set into multiple subsets, only one LPM is required for each field (line 1-2). Hence we maintain a similar performance as the naive crossproduct algorithm as far as LPM is concerned. After the LPM phase, individual rule subset tables are probed one by one with the keys formed from the longest matching prefixes within that subset (line 3-7). However a probe is not required for a subset if there is no sub-prefix corresponding to at least one field within that subset. In this case, we simply move to the next subset (line 5-6). Hence, the number of rule subset tables probed can be less than the actual number of subsets, depending on the actual prefix values in the rule set. However, for the purpose of analysis, we will stick to a conservative assumption that all the fields have some sub-prefix available for each subset and hence all the g subsets need to be probed.

We will now explain how we can avoid probing all these subsets by using Bloom filters. If a packet can match at the most p rules and if all these rules reside in distinct hash tables then only p of these g hash table probes will be successful and return a matching rule. Other memory accesses are unnecessary, which can be filtered out using on-chip Bloom filters. We maintain one Bloom filter in the on-chip memory corresponding to each off-chip rule subset hash table. We first query the Bloom filters with the keys to be looked up in the subsets. If the filter shows a match, we look up the key in the off-chip hash table. The flow of our algorithm is illustrated in the Figure 3.

From equation [4], the average number of hash table accesses t_i for the LPM on field i, with length W_i is $t_i = 1 + \sum_{j=1}^{W_i-1} f_j$, where f_j is the false positive probability of Bloom filter j. If we tune the Bloom filters to exhibit the same false positive probability, f, by allocating the appropriate amount of memory and the number of hash functions then the average hash table accesses on field i can be expressed as:

$$t_i = 1 + (W_i - 1)f \tag{1}$$

For IPv4, we need to perform LPM on the source and destination IP address (32 bits each) and the source and destination ports (16 bits each). The protocol field can be looked up in a 256 entry direct lookup array kept in the on-chip registers. We don't need memory accesses for protocol field lookup. We can use a set of 32 Bloom filters to store the source and destination IP address prefixes of different lengths. While storing a prefix, we tag it with its type to create a unique key (for instance, source IP type = 1, destination IP type = 2 etc.). While querying a Bloom filter with a prefix, we create the key by combining the prefix with its type. Similarly the same set of Bloom filters can be used to store the source and destination port prefixes as well. The Bloom filters 1 to 16 can be used to store the source port prefixes and 17 to 32 can be used for destination port prefixes. Hence the total number of hash table accesses required for LPM on all of these four fields can be expressed as

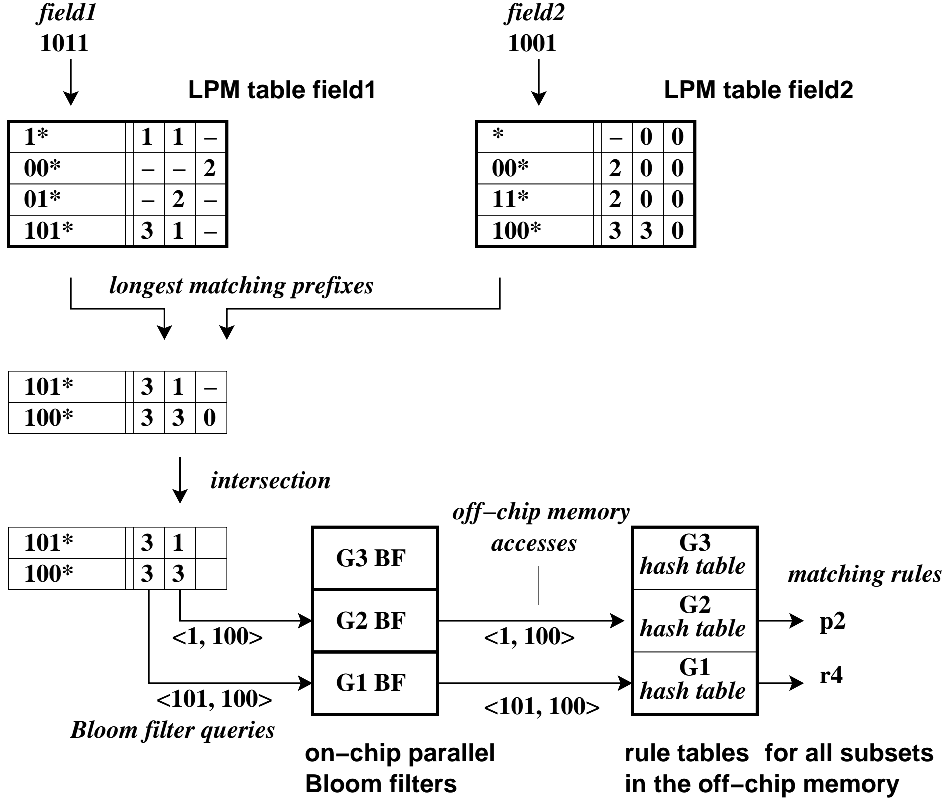


Figure 3: **Illustration of the flow of algorithm. First, LPM is performed on each field. The result is used to form a set of g tuples, each of which indicates how many prefix bits to use for constructing keys corresponding to that subset. The keys are looked up in Bloom filters first. Only the keys matched in Bloom filters are used to query the corresponding rule subset hash table kept in the off-chip memory.**

$$\begin{aligned} T_{lpm} &= (1+31f)+(1+31f)+(1+15f)+(1+15f) \\ &= 4+92f \end{aligned} \tag{2}$$

We need g more Bloom filters for storing the rules of each subset. During the rule lookup phase, when we query the Bloom filters of all the g subsets, we will have up to p true matches and the remaining $g-p$ Bloom filters can show a match, each with false positive probability of f. Hence the hash probes required in the rule matching are

$$T_g = p + (g-p)f \tag{3}$$

The total number of hash table probes required in the entire process of packet classification is

$$T = T_g + T_{lpm} = 4 + p + (92 + g - p)f = 4 + p + \epsilon \tag{4}$$

where $\epsilon = (92+g-p)f$. By keeping the value of f small (e.g. 0.0005), the ϵ can be made negligibly small, giving us the total accesses equal to $\approx 4+p$. It should be noted that so far we have dealt with the number of hash table accesses and not the memory accesses. A carefully constructed hash table requires close to one memory access for a single hash table lookup.

Secondly, our algorithm is a "multi-match" algorithm as opposed to the priority rule match. For our algorithm, priorities associated with all the matching rules need to be explicitly compared to pick the highest priority match.

As the equation 4 shows, the efficiency of the algorithm depends on how small g and f are. In the next section, we explore the trade-off involved in minimizing the values of these two system parameters.

5 Intelligent Grouping

Note that the number of subsets g and the false positive probability f of the Bloom filters are related. If we try to create fewer subsets with a given rule set then it is possible that within each subset there is still a significant number of crossproducts. Hence more rules need to be inserted in the set, which will consume more memory in the Bloom filter in order to maintain the same false positive probability. Hence, decreasing g can increase f if our memory budget is fixed. On the other hand, we do not want a very large number of subsets because it will need a large number of Bloom filters requiring more hardware resources. Hence we would like to limit g to a moderately small value. The key to reducing overhead of pseudo-rules is to divide the rule set into subsets intelligently to minimize the crossproducts. The following questions arise. How can we reduce the number of subsets as well as the pseudo-rules? These appear to be conflicting goals. The pseudo-rules are required only when there are overlapping prefixes of different rules. So, is there an overlap-free decomposition into subsets such that we don't need to insert any pseudo-rules at all? Alternatively, we would also like to know: given a fixed number of subsets, how can we create them with minimum number of pseudo-rules? We address these questions in this section.

5.1 A Problem Formulation

The problem of constructing subsets of overlap-free rules from a given rule set can be modeled as graph coloring problem. We represent the rule set with a graph $G = (V, E)$ in which each vertex in V represents a rule. We add an edge between two vertices if the two rules overlap in at least one dimension, i.e. the two rules create extra crossproduct rules if they are kept in the same subset. Now, we want to color all the vertices with minimum number of colors such that no two vertices connected by an edge have the same color. A color is equivalent to a subset. Graph coloring is known to be an NP-complete problem.

With the graph theoretic problem formulation and heuristic solutions, potentially a tight bound can be found on the number of such subsets. However, we avoid the graph theoretic solutions and seek a simpler heuristic solution that is specific to this problem. Our heuristic of forming subsets is based on the concept of Nested Level Tuple (NLT) explained in the next section. Our solution is simple and provides a loose yet practical upper bound on the number of subsets. Moreover, it requires very little computation. In fact, it turns out to be a highly optimized variant of the Tuple Space Search algorithm. We will discuss the relevance of this in the next section.

Although obtaining subsets of overlap-free rules is our objective, potentially such a partitioning can result in a large number of subsets. Instead, we fix a particular number of subsets and try to partition the rules in them such that the overall pseudo-rules are minimized. How can we create such subsets? We provide an approximate model of this problem by extending the graph model described above. We create a graph $G = (V, E)$ as described above and assign weights to each edge, where the weight equals the number of crossproduct rules due to the overlap of the two rules corresponding to the vertices connected by the edge (i.e. pairwise crossproduct rules). Given this weighted graph, we wish to color the vertices with g colors such that the sum of the weights on the edges connecting vertices of the same color is minimum. Since the rules with the same color go in the same subset, we wish to minimize the sum of pairwise crossproduct rules between all of them, hence the sum of the weights should be minimum. This problem is a standard MIN

K-PARTITION problem which is also NP-complete.

This is an approximate model of the problem because in the context of our problem, the total number of crossproduct rules can be less than the sum of the pairwise crossproduct rules. This is because, some of the crossproduct rules can be common to multiple sets of pairwise crossproduct rules and thus redundant. However, the sum of the pairwise crossproduct rules is an upper bound on the amount of expansion within a subset. Hence, the approximation is quite close to the accurate model.

Again, although a graph theoretic solution is possible for this problem, we avoid this approach and seek a simpler solution by taking advantage of the nature of the problem. We describe a simple heuristic solution in Section 5.3.1 which modifies the NLT based solution for the first problem. Specifically, we use the first heuristic to produce an overlap-free grouping. Given a fixed number of subsets (colors), we pick as many most populated subsets and merge the remaining subsets to them with the objective of reducing the overall crossproduct rules generated by merging.

5.2 Overlap-free Grouping

A loose bound on the number of overlap-free subsets is the number of prefix length tuples. We now describe the Tuple Space Search (TSS) algorithm. While TSS provides one loose bound, we seek a much tighter bound by modifying TSS using a simple technique. We describe our modifications at the end of this section.

5.2.1 Tuple Space Search (TSS)

A Prefix Length Tuple (PLT) is the combination of prefix lengths of different fields. For instance, the PLT [32, 24, 16, 7, 0] implies that the source IP prefix length is 32, the destination IP prefix length is 24, the source port prefix length is 16, the destination port prefix length is 7 and the protocol prefix length is 0 (wild-card). Each rule is contained within a tuple. For IPv4 5-tuple packet classification, the PLT space consists of $33 \times 33 \times 17 \times 17 \times 2 = 629442$ PLTs. In the worst case, each rule can represent a unique tuple and hence the number of PLTs will be the number of rules. For instance, the tuples associated with the rules in our example rule set are [1, 0], [1, 2], [2, 3], [3, 3], [3, 2], and [2, 0] each containing a single rule. However, in reality, the number of PLTs is smaller than the number of rules.

The TSS algorithm maintains all the rules belonging to a PLT in an independent hash table. Upon receiving a packet, it simply looks up all the hash tables by probing them with the keys formed by considering the appropriate number of bits of each field corresponding to that PLT. This naive approach requires several hash lookups. However, they can be significantly reduced by the tuple pruning technique. The TSS algorithm first gets the longest matching prefix of each field. With each longest matching prefix of a given field, a list of PLTs corresponding to the given prefix as well as any shorter prefix is maintained. After reading the list of PLTs associated with the longest matching prefix of each field, only the PLTs in the intersection of these lists need to be looked up. We can illustrate the process with our example rule set. The algorithm is illustrated in Figure 4 which uses our example rule set. As the figure shows, each LPM table contains prefix entries and a list of PLTs that the prefix as well as its sub-prefixes are associated with. For instance, the prefix 1* of the first field is associated with rules $r_1 = [1*, *]$ and $r_2 = [1*, 00*]$ which are contained in the PLTs [1, 0] and [1, 2] respectively. Hence, the LPM entry $1*$ of the first field contains these two PLTs in the list. Likewise, the prefix $101*$ of the first field is associated with PLTs [3, 2] and [3, 3]. Since, $1*$ is a sub-prefix of $101*$, the PLTs [1, 0] and [1, 2] are also contained in the list associated with $101*$.

If the matching prefixes of the two fields were $101*$ and $100*$ then the common PLT list will contain [3, 3] and [1, 0]. Hence, it implies that it is likely that the rules $\{101*,\ 100*\}$ and $\{1*,\ *\}$ are contained in the table. These keys are used to probe the respective hash tables and matching entries are found.

Before we elaborate on the relevance of this algorithm to our algorithm, it is important to mention that the actual TSS algorithm as proposed in [10] does not perform a LPM for source and destination port. Instead, a

different technique based on *Range ID* is used. The authors observed that when the port ranges are converted into prefixes, the resulting expansion could be large. To avoid this expansion, a unique ID is assigned to each range. LPM is performed only on source and destination addresses. The hash key is constructed by taking the appropriate bits from these addresses and combining the range IDs associated with source and destination ports. Range ID can be obtained from the port number using different techniques, including a search tree or a direct lookup. While the search tree based lookup requires more memory accesses, the direct lookup array requires more memory, potentially two arrays containing 64K entries each. Secondly, for assigning a unique ID to a given range, all the ranges must be non-overlapping. If they are overlapping then the overlap must be removed by breaking a single range into multiple smaller ranges which are mutually exclusive. This division of overlapping rangs into smaller non-overlapping ranges essentially means geometric intersection on for each port which results in some rule expansion.

Instead of using the range ID approach, we will use the range to prefix conversion approach for our algorithm. This will allow us to use the Bloom filter based LPM technique for port matching as opposed to the search tree based technique for Range ID matching. Moreover, potentially it will consume less memory compared to the direct lookup array for Range ID matching. Finally, it will not restrict us to using non-overlapping ranges and allow flexible specification of ranges. Cosnidering these factors, we will use the version of TSS algorithm that deals with the prefix representation of port ranges and performs LPM for each field.

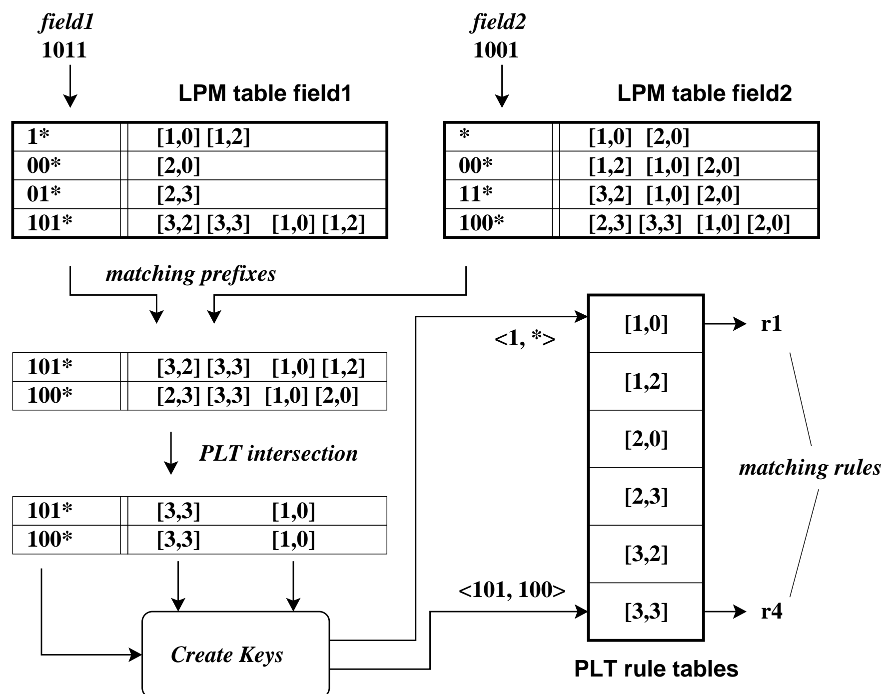


Figure 4: **Illustration of the Tuple Space Search algorithm**

We can now draw a parallel between TSS and our algorithm. Note that the rules contained in the same PLT share the same prefix lengths of each field. Therefore, among any two prefixes of the same prefix length, none is the ancestor of the other. Due to this property, the rules contained within the same PLT do not need crossproducts. Indeed, the number of distinct PLTs in the rule set is essentially one loose upper

bound on the number of overlap-free subsets. In fact, when we use the PLTs as the subsets, our algorithm is the same as TSS except for a few differences in arranging the data structures. With each prefix in the LPM tables, TSS maintains a list of PLTs. Instead, we maintain an array with the number of entries equal to the total number of PLTs, each entry containing the length of the prefix within that PLT. This is illustrated with the Figure 5. For instance, consider the prefix 101* of the first field. There are six entries next to it, each corresponding to a subset (or a PLT). The PLTs are ordered and indexed. As the figure shows, PLT [1,0] is first, [1,2] is second and so on. The first entry among the six is 1 which implies that the given prefix has a sub-prefix which corresponds to a rule contained in the first PLT (which is [1,0]) and the length of this sub-prefix is 1. Likewise, the fifth entry, which is 3 implies that the given prefix has a sub-prefix which corresponds to a rule contained in the fifth PLT (which is [3,2]) and the length of this sub-prefix is 3. When the entry is '-', it means that there is no sub-prefix of the given prefix belonging to any rule in that PLT. In other words, the prefix and its sub-prefixes have nothing to do with that PLT. When we perform LPM on each field and read the array, the intersection becomes easy. We need to consider only those PLTs for which the prefix length in each field is specified. If at least one prefix has '-' for a given PLT then it can be ignored. As the figure shows, after LPM on 1011 and 1001 respectively, the only remaining PLTs are the first and the sixth. The prefix lengths of the individual fields are [1,0] and [3,3]. Now the appropriate number of bits can be considered to construct the keys and the PLT rule sets can be queried.

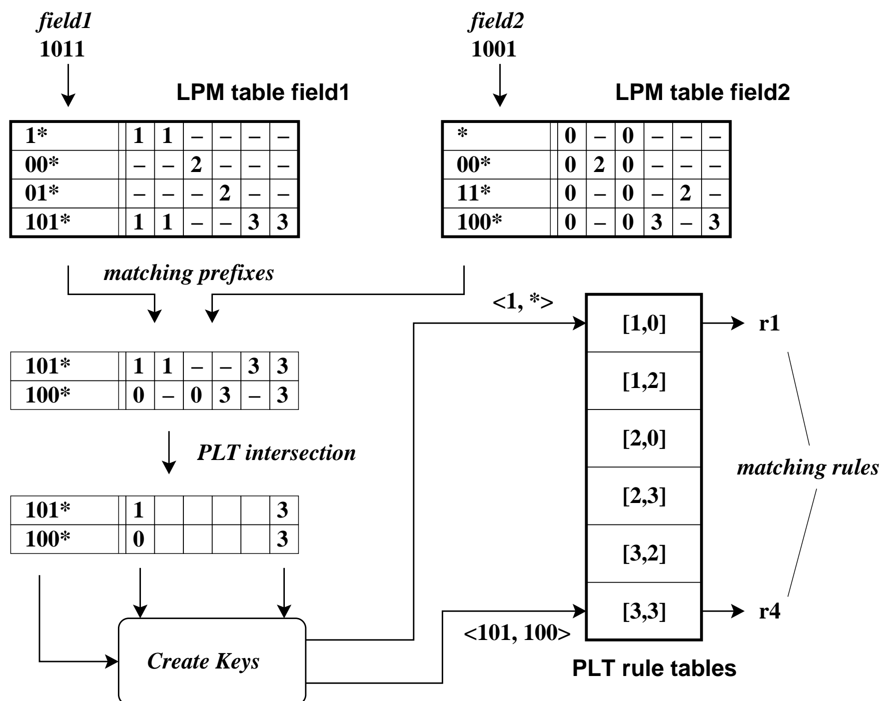


Figure 5: **Illustration of the Tuple Space Search algorithm with an alternative LPM table structure. It equivalent to our algorithm with overlap-free rule subsets.**

Note that there is a bit of redundancy in the LPM data structure which can be used to simplify it further, as proposed in original TSS algorithm. Instead of maintaining the prefix length in each entry, we can simply set a bit to indicate that the given prefix or its sub prefix belongs to that PLT. Thus, the array can be replaced by a bit map with the number of bits equal to the number of PLTs. To take the intersection, we just perform

a bit-wise AND. Finally, for all the remaining PLTs after intersection, we just lookup a table to get the associated prefix lengths for each field and after having obtained those, we can construct the keys as before to probe the appropriate PLT rule tables. This is illustrated in Figure 6. Note that this optimization is possible only because we know that there is a unique prefix length of each field associated with a PLT. In the context of a generic crossproduct, it is not true that a given subset of rules contains prefixes of a specific length for each field; there can be multiple prefixes with different lengths within the same subset. Hence this bit-map data structure can be used only in this special case.

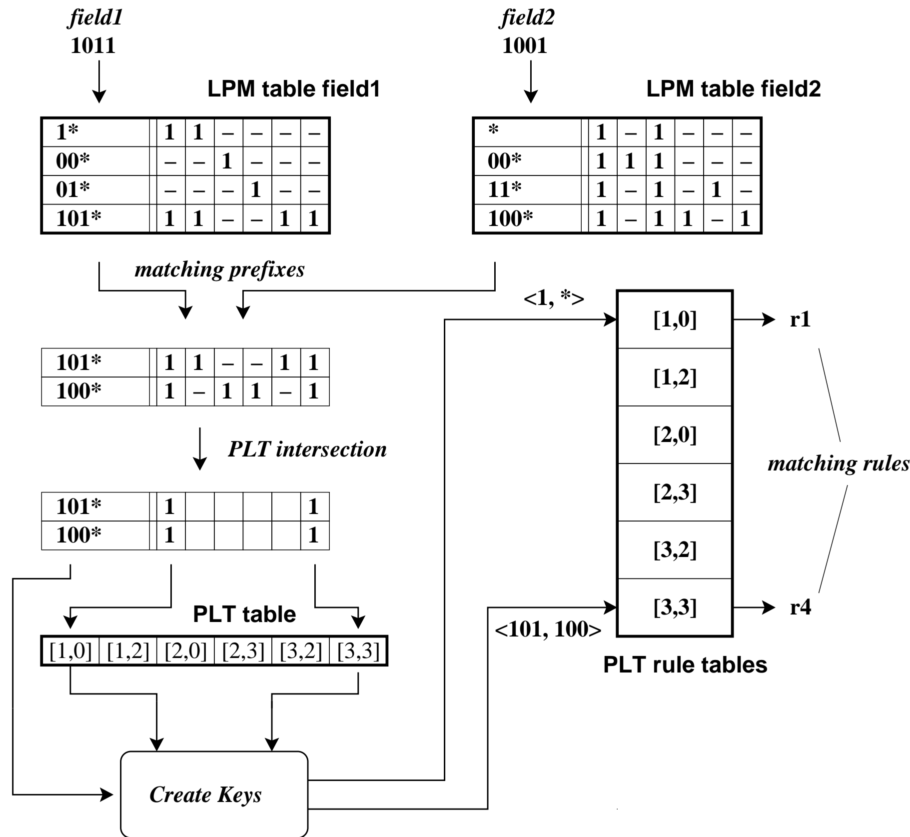


Figure 6: **The LPM table can be compressed further by using a bit map.**

After having discussed the original TSS algorithm, we now discuss the differences between our approach and TSS.

- The original TSS algorithm used conventional trie based techniques for LPM. In our case, we use the Bloom filter based LPM algorithm which is fast.
- The original TSS algorithm probes all the PLTs obtained after pruning whereas we use one more stage of filtering using on-chip Bloom filters. Thus all the PLT queries after pruning can be passed through Bloom filters so that only the potentially successful ones (approximately 'p') will be executed.

By using Bloom filters for memory access filtering, the algorithm performance can be accelerated significantly. However, one important drawback of the system combining Bloom filters and TSS is that the number of PLTs and hence the number of Bloom filters can be very high. We experimented with our rule sets and found that the number of PLTs can be as high as 11,000 for just 25,000 rules as indicated by the Table 1. It is impractical to support such a large number of Bloom filters. However, this problem can possibly

be mitigated by using the same set of *physical* Bloom filters to host a large set of *virtual* Bloom filters. When we store an item in a Bloom filter, we can combine the *type* of the item along with the actual item in order to create a unique key, as discussed before. Thus items of different types can reside in the same physical Bloom filter. When we query the filter with an item, we can combine the type with the key. Therefore, only the item belonging to the correct type will match. This is equivalent to having as many Bloom filters as key types, superimposed on the same physical substrate Bloom filter. Hence we call them virtual Bloom filters. In this fashion, if we have *b* physical Bloom filters to support *B* PLTs, we can design a mapping such that each physical Bloom filter will get to host $\leq \lceil B/b \rceil$ PLTs. We can time multiplex the probing of all the PLT Bloom filters by probing b of them at a time and thus covering all the B probes in $\lceil B/b \rceil$ iterations (or clock cycles). Moreover, after pruning the PLTs, only a few remain to be checked and hence the actual number of probes can be much less than the worst case of B. In spite of that, the number of PLTs to be checked can still be high and variable.

Secondly, it is impractical to maintain an array with each prefix having 11,000 entries. Neither the bit-map technique is practical for the same reason. Hence, we must use the original TSS technique which maintains a list of PLTs along with each prefixes entry. Unfortunately, this will make the process of taking the intersection of the PLT lists associated with the matching prefixes of all the fields very difficult. The problem can be formulated as follows. We are given t sets of numbers S_1,..,S_t, set i containing n_i numbers. Each number is taken from a large universe U. How can we take the intersection of all the sets S_i in hardware? Note that the intersection would have been very easy if the universe U was small. In that case, we could maintain a bit-map of $|U|$ bits for each set S_i and set the bits indexed by the numbers present in that set. Intersection is just the bit-wise AND.

In the light of the drawbacks mentioned above, we now illustrate a technique to reduce the number of subsets substantially. In other words, we proved a tighter upper bound on the number of overlap-free rule subsets a rule set can be partitioned into. When the number of subsets is substantially reduced, the bit-map technique can be used which in turn makes the intersection process easier. This reduction in the number of subsets is based on the concept of Nested Level Tuple which is explained below.

5.2.2 Nested Level Tuple Space Search (NLTSS) Algorithm

We begin by constructing an independent binary prefix-trie with the prefixes of each field in the given rule set just as shown in Figure 1(B). We will use some formal definitions given below.

Nested Level: *The nested level of a marked node in a binary trie is the number of proper ancestors of this node which are also marked. We treat the root node as if it were marked.* For example, the nested level of node m_2 and m_3 is 1 and the nested level of node m_4 is 2.

Nested Level Tree: *Given a binary trie with marked nodes, we construct a Nested Level tree by removing the unmarked nodes and connecting each marked node to its nearest ancestor.* Figure 7 illustrates a nested level tree for field f_1 in our example rule set.

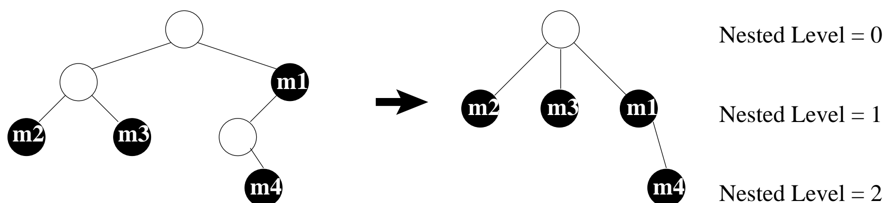


Figure 7: **Illustration of Nested Level Tree**

Nested Level Tuple (NLT): *For each field involved in the rule set, we create a Nested Level Tree (See Figure 8). The Nested Level Tuple (NLT) associated with a rule r is the tuple of nested levels associated with each field prefix of that rule.* For instance, in Figure 8, the NLT for r_6 is [1,0] and for r_4 is [2,1].

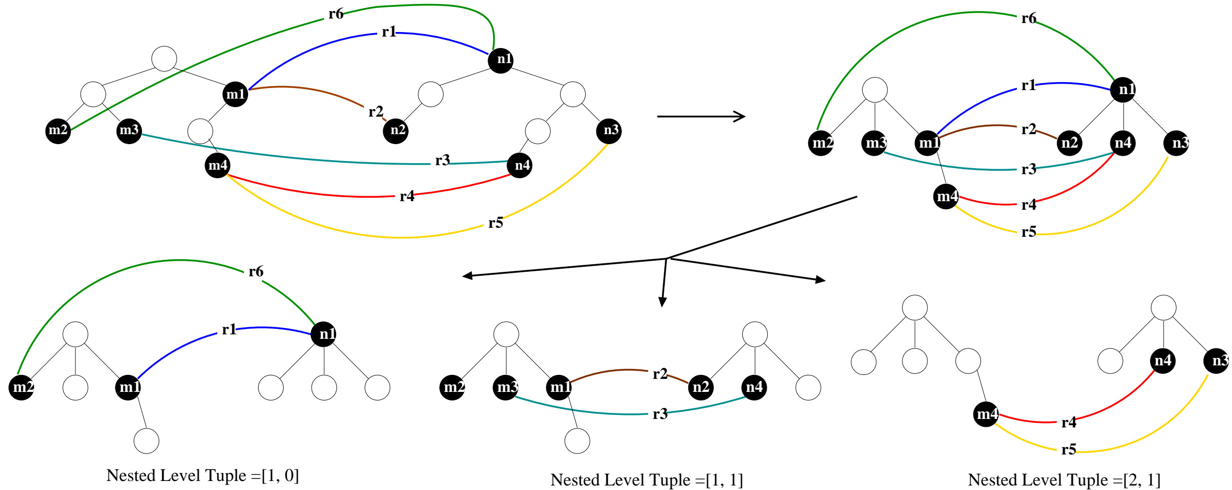


Figure 8: **Overlap free grouping of rules**

From the definition of the nested level, it is clear that among the nodes at the same nested level, no one is the ancestor of the other. Therefore, the prefixes represented by the nodes at the same nested level in a tree do not overlap with each other. **Since there is no overlap between the prefixes contained in the same nested level of the tree, the set of rules contained in the same Nested Level Tuple do not create any crossproduct.** This is illustrated in Figure 8. This gives us one bound on the number of subsets such that each subset contains overlap-free rules.

We experimented with our rule sets to obtain the number of NLTs in each of them. The numbers are presented in the Table 1. While a consistent relationship can not be derived between the number of rules and the number of NLTs from the observations of the rule sets, it is clear that even a large rule set containing several thousand rules can map to less than 200 NLTs. The maximum NLTs were found to be 151 for about 25,000 rules. Given that there are very few NLTs compared to the PLTs, it becomes feasible to use the bit-map to indicate the subsets a prefix belongs to. Therefore, it also becomes feasible to take an intersection of the bit-maps associated with the longest matching prefix of each field for pruning the rule subsets to lookup.

However, given an NLT, we just know the nested level associated with each prefix. We don't know the exact prefix length to use to form our query key for that NLT rule set. Therefore, we need to maintain another bit map with each prefix which gives a prefix length to nested level mapping. We call this bit-map a PL/NL bit-map. For instance, for an IP address prefix, we would maintain PL/NL bit-map of 32 bits in which a bit set at a position indicates that the prefix of the corresponding length is present in the rule set. Given a particular bit that is set in the PL/NL bit-map, we can calculate the nested level of the corresponding prefix just by summing up all the number of bits set before the given bit. Let's illustrate this with an example. Consider an 8 bit IP address and the PL/NL bit-map associated with it as follows:

IP address : 10110110
PL/NL bit-map : 10010101

Thus, the prefixes of this IP address available in the rule set are: 1* (nested level 1), 1011* (nested level 2), 101101* (nested level 3) and 10110110 (nested level 4). To get the nested level of the prefix 101101* we just need to sum up all he bits set in the bit map up to the bit corresponding to this prefix. If we are

interested in knowing the prefix length at a particular nested level then we can keep adding the bits in the PL/NL bit-map until it matches the specified nested level and return the bit position of the set bit as the prefix length. Thus, we can construct the PLT from a NLT using the PL/NL bit-maps associated with the involved prefixes. The PLT tell us which bits to use to construct the key while probing the associated rule set (or Bloom filter). The modified data structures and the flow of the algorithm is shown in Figure 9. As the figure shows, each prefix entry in the LPM tables has a PL/NL bit-map and a NLT bit-map. For instance, prefix $101*$ of field 1 has a PL/NL bit map of 1010 which indicates that the sub-prefixes associated with the prefix are of length 1 (i.e. prefix $1*$) and 3 (i.e. prefix $101*$ itself). Therefore, the nested level associated with the prefix $1*$ is 1 and with $101*$ is 2. Another bit-map, NLT bit-map, contains as many bits as the number of NLTs. The bits corresponding to the NLTs to which the prefix and sub-prefixes belong are set. Thus $101*$ belongs to all the three NLTs whereas $1*$ belongs to NLT 1 and 2. After the longest matching prefixes are read out, the associated NLT bit-maps are intersected to find the common set of NLTs that all the prefixes belong to. As the figure shows, since the prefixes belong to all the NLTs, the intersection contains all the NLTs. From this intersection bit-map we obtain the indices of the NLTs to check. From the NLT table, we obtain the actual NLTs. Combining the knowledge from the PL/NL bit maps of each field, we convert the nested level to the prefix length and obtain the list of PLTs. This list tells us how many bits to consider to form the probe key. The probe is first filtered through the on-chip Bloom filters and only the successful ones are used to query the off-chip rule tables. As the example shows, the key $\langle 1, 100 \rangle$ gets filtered out and doesn't need the off-chip memory access.

Note that the bit-map technique can be used instead of the prefix length array only because there is a unique nested level or prefix length associated with a subset for a particular field. For a generic multi-subset crossproduct, we can use the bit-map technique since there can be multiple sub-prefixes of the same prefix associated with the same sub-set. Therefore, we need to list the individual prefix lengths, just as shown in Figure 5 or 3.

5.3 Limiting the Number of Subsets

While the NLT based grouping works fine in practice, we might ask, is there still room for improvement? Can the number of subsets be reduced further? This brings us back to our second question: how can we limit the number of subsets to a desired value? While the NLT technique gives us crossproduct-free subsets of rules, we can still improve upon the it by merging some of the NLTs and applying the crossproduct technique to them in order to limit the number of subsets. Fewer subsets also means fewer Bloom filters and hence a more resource efficient architecture. In the next subsection, we describe our NLT merging technique and the results after applying the crossproduct algorithm.

5.3.1 NLT Merging and Crossproduct (NLTMC) Algorithm

In order to reduce the subsets to a given threshold, we need to find the NLTs that can be merged. We exploit an observation that holds across all the rule sets we analyzed: the distribution of rules across NLTs is highly skewed. Most of the rules are contained within just a few NLTs. Figure 10 shows the plot of the cumulative distribution of rules across the number of NLTs.

This indicates that we can take care of a large fraction of rules with just a few subsets. Hence what we need is an NLT merging algorithm whereby we start with the overlap-free NLT set, retain the most dense NLTs equal to the specified subset limit and then merge the rules in the remaining NLTs to these fixed subsets with the objective of minimizing the pseudo-rule overhead. It is possible to devise clever heuristics to meet this objective. Here, we provide a simple heuristic that proved very effective in our experiments. Our NLT merging algorithm works as follows.

- Sort the NLTs according to the number of rules in them.

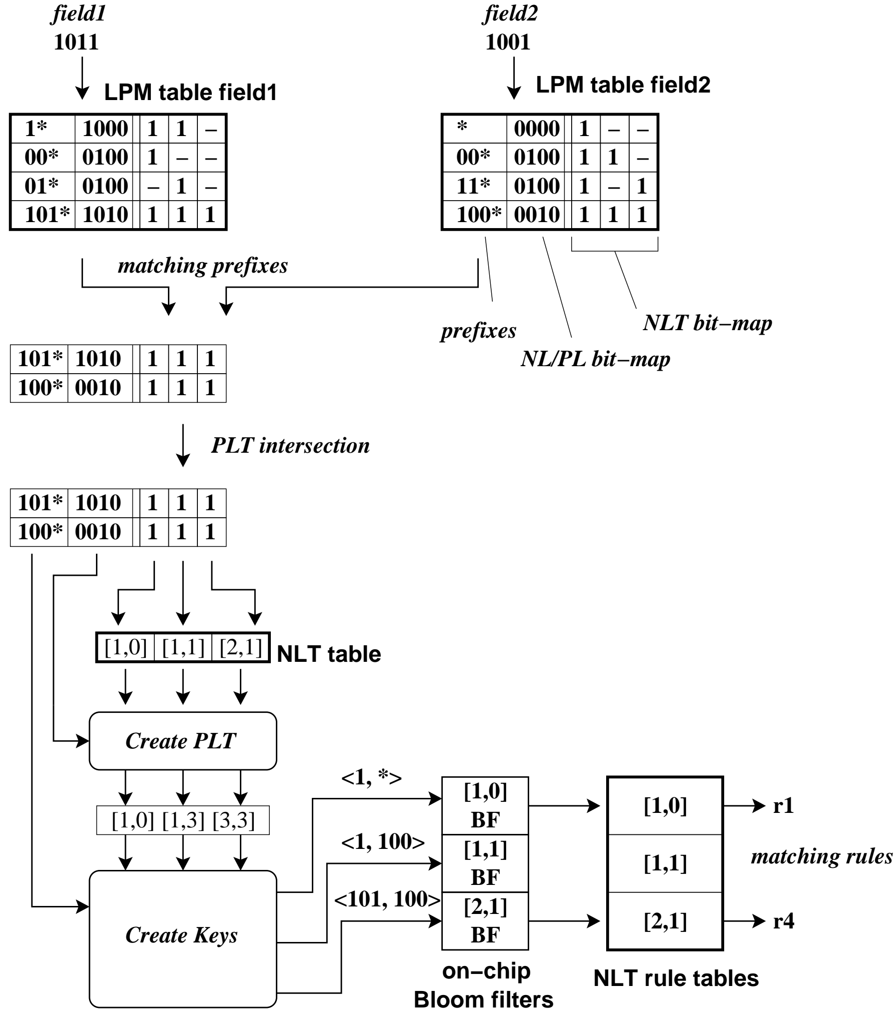


Figure 9: **Using NLT based grouping to form the subsets. Each prefix entry in LPM table needs a NL/PL bit-map and another bit-map indicating the NLTs to which the prefix or its sub-prefixes belong.**

- Pick the most dense g NLTs where g is the given limit on the number of subsets. Merge the remaining NLTs to these g NLTs.

- While any of the remaining NLTs can be merged with any one among the fixed g NLTs, a blind merging will not be effective. To optimize the merging process, we choose the most appropriate NLT to merge with as follows. Take the "distance" between the NLT i and each of the fixed g NLTs. We merge the NLT i with an NLT having minimum distance. In case of a tie, choose the NLT with minimum rules to merge with. We define the distance between the two NLTs to be the sum of differences between individual field nested levels. For instance, the NLT [4, 3, 1, 2, 1] and [4, 1, 0, 2, 1] have a distance of $|3-1|+|1-0|=3$. The intuition behind the concept of distance is that when the distance between the NLTs is large, it is likely that one NLT will have several descendant nodes corresponding to the nodes in another NLTs thereby potentially creating a large crossproduct. Shorter distance will potentially generate fewer crossproducts.

- Although, merging helps us reduce the number of NLTs, it can still result in a large number of crossproducts. At this point, while merging a NLT with another, we try to insert a rule and see

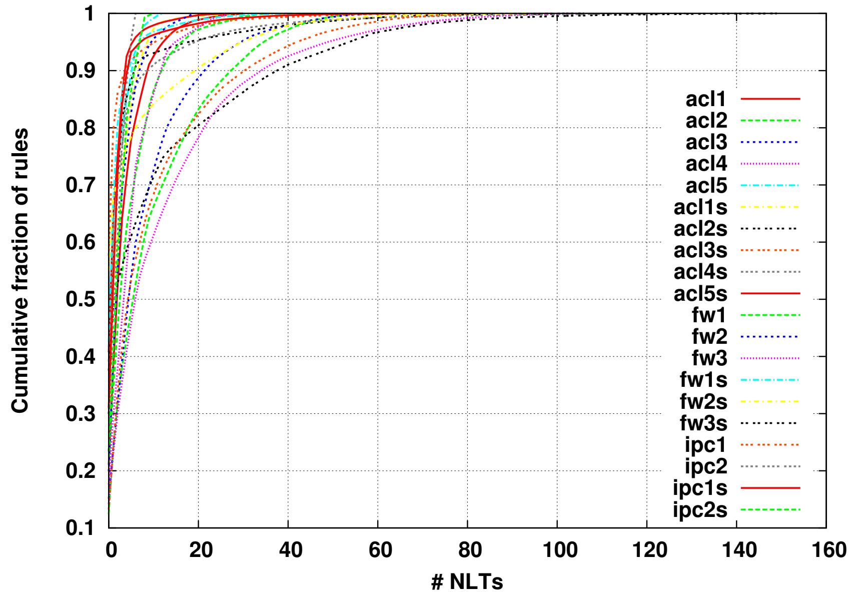


Figure 10: **Cumulative distribution of the rules across NLTs. More than 90% rules are contained within just 40 NLTs.**

how many pseudo-rules it generates. If the number exceeds a threshold then we don't insert it. We consider it to be a "spoiler". We denote by t this threshold on pseudo-rules to consider a rule spoiler. All the spoilers can be taken care of by some other efficient technique such as a tiny on-chip TCAM. We emphasize that such an architecture will be significantly cheaper and power efficient compared to using a TCAM for all the rules. As we will see, our experiments show that the spoilers are typically less than 1% to 2% and hence the required TCAM is not a significant overhead.

In summary, given a fixed number of subsets, we begin by formation of NLTs. If the the NLTs are greater than the subset limit, we pick the most dense NLTs equal to the number of subsets and merge the remaining NLTs to these fixed NLTs. While merging, we isolate the spoilers. This proves to be an effective technique to meet the objective of containing the tuples as well as reducing the spoilers, as indicated by the results presented in Table 1. We denote by α the ratio of the size of the new rule set after executing our algorithm, to the size of the original rule set (after range to prefix expansion). We experimented with different values of g, i.e. the desired limit on NLTs. The pseudo-rule threshold was arbitrarily fixed to $t = 20$.

From the results it is clear that even with the number of subsets as small as 16, the rule set can be partitioned without much expansion overhead. The average expansion factor for $g = 16$ is just 1.43. Among the 20 rule sets considered above, the maximum expansion was observed to be almost four times (acl3s) for 16 subsets. For all the other rule sets, the expansion is less than two times. Furthermore, it can also be observed that as we increase the number of subsets, the expansion decreases as expected. However this trend has an exception for fw3s where both $g = 24$ and $g = 32$ show larger expansion compared to $g = 16$. This is because the $g = 16$ configuration throws out more spoilers compared to $g = 24$ and $g = 32$. Thus, our algorithm in this particular case trades off more spoilers for less expansion. Overall, it can also be observed that the spoilers are very few, on an average $\beta < 2\%$. As we increase the number of subsets, the spoilers are reduced significantly. Clearly, $g = 32$ is the most attractive choice for the number of subsets due to the small number of spoilers and the small expansion factor.

rule set	rules	δ	PLT	NLT	prefixes	g=16		g=24		g=32	
						α	β	α	β	α	β
acl1	1247	2.4e+4	79	31	610	1.03	0.00	1.03	0.00	1.00	0.00
acl2	1216	7.6e+3	195	57	437	1.93	4.19	1.24	1.40	1.17	0.00
acl3	4405	2.3e+5	367	63	1211	1.29	4.45	1.16	0.75	1.14	0.25
acl4	5358	4.3e+5	397	107	1445	1.74	7.95	1.52	2.24	1.20	0.62
acl5	4668	7.0e+2	69	14	304	1.00	0.00	1.00	0.00	1.00	0.00
acl1s	12507	3.2e+4	1349	45	1524	1.03	0.28	1.00	0.10	1.00	0.00
acl2s	18589	1.0e+3	6131	107	626	1.12	2.32	1.14	0.56	1.14	0.39
acl3s	17395	2.5e+4	4136	81	947	3.99	0.71	2.27	0.54	2.26	0.21
acl4s	16291	4.4e+4	4003	130	1090	1.46	2.22	1.45	0.53	1.42	0.42
acl5s	13545	2.3e+4	1197	31	2401	1.03	0.00	1.00	0.00	1.00	0.00
fw1	914	3.0e+5	221	37	205	1.37	0.11	1.10	0.11	1.03	0.00
fw2	543	7.4e+3	159	21	132	1.06	0.00	1.00	0.00	1.00	0.00
fw3	409	1.6e+4	169	29	147	1.25	0.00	1.03	0.00	1.00	0.00
fw1s	32135	5.7e+6	237	50	337	1.92	0.80	1.15	0.012	1.09	0.006
fw2s	26234	1.5e+3	11016	95	271	1.60	2.81	1.46	1.47	1.46	0.42
fw3s	24990	6.7e+3	11296	151	460	1.53	6.45	2.05	1.45	1.80	0.94
ipc1	2179	1.9e+5	244	83	396	1.73	5.69	2.10	1.19	1.41	0.73
ipc2	134	3.1e+2	8	8	72	1.00	0.00	1.00	0.00	1.00	0.00
ipc1s	12725	6.0e+4	3433	65	519	1.86	1.09	1.12	0.26	1.03	0.09
ipc2s	9529	1.7e+4	782	11	4596	1.00	0.00	1.00	0.00	1.00	0.00
avg						1.43	1.95	1.28	0.70	1.20	0.34

Table 1: **Results with different rule sets. δ denotes the expansion factor on the original rule set after naive crossproduct.α denotes the expansion factor on the original rule set after Multi-subset Crossproduct. β denotes the** ***percentage*** **of the original rules which are treated as spoilers.**

6 Architecture

In this section we describe the architecture of the entire system and discuss some of the engineering considerations in a hardware implementation of our algorithm.

6.1 Hash Table Architecture

An important issue in any hash table based algorithm is of reducing hash collisions. To reduce the collisions in the hash table, Song et. al. propose a Fast Hash Table (FHT) architecture [9]. We borrow the example from [9] and show how FHT functions using Figure 11(A). As the figures shows, four items x, y, z and w are being inserted in the hash table and the counters of the buckets to which they hash are incremented. After the hash table is pruned by removing the unnecessary copies of the items, it looks as shown in Figure 11(B). This hash table significantly reduces the collisions which makes it suitable for our purpose. In fact, all we need to do is to convert our ordinary Bloom filter into a counting Bloom filter and associate a hash bucket with it. Each hash bucket keeps the pointer to the list of items hashed to it. As will be explained in the next subsection, we use the ratio of 16 hash buckets per item. With this ratio, using the results from [9], it can be shown that among 128K items, there are only less than 75 items that collide. This is an acceptably small number of collisions and the colliding items can be kept in the on-chip memory. Therefore, it is reasonable to assume that the with FHT, we need only one memory access to read an item from the hash table.

We modify FHT to further reduce the memory consumption by compressing the pointer array.[1] Note that the bucket associated with a non-empty item list is sparse in an FHT. This sparsity can be exploited to compress the pointer array. Figure 11 explains how exactly we compress the pointer array. Let L be the number of items stored in a m-bucket array where $L < m$. We divide the array into smaller segments of s buckets. For each bucket we maintain a bit indicating if it is occupied or not. Then we keep a pointer to the first item falling in that segment. The first item of all the other lists in that segment are kept in successive memory locations in the item memory. Each of these items can be accessed with reference to the pointer to the first item. When an item in a bucket is to be accessed, we check to see if the bucket is occupied or not. If it is, then we count the number of bits set to 1 within that segment up to the given bucket and add this offset to the base pointer to get the required item. For instance, consider the bucket number 4 in the figure which contains the item z. To access this item, we first see if the bit corresponding to the bucket is set. Then we count the number of bits set to 1 before the given bit within that segment. There is just one bit set before the bit corresponding to z. Hence we add the offset 1 to the base pointer associated with that segment and access the required item. Here, the base pointer points to x and z is arranged right next to it. Hence we retrieve z.

With this technique, we need just a s-bit vector and a pointer to the first item within that segment as opposed to s pointers. If the length of a pointer is t bits, it results in a space reduction from st bits to $s + t$ bits. The bit-vector and the base pointer can be arranged compactly in the SRAM as shown in the figure (D). This compression technique using bit-vector is not new. It has been used in various data structures previously including the encoding of the multibit-trie [5]. However, its application in the context of hash table compression is new. Any hash table can be compressed with this bit-vector technique. In the next subsection, we will evaluate the amount of memory required for the counting Bloom filters and the pointer array to achieve a desired performance.

6.2 Memory Requirement

There are three data structures in our algorithm that consume memory. First is the counting Bloom filter, second is the pointer array, and the third is the actual item memory.

[1]This compression scheme was jointly developed by Haoyu Song and Sarang Dharmapurikar.

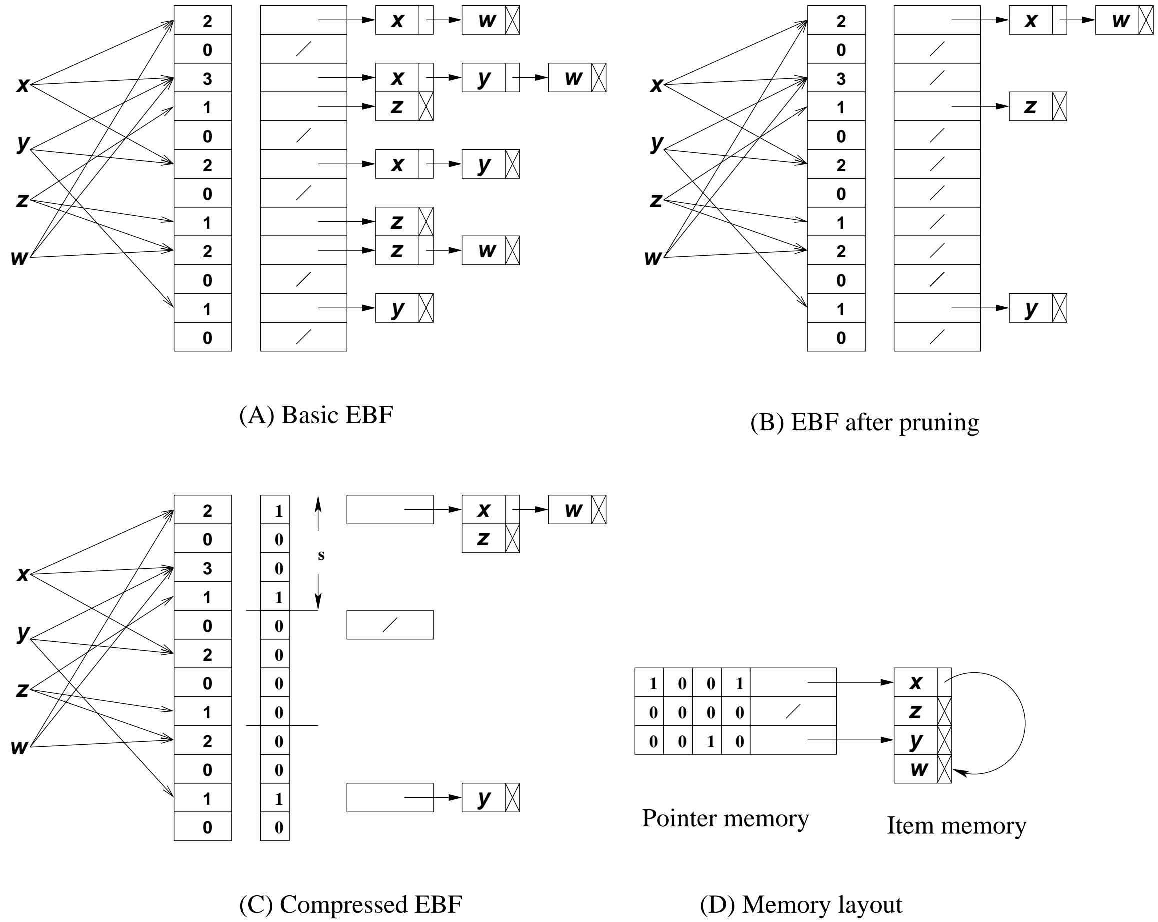


Figure 11: **Illustration of Fast Hash Table and its compression. The example is borrowed from [9]. (A) The basic FHT (B) FHT after pruning (C) Compressing pointer array (D) Arranging pointer array compactly in memory**

Item Memory: The item memory consists of two types of items: prefixes for all the four fields and rules. The prefix entries in the LPM table depend on the algorithm we choose. When we implement the NLTSS algorithm explained in Section 5.2, the optimized prefix entry shown in Figure 9 contains the 32-bit IP prefix, 32 bits for the PL/NL bit-map, and g bits for the NLT bitmap for as many NLTs. For a port prefix, we need 16 bits of prefix and 16 bits of PL/NL bitmap, hence 32 bits less. However, for the sake of uniformity, we will use the same amount of space for port prefixes as used for IP address prefixes. Hence a prefix entry needs 64+g+2 bits, the last 2 bits being used for specifying the particular field out of source/destination IP and source/destination port. We round it up to the nearest multiple of 36 since SRAM memory is available with this word size. Thus, a single prefix entry requires b_{NLTSS} bits given as follows.

$$b_{NLTSS} = \lceil (66 + g)/36 \rceil \times 36 \tag{5}$$

For the NLTMC$_g$ algorithm, the LPM data structure is as shown in Figure 2(B), with each prefix, we maintain a word which contains the prefix length information of all the g subsets. Each entry in this array takes a value between 0 to W or NULL where W is the maximum length of the prefix. Therefore, there are $W + 2$ possible values requiring $\lceil log_2(W + 2) \rceil$ bits per entry, which is 6 bits for the IP addresses and 5 bits for the ports. For an IP prefix, we would need 33 bits to specify a prefix of arbitrary length, $g \times 6$ bits to maintain the sub-prefix information for the g subsets, and finally 2 more bits to indicate the field that the

prefix belongs to. Totally, we need $6g + 34$ bits to store prefix item for this algorithm. Rounding it up to the nearest multiple of 36 gives us b_{NLTMC_g} bits per entry given as follows.

$$b_{NLTMC_g} = \lceil (6g + 34)/36 \rceil \times 36 \quad (6)$$

The actual rule can be specified by using 33 bits for each source and destination IP, 17 bits for each source and destination port, and 9 bits for protocol. If we use 17 bits for the next pointer, a rule item requires 126 bits totally. Again, we round it up to 144.

To compute the average number of *item bytes* per original rule, incorporating all of the above parameters, we use the following formula:

$$M_{of} = \frac{\#rules \times \alpha_g \times 144 + \#prefixes \times b}{\#rules \times 8} \quad (7)$$

where, b is b_{NLTSS} or b_{NLTMC_g} depending on the algorithm. and α_g is 1 for NLTSS and as specified in Table 1 for the NLTMC with $g =$ 16, 24 and 32 subsets.

Bloom filters and pointer array: Now we compute the memory required for Bloom filters and pointer array.

We use $k = 12$ hash functions and set buckets per item to 16 (i.e. $m/n = 16$) which gives a false positive probability of 0.00046, low enough for our purpose. Keeping the ratio of m/n fixed, we experiment with different values of m. Since FHT needs a counting Bloom filter, each bucket of the Bloom filter is a counter of 2 bits. Moreover, associated with each bucket of the Bloom filter is a hash table bucket containing the pointer to the actual items. Therefore, we have m pointers for n items. If we restrict the maximum number of items in each Bloom filter to 64K, then we can use a 16 bit pointer. We compress the array as described earlier using $s = 16$ as the segment length. Thus, for every 16 entries of the array, we have a 16-bit vector and 16-bit pointer. Therefore, the memory consumption per bucket due to the pointer array is $((m/16) \times (16 + 16))/m = 2$ bits. The total memory consumption *per bucket* due to the pointer array and the counting Bloom filter together is now $2 + 2 = 4$ bits. Since there are 16 buckets per item ($m/n = 16$), the number of bits per item is $4 \times 16 = 64$. The total number of items in the system is simply the number of rules after expansion (#rules $\times \alpha$) plus the unique prefixes of all the fields. Hence the memory consumption per original rule in *bytes* due to the Bloom filters and the pointer array is

$$M_{on} = \frac{64 \times (\alpha_g \times \#rules + \#prefixes)}{\#rules \times 8} \quad (8)$$

Again, $\alpha_g = 1$ for the NLTSS algorithm. The average bytes required per original rule is therefore just the sum of the two components:

$$M = M_{of} + M_{on} \quad (9)$$

We evaluated this memory requirement for each of our rule sets, and the numbers are shown in Table 2.

As the table shows, the NLTSS algorithm requires fewer bytes compared to all the configurations of the NLTMC algorithm. This is due to two reasons. First, there is rule set expansion due to crossproducts in NLTMC which is absent from the NLTSS algorithm. Second, NLTMC requires a wider word for each prefix entry in the LPM table. As we increase the number of subsets from 16 to 32, some interesting observations can be made about memory requirement for different rule sets. Consider for instance acl1 rule set. With increase in the number of subsets, the LPM entry becomes wider and hence requires more off-chip memory per rule. On the other hand, acl4 shows exactly opposite trend. This is because, with fewer subsets, acl4 shows a higher factor of rule set expansion due to crossproducts. Hence, with fewer subsets, the overall memory required per rule is larger. A combination of both of these factors can be seen in acl2 where the memory requirement is highest for $g = 16$ subsets, lowest for $g = 24$ subsets and between these two values for $g = 32$ configuration. This is because, with 16 subsets, there is too much rule set expansion that dwarfs

rule set	NLTSS					NLTMC														
						$g = 16$					$g = 24$					$g = 32$				
	Memory		Throughput			Memory		Throughput			Memory		Throughput			Memory		Throughput		
	M_{on}	M_{of}	≤ 4	6	8	M_{on}	M_{of}	≤ 4	6	8	M_{on}	M_{of}	≤ 4	6	8	M_{on}	M_{of}	≤ 4	6	8
acl1	12	25	38	25	19	12	28	38	25	19	12	30	25	25	19	12	34	19	19	19
acl2	11	25	38	25	19	18	42	38	25	19	13	31	25	25	19	12	33	19	19	19
acl3	10	23	38	25	19	12	29	38	25	19	11	28	25	25	19	11	30	19	19	19
acl4	10	25	25	25	19	16	37	38	25	19	14	34	25	25	19	12	31	19	19	19
acl5	8	19	38	25	19	8	20	38	25	19	8	20	25	25	19	8	21	19	19	19
acl1s	9	21	38	25	19	9	21	38	25	19	9	21	25	25	19	9	22	19	19	19
acl2s	8	19	25	25	19	9	21	38	25	19	9	22	25	25	19	9	22	19	19	19
acl3s	8	20	25	25	19	33	73	38	25	19	19	43	25	25	19	19	43	19	19	19
acl4s	8	20	25	25	19	12	28	38	25	19	12	28	25	25	19	12	28	19	19	19
acl5s	9	21	38	25	19	9	22	38	25	19	9	22	25	25	19	9	24	19	19	19
fw1	10	22	38	25	19	13	29	38	25	19	10	25	25	25	19	10	26	19	19	19
fw2	10	22	38	25	19	10	24	38	25	19	10	24	25	25	19	10	26	19	19	19
fw3	11	23	38	25	19	13	29	38	25	19	11	27	25	25	19	11	30	19	19	19
fw1s	8	19	38	25	19	15	35	38	25	19	9	21	25	25	19	9	20	19	19	19
fw2s	8	19	25	25	19	13	29	38	25	19	12	27	25	25	19	12	27	19	19	19
fw3s	8	19	19	19	19	12	28	38	25	19	17	38	25	25	19	15	33	19	19	19
ipc1	9	23	25	25	19	15	35	38	25	19	18	42	25	25	19	13	32	19	19	19
ipc2	12	26	38	25	19	12	28	38	25	19	12	31	25	25	19	12	35	19	19	19
ipc1s	8	19	38	25	19	15	35	38	25	19	9	22	25	25	19	8	20	19	19	19
ipc2s	12	25	38	25	19	12	27	38	25	19	12	29	25	25	19	12	34	19	19	19
avg	10	22	34	25	19	14	31	38	25	19	12	29	25	25	19	12	29	19	19	19

Table 2: **The performance of different algorithms with different parameters.** M_{on} **and** M_{of} **denote the average on-chip and off-chip memory in bytes per rule. The throughput is in Million Packets per second. Throughput was computed for different number of matching rules per packets,** $p \leq 4$**,** $p = 6$**,** $p = 8$**. When** $p \leq 4$**, LPM is the bottleneck and throughput is decided by how wide the LPM entry is.**

the effect of shorter LPM entry, thereby requiring a larger amount of memory per rule. With 24 subsets, the expansion gets reduced and its effect dominates the increase in the LPM entry size. With 32 subsets, the LPM entry becomes wider and hence results in more memory per rule while the effect of reduced expansion is not much. Thus, different NLTMC configurations are suitable for different rule sets. Although, it is clear that, NLTSS always beats all the configurations of NLTMC in terms of memory efficiency due to the reasons mentioned above.

On the other hand, NLTMC requires fewer and fixed number of subsets of rules whereas the NLTSS requires many more, potentially up to 151 as the results of Table 1 indicate. Fewer subsets also implies a fewer Bloom filters and hence a more resource efficient architecture. Thus, potentially we can save a significant amount of logic gates resources required to implement Bloom filters if we choose NLTMC, but at the cost of more memory.

6.3 Classification Throughput

The speed of the classification depends on multiple parameters, including the implementation choice (pipelined/non-pipelined), the number of memory chips used for off-chip tables, the memory technology used, and the number of matching rules per packet (i.e. the value of p). We will make the following assumptions.

Memory technology: We will assume the availability of a 300 MHz DDR SRAM chips with 36-bit wide data bus which are available commercially. Such SRAM can allow reading two 36-bit words in each clock cycle of a 300 MHz clock. The smallest burst length is two words (72 bits).

Pipelining: We will use a pipelined implementation of the algorithm. The first stage of pipeline executes the LPM on all the fields and the second stage executes the rule lookup. In order to pipeline them, we will need two separate memory chips, the first containing the LPM tables and the second containing rules. Here, we will also need two separate sets of Bloom filters, the first for LPM and the second for rule lookup. Let τ_{lpm} denote the time to perform a single LPM lookup in the off-chip memory in terms of the number of clock cycles of the system clock. Likewise, let τ_{rule} be the time required for a single rule lookup. If a packet matches p rules in a rule set then, with a pipelined implementation, a packet can be classified in time $max\{4\tau_{lpm}, p\tau_{rule}\}$. Typically, p is ≤ 6 as noted in [8] [6]. We will evaluate the throughput for different values of p.

Choice of algorithm: As before, we have a choice between NLTSS and NLTMC. It should be recalled that depending on the algorithm and the configuration used, the width of an LPM entry can be different. Therefore, LPM lookup time (τ_{lpm}) is different for these two algorithms and different configurations of NLTMC. Secondly, for the NLTSS, the LPM entry width differs with the rule set under consideration whereas it is constant with a specific configuration for NLTMC. We evaluate the throughput for each rule set. Finally, we always need to read the data in the bursts of 72 bits (2 words, 36 bits each) due to which we might need to read more words than we actually need. This too will affect the throughput. Let, $\tau_{(lpm,NLTSS)}$ and $\tau_{(lpm,NLTMC_g)}$ denote the time in clock ticks (of 300MHz clock) to read a LPM entry for NLTSS and NLTMC$_g$ respectively. These can be expressed as follows.

$$\tau_{(lpm,NLTSS)} = \lceil b_{NLTSS}/72 \rceil \tag{10}$$

and

$$\tau_{(lpm,NLTMC_g)} = \lceil b_{NLTMC_g}/72 \rceil \tag{11}$$

Recall that each rule can fit in 144 bits and needs exactly two clock cycles to read. Hence $\tau_{rule} = 2$. The throughput can be given as

$$R_{NLTSS} = \frac{300 \times 10^6}{max\{4\tau_{(lpm,NLTSS)}, 2p\}} \quad packets/second \tag{12}$$

and

$$R_{NLTMC_g} = \frac{300 \times 10^6}{max\{4\tau_{(lpm,NLTMC_g)}, 2p\}} \quad packets/second \tag{13}$$

The throughput is shown in the Table 2. Let's consider the case of NLTSS. When $p \leq 4$, the $max\{4\tau_{(lpm,NLTSS)}, 2p\} = 4\tau_{(lpm,NLTSS)}$ and hence, the LPM phase becomes the bottleneck in the pipeline hence throughput depends on how wide the LPM entry is. It can be seen that a throughput of 38 million packets per second (Mpps) can be achieved for some rule sets having fewer NLTs and hence shorter LPM entry. When the matching rules per packet increases, rule matching phase becomes the bottleneck and limits the throughput. With $p = 6$, the throughput is 25 Mpps and with $p = 8$, it is 19 Mpps. In some cases, such as fw3s, the LPM entry is so wide that the LPM phase continues to be the bottleneck and limits the throughput to 19 Mpps even if the matching rules per packet is 8.

Now, let's consider the NLTMC$_g$ algorithm. As mentioned before, for each value of g the LPM entry has a fixed width across all the rule sets. Therefore throughput is constant for all the rule sets. As can be seen from the table, just like

For $g = 16$ and $p \leq 4$, LPM is bottleneck but since the LPM word is short due to smaller number of subsets, the throughput can be as high as 38 Mpps. As p increases, throughput decreases since rule matching becomes the bottleneck. Likewise, for $g = 24$, LPM is the bottleneck up to $p = 6$ and throughput is limited to 25 Mpps. With $p = 8$, rule matching is the bottleneck and throughput reduces to 19 Mpps. For $g = 32$, when $p \leq 4$, the throughput is 19 Mpps because LPM is the bottleneck due to wide entry and it continues to be the bottleneck even if $p = 8$.

With NLTMC, it is clear that the configuration with fewer subsets gives better throughput due to shorter LPM words. On the other hand, it should be recalled that it can also cause more memory consumption due to more crossproducts as discussed above. Hence there is a trade-off between throughput and memory requirement. Another interesting point to note is that in some cases, NLTSS shows a better throughput than NLTMC$_1$6 and in some cases it is the opposite. In case of fw3s, all the NLTMC configurations offer a consistently high throughput because there are 151 bits in the NLT bit-map of the LPM entries of NLTSS which slows it down. Hence, in such cases, restricting the number of subsets to a smaller value through merging and crossproducts makes sense. Overall, it can be seen that the throughput depends on the nature of the rule set and appropriate configuration can be chosen that suits the requirements.

7 Summary

TCAM is widely used for high-speed packet classification. However, due to the excessive power consumption and the high cost of TCAM devices, algorithmic solutions that are cost-effective, fast and power-efficient are still of great interest. In this paper, we propose an efficient solution that meets all of the above criteria to a great extent. Our solution combines Bloom filters implemented in high-speed on-chip memories with our Multi-Subset Crossproducting Algorithm. Our algorithm can classify a single packet in only $4 + p$ memory accesses on an average where p is the number of rules a given packet can match. The classification reports all the p matching rules. Hence, our solution is naturally a multi-match algorithm. Furthermore, the pipelined implementation of our algorithm can classify packets in $max\{4, p\}$ memory accesses.

Due to its primary reliance on memory, our algorithm is power-efficient. It consumes about an average 30 to 36 bytes per rule of memory (on-chip and off-chip combined). Hence rule sets as large as 128K can be easily supported in less than 5MB of SRAM. Using two 300MHz 36-bit wide SRAM chips, packets can be classified at OC-192 speed.

References

[1] IDT Generic Part: 71P72604
. http://www.idt.com/?catID=58745&genID=71P72604.

[2] IDT Generic Part: 75K72100
. http://www.idt.com/?catID=58523&genID=75K72100.

[3] Florin Baboescu and George Varghese. Scalable Packet Classification. In *ACM SIGCOMM*, 2001.

[4] Sarang Dharmapurikar, P. Krishnamurthy, and Dave Taylor. Longest Prefix Matching using Bloom Filters. In *ACM SIGCOMM*, August 2003.

[5] Will Eatherton. Fast IP Lookup Using Tree Bitmap. *Washington University Master Thesis*, 1999.

[6] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.

[7] T. V. Lakshman and D. Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM*, 1998.

[8] K. Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for Advanced Packet Classification using Ternary CAM. In *ACM SIGCOMM*, 2005.

[9] Haoyu Song, Sarang Dharmarpurikar, Jonathan Turner, and John Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *ACM SIGCOMM*, 2005.

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

[11] V. Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast and Scalable Layer Four Switching. In *ACM SIGCOMM*, 1998.

[12] David Taylor and Jon Turner. Scalable Packet Classification Using Distributed Crossproducting of Field Labels. In *IEEE INFOCOM*, July 2005.

[13] David E. Taylor. Survey and taxonomy of packet classification techniques. *Washington University Technical Report, WUCSE-2004*, 2004.

[14] David E. Taylor and Jonathan S. Turner. Classbench: A Packet Classification Benchmark. In *IEEE INFOCOM*, 2005.

[15] Fang Yu and Randy H. Katz. Efficient Multi-Match Packet Classification with TCAM. In *IEEE Hot Interconnects*, August 2003.

[16] Fang Yu, T. V. Lakshman, Martin Austin Motoyama, and Randy H. Katz. Ssa: a power and memory efficient scheme to multi-match packet classification. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, 2005.