Fast Packet Classification Using Bloom filters

Sarang Dharmapurikar sarang@arl.wustl.edu
Haoyu Song hs1@arl.wustl.edu
Jonathan Turner jst@arl.wustl.edu
John Lockwood lockwood@arl.wustl.edu

Washington University in Saint Louis, Saint Louis, MO 63130, USA

ABSTRACT

Ternary Content Addressable Memory (TCAM), although widely used in practice for general packet classification, is an expensive and high power-consuming device. 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. Therefore, the development of fast and power-efficient algorithmic packet classification techniques continues to be a research subject.

In this paper we propose a new approach to packet classification, which combines architectural and algorithmic techniques. Our starting point is the well-known crossproduct 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 requirement, without compromising on performance. Unnecessary accesses to the off-chip memory are avoided by filtering them through on-chip Bloom filters. For packets that match p rules in a rule set, our algorithm requires just $4+p+\epsilon$ independent memory accesses, to return all matching rules, where $\epsilon \ll 1$ is a small constant that depends on the false positive rate of the Bloom filters. Using two commodity SRAM chips, a throughput of 38 Million packets per second can be achieved. 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 to 1.4. The average memory consumption per rule is 32 to 45 bytes.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

Keywords

Packet Classification, TCAM

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS'06, December 3–5, 2006, San Jose, California, USA.
Copyright 2006 ACM 1-59593-580-0/06/0012 ...$5.00.

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 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 the 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 has been able to displace TCAMs, in practical applications.

TCAMs offer consistently high performance, which is 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 [14], [11]. This gap between SRAM and TCAM cost and power consumption makes it worthwhile to continue to explore better algorithmic solutions.

We propose a memory efficient and fast algorithm based on Bloom filters. Our starting point is the naive crossproduct algorithm [9]. Naive crossproduct suffers the exorbitant memory overhead due to the extra crossproduct rules introduced. 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. 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 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 proba-

bility of the Bloom filters. The first four memory accesses are required to perform a Longest Prefix Matching (LPM) on the source/destination addresses and the source/destination ports. The next p memory accesses are requires to lookup the p matching rules for a given packet. Furthermore, the LPM phase and the rule lookup phase can be pipelined with two independent memory chips such that the memory accesses per packet can be reduced to $max\{4, p\}$. 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 32 to 45 bytes per rule.

The rest of the paper is organized as follows. In the next section we discuss the related work. We describe the naive crossproduct algorithm in more details in Section 3. In Section 4, we discuss our Multi-Subset Crossproduct 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 evaluate and discuss the performance of our algorithm in terms of memory requirement and throughput. Section 7 concludes the paper.

2. RELATED WORK

There is a vast body of literature on packet classification. An excellent survey and taxonomy of the existing algorithms and architectures can be found in [11]. Here, we discuss only the algorithms that are closely related to our work. Algorithms that can provide deterministic lookup throughput is somewhat akin to the basic crossproduct algorithm [9]. The basic idea of the crossproduct 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 [5] or by using any range searching or LPM algorithms. The BV [6] 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 as in RFC, the DCFL [10] 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.

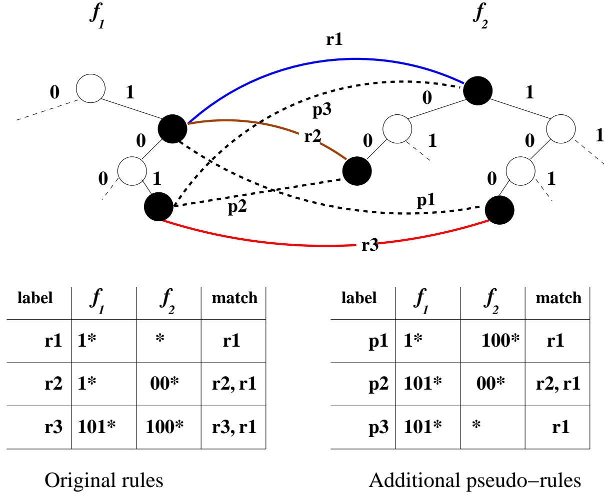
TCAMs are widely used for packet classification. Latest TCAM devices also include the banking mechanism to reduce the power consumption by selectively turning off the unused banks. Traditionally, TCAM devices needed to expand the range values into prefixes for storing a rule with range specifications. The recently introduced algorithm, DIRPE [7], uses a clever technique to encode ranges differently which results in overall lesser 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 [13]. 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 [14], 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. 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 to probe all the subsets formed, one by one, requiring as many TCAM accesses whereas our algorithm needs only p memory accesses, just as many matching rules as there are per packet.

3. NAIVE CROSSPRODUCT ALGORITHM

Before we explain out multi-subset crossproduct algorithm, we explain the basic crossproduct algorithm. The basic crossproduct algorithm first performs a longest prefix match (LPM) on each field. Let v_i be the longest matching prefix of field f_i. It then looks up the key $\langle v_1, v_2, \ldots, v_k \rangle$ into a crossproduct rule table (implemented as a hash table). If there is a matching key, then the associated rule IDs are returned. For this algorithm to work, the rule table needs to be modified. It can be explained through the following example. We will assume only two fields, f_1 and f_2, for the purpose of illustration. Let each field be 4-bit wide. Consider a rule set with three rules $r_1 : \langle 1*, * \rangle$, $r_2 : \langle 1*, 00* \rangle$, $r_3 : \langle 101*, 100* \rangle$. We can represent these rules with a trie built for each field as shown in Figure 1. In this figure, the nodes corresponding to the valid prefixes of a field are colored black. A connection between the nodes of two fields represents a rule. It is important to note that a match for $r_2 : \langle 1*, 00* \rangle$ also means a match for $r_1 : \langle 1*, * \rangle$ because the prefix $*$ of the second field is a prefix of 00*. Hence, r_2 is more specific than r_1 or put it differently, r_2 is contained in r_1. Therefore, when r_2 matches, both r_2 and r_1 should be returned as the matching rule IDs. With the similar argument, the matching rule IDs associated with r_3 are r_3 and r_1.

Suppose a packet arrives whose field f_1 has a longest matching prefix 101* and field f_2 has 00*. There is no original rule $\langle$101*, 00*$\rangle$. However, note that 1* is a pre-



label	f1	f2	match
r1	1*	*	r1
r2	1*	00*	r2, r1
r3	101*	100*	r3, r1

Original rules

label	f1	f2	match
p1	1*	100*	r1
p2	101*	00*	r2, r1
p3	101*	*	r1

Additional pseudo–rules

Figure 1: Illustration of the basic crossproduct algorithm

fix of 101*. Therefore, a match for more specific prefix 101* automatically implies a match for less specific prefix 1*. Hence, if we try to look for the key ⟨101*, 00*⟩, then we should get a match for the rule $r_2 : \langle 1*, 00*\rangle$. To maintain the correctness of the search, we add a *pseudo rule* to the rule table, $p_1 : \langle 101*, 00*\rangle$ and associate the rule ID r_2 with it. Similarly, if 1* is the longest matching prefix for f_1 and 100* for f_2, then although there is no original rule ⟨1*, 100*⟩, it implies a match for $r_1 : \langle 1*, *\rangle$. Hence, we need to add a pseudo-rule $p_2 : \langle 1*, 100*\rangle$ to the table and associate rule ID r_1 with it. To summarize, a match for a prefix is also a match for its shorter prefixes. When the search on individual fields stops after longest matching prefix has been found, these prefixes must also account for the rules formed by their sub-prefixes. If a rule formed by the longest matching prefixes does not exist in the table, it must be artificially added to the set and the original rule IDs that should match are associated with this new pseudo rule. It is not hard to see that if we build such a rule table then the only additional pseudo-rules required are p_1, p_2, and p_3 as shown in Figure 1.

Let $P.f_i$ denote the value of field i in packet P. The packet classification process can be summarized in the following pseudo-code.

ClassifyPacket(P)
1. **for each** field i
2. $\quad v_i \leftarrow LPM(P.f_i)$
3. $\{match, \{Id\}\} \leftarrow HashLookup(\langle v_1, \ldots, v_k\rangle)$

As the algorithm describes, we first execute LPM on each field value. Then we look up the key formed by all the longest matching prefixes 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 protocol 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 [4], we need approximately one memory access per LPM. Therefore, the entire classification process takes five memory accesses with very high probability to classify a packet. However, the overhead of pseudo-rules can be very large. We consider another example rule set as shown in Figure 2(A) and add pseudo-rules to it. As can be seen, this rule set with six rules requires seven more pseudo-rules. If each field has 100 unique prefixes 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. We analyzed several real and synthetic rule sets and found that the naive crossproduct expands the rule set by a *factor* of 200 to 5.7×10^6! Clearly, due to a very large overhead of pseudo-rules, this algorithm is impractical. We modify this algorithm to reduce this expansion overhead to just a factor of 1.2 on an average while retaining the speed.

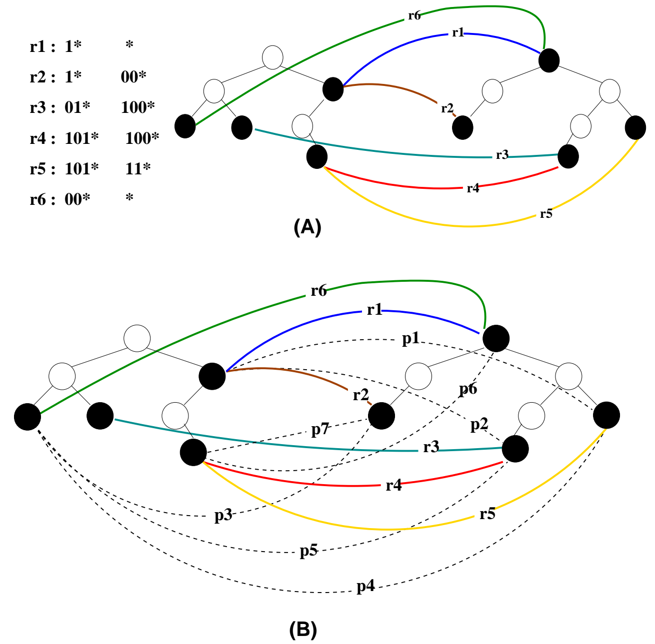
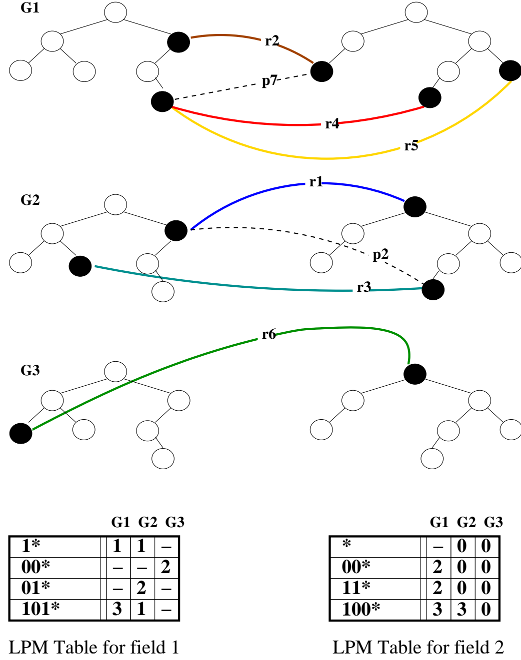


Figure 2: An example of naive crossproduct. (A) The rule set and its trie-based representation. (B) Rule set after adding the pseudo-rules.

4. MULTI-SUBSET CROSSPRODUCT 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 3. We divide the rule set into three subsets. How to form these subsets is discussed in Section 5. For now, let's assume an arbitrary partitioning. Within each subset, we take a crossproduct, This results in inserting pseudo-rules p_7 in subset 1 (G_1) and p_2 in subset 2 (G_2). All the other



	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

Figure 3: Dividing rules in separate subsets to reduce overlap. The corresponding LPM tables.

pseudo-rules shown in Figure 2(B) 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. Due to the rule set partitioning, the number of overlapping prefixes of a field i get reduced by a factor of x_i, the resulting reduction in the crossproduct rules is of the order Πx_i and hence large. Now, 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 look up the final rule. We now describe how to avoid the first overhead and reduce second overhead.

With reference to our example in Figure 3, due to the partitioning of rules in subsets G_1, G_2 and G_3, the sets of valid prefixes of the first field are {1*, 101*} for G_1, {1*, 01*} for G_2 and {00*} for G_3. Hence, the longest prefix in one subset might not be the longest prefix in other subset requiring a separate LPM for each subset. However, an independent LPM for each subset can be easily avoided by modifying the LPM data structure. For each field, we maintain only one global prefix table 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 any subset is either the prefix that matches or its sub-prefix.* With each prefix in the LPM table, we maintain a list of sub-prefixes, each is the longest prefix within a subset. For instance, consider the field 1 of our example. If we find that the longest matching prefix for this field is 101* then we know that the longest matching prefixes within each group must be either this prefix or its sub-prefix. In G_1, the longest matching prefix is 101*, in G_2 it is 1* and in G_3 it is NULL. Let's denote by t_i an arbitrary entry in the LPM table of field i. t_i is a record that consists of a prefix $t_i.v$ which is the lookup key portion of that entry and the corresponding sub-prefix for g subsets, $t_i.u[1] \dots t_i.u[g]$. Each $t_i.u[j]$ is a prefix of $t_i.v$ and $t_i.u[j]$ is the longest matching prefix of field i in subset j 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 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 its length. The prefix $t_i.u[j]$ can always be obtained by considering the correct number of bits of $t_i.v$.

The LPM table corresponding to the example shown in Figure 3 is also shown. In this example, since we have three subsets, with each prefix we have three entries. For instance, the table for field 1 tells 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*` itself) that is the longest prefix in G_1; there is a sub prefix of length 1 (which is `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. 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 G_1 rule hash table with the key $\langle$`101, 100`$\rangle$, G_2 rule hash table with the key$\langle$`1, 100`$\rangle$ and we don't need to probe 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**($t_i.u[j] == NULL$) skip this subset
6. $\{match, \{Id\}\} \leftarrow HashLookup_j(t_1.u[j], \dots, t_k.u[j])$

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 3-6). A probe is not required for a subset if there is no sub-prefix corresponding at least one field within that subset. In this case, we simply move to the next subset (line 4-5). 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 avoided 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.* With a very high probability, only those Bloom filters containing a matching rule show a match. The flow of our

algorithm is illustrated in the Figure 4.

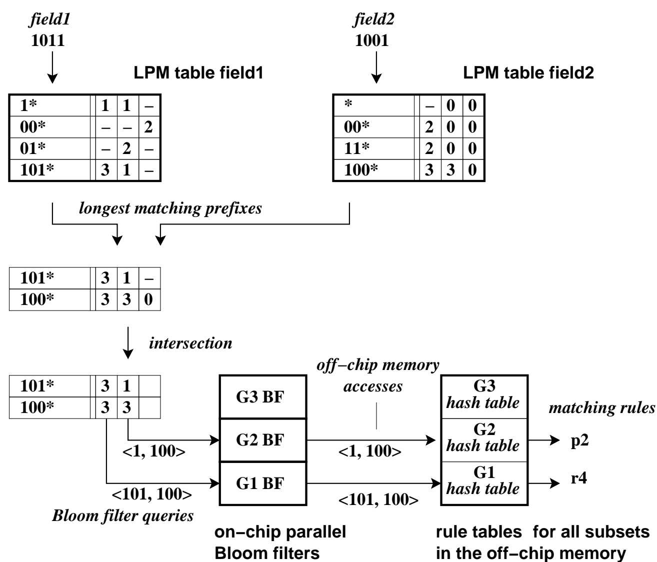


Figure 4: Illustration of the flow of algorithm. First, LPM is performed on each field. The result is used to form a set of g tuples, each of which indicates how many prefix bits to use for constructing keys corresponding to that subset. The keys are looked up in Bloom filters first. Only the keys matched in Bloom filters are used to query the corresponding rule subset hash table kept in the off-chip memory.

The average memory accesses for LPM on field i using the LPM technique discussed in [4] can be expressed as

$$t_i = 1 + (W_i - 1)f \qquad (1)$$

where W_i is the width of field i in bits and f is the false positive probability of each Bloom filter (assuming that they have been tuned to exhibit the same false positive probability). 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

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

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

$$T_g = p + (g - p)f \qquad (3)$$

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

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

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 $\approx 4 + p$. 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

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. 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 pseudo-rules are required only when rules within the same subset have overlapping prefixes. 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 pseudo-rules? We address these questions in this section.

5.1 Overlap-free Grouping

We illustrate a simple technique based on the concept of *Nested Level Tuple* which gives a partition of rules into subsets such that no subset needs to generate crossproduct rules. We call the resulting algorithm Nested Level Tuple Space Search (NLTSS). We then illustrate a technique to reduce the number of subsets formed using NLTs to a predetermined fixed number by merging some of the subsets and producing crossproduct for them.

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 ancestors of this node which are also marked. We always assume that the root node is*

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 5 illustrates a nested level tree for field f_1 in our example rule set.

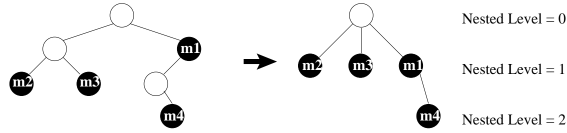


Figure 5: Illustration of Nested Level Tree

Nested Level Tuple (NLT): *For each field involved in the rule set, we create a Nested Level Tree (See Figure 6). 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 6, the NLT for r_6 is [1,0] and for r_4 is [2,1].

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. *Hence the set of rules contained in the same Nested Level Tuple do not need any crossproduct rules.* This is illustrated in Figure 6. For instance, in NLT [1,0] there are two rules r_1 and r_6. Their prefixes for field 1 are 1* and 00*, none of which is more specific than the other (hence overlap-free). Likewise, they share the same prefix, that is *, for field 2. Therefore, no crossproduct is required.

The number of NLTs 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, 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.

A property of the NLT rule subset is that a prefix does not have any sub-prefix within the same subset. Using this property, the LPM entry data structure shown in Figure 4 can be compressed further. Associated with each prefix, we can maintain a bit-map with as many bits as there are NLTs. A bit corresponding to a NLT in this bit-map is set if the prefix or its sub-prefixes belongs to a rule that is contained in this NLT. We can take an intersection of the bit-maps associated with the longest matching prefix of each field for pruning the rule subsets to look up.

However, given an NLT, we just know the nested level associated with each prefix. We don't know the exact prefix length 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. We 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 *prefix length tuple* from a NLT using the PL/NL bit-maps associated with the involved prefixes. The prefix length tuple tells 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 the Figure 7. 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, i.e. the 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 prefix length tuples. 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 not 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 4.

5.2 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 that we set to explore in the beginning of this section: 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

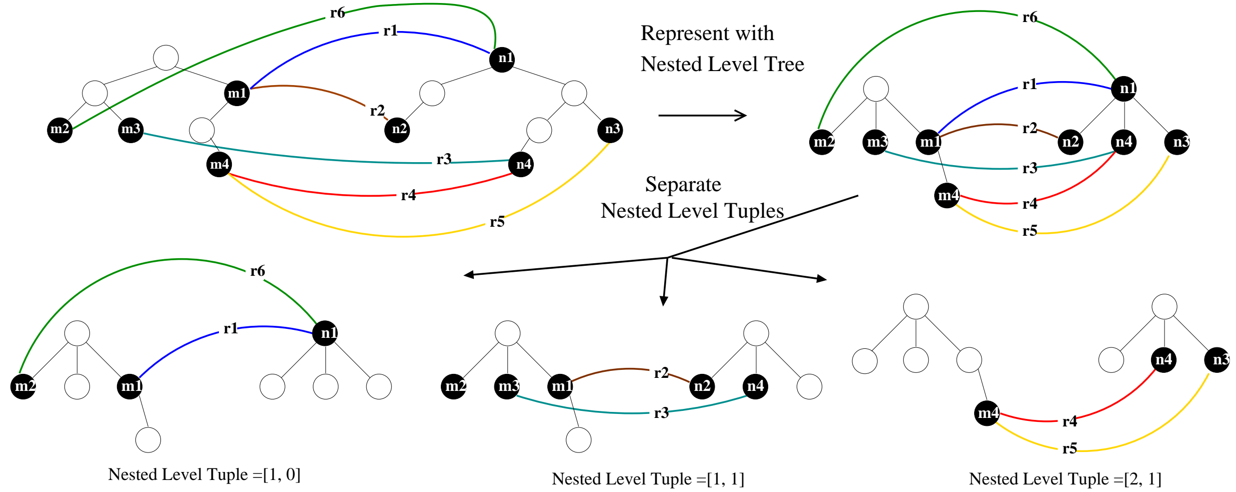


Figure 6: Crossproduct-free grouping of rules using Nested Level Tuples (NLT)

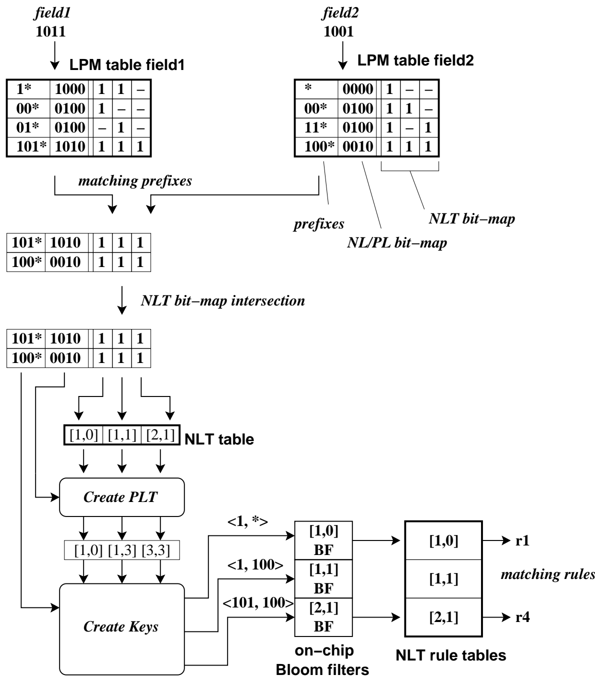


Figure 7: Using NLT based grouping to form the subsets. Each prefix entry in LPM table needs a NL/PL bit-map and another bit-map indicating the NLTs to which the prefix or its sub-prefixes belong.

it by merging some of the NLTs and applying crossproduct technique to them in order to limit the number of subsets. Fewer subsets also means fewer Bloom filters and hence a resource efficient architecture. In the next subsection, we describe our NLT merging technique and the results after applying the crossproduct algorithm.

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 8 shows the plot of the cumulative distribution of rules across the number of NLTs.

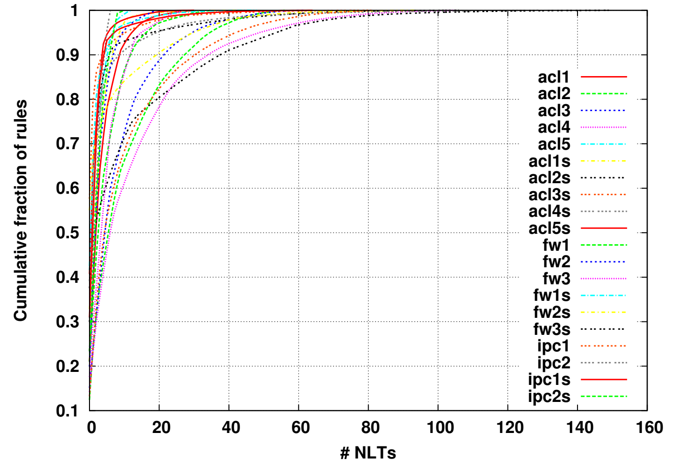


Figure 8: Cumulative distribution of the rules across NLTs. More than 90% rules are contained within just 40 NLTs.

This indicates that we can take care of a large fraction of rules with just a few subsets. Hence we design an NLT merging algorithm such that we start with the overlap-free NLT set, retain the most dense (i.e. containing a large number of rules) 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.

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

2. 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.

3. 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 the "closest" NLT. 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.

4. 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 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.

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 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 $t=20$.

				g=16		g=32	
set	rules	NLT	prefixes	α	β	α	β
acl1	1247	31	610	1.03	0.00	1.00	0.00
acl2	1216	57	437	1.93	4.19	1.17	0.00
acl3	4405	63	1211	1.29	4.45	1.14	0.25
acl4	5358	107	1445	1.74	7.95	1.20	0.62
acl5	4668	14	304	1.00	0.00	1.00	0.00
acl1s	12507	45	1524	1.03	0.28	1.00	0.00
acl2s	18589	107	626	1.12	2.32	1.14	0.39
acl3s	17395	81	947	3.99	0.71	2.26	0.21
acl4s	16291	130	1090	1.46	2.22	1.42	0.42
acl5s	13545	31	2401	1.03	0.00	1.00	0.00
fw1	914	37	205	1.37	0.11	1.03	0.00
fw2	543	21	132	1.06	0.00	1.00	0.00
fw3	409	29	147	1.25	0.00	1.00	0.00
fw1s	32135	50	337	1.92	0.80	1.09	0.006
fw2s	26234	95	271	1.60	2.81	1.46	0.42
fw3s	24990	151	460	1.53	6.45	1.80	0.94
ipc1	2179	83	396	1.73	5.69	1.41	0.73
ipc2	134	8	72	1.00	0.00	1.00	0.00
ipc1s	12725	65	519	1.86	1.09	1.03	0.09
ipc2s	9529	11	4596	1.00	0.00	1.00	0.00
avg				1.43	1.95	1.20	0.34

Table 1: Results with different rule sets. 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.

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 $g=32$ shows a larger expansion compared to $g=16$. This is because the $g=16$ configuration throws out more number of spoilers compared to the $g=32$ configuration. 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 very few spoilers and a small expansion factor. At the same time, $g=32$ needs to use wider LPM entries since the information regarding all the 32 subsets needs to be kept with each prefix. The wider entry also implies more cycles spent in memory accesses and hence lesser throughput.

6. PERFORMANCE

in this section we evaluate the performance of NLTSS and NLTMC algorithms in terms of the memory requirement and throughput.

6.1 Memory Requirement

The two data structures that consume memory in our algorithm are: (1) On-chip Bloom filters (2) off-chip item memory. For an efficient implementation of on-chip Bloom filter and off-chip hash tables, we use the Fast Hash Table (FHT) architecture proposed in [8]. If we use 12 hash functions for 128K items and a ratio of 16 buckets per item then it can be shown that the number of colliding items is less than 75. This is an acceptably small collision. All the colliding items can be kept in an on-chip memory. Therefore, it is reasonable to assume that we need just one memory access to look up an item in FHT. With the aforementioned tuning, it can be shown that the Bloom filters' false positive probability is 0.00046, low enough for our purpose. To achieve this performance, the required on-chip memory per *item*, where an item can be either a rule or a prefix, is 64 bits. Therefore, the average number of on-chip bits *per rule* is given as:

$$m_o = \frac{64 \times (\alpha_g \times \#rules + \#prefixes)}{\#rules \times 8} \tag{5}$$

where $\alpha_g \times \#rules$ is the size of the new rule set after expansion by using g subsets. In case of NLTMC, the # prefixes and α_g can be obtained from Table 1 for each rule set. For NLTSS, the number of prefixes are the same, however there is no rule set expansion and hence $\alpha_g=1$.

To evaluate the off-chip item memory requirement, we need to consider two types of items: the actual rules and the prefix entries in the LPM tables. In order to specify an arbitrary length prefix of a W-bit field, $W+1$ bits are enough [12]. Hence, the actual rule can be specified using 33 bits for each source and destination IP prefix, 17 bits for each source and destination port prefix, and 9 bits for protocol (totally 126 bits). We round it up to the nearest multiple

of 36 (i.e. 144) since the commodity off-chip SRAM is available in this words size. On the other hand, the number of bits required to specify the prefix entry in the LPM tables depends on the algorithm used. For NLTSS, as shown in Figure 7, each prefix entry contains a W-bit prefix value, a W-bit PL/NL bit map, a g-bit NLT bit-map, and two more bits for distinguishing the type of prefix – source/destination IP or source/destination port (not shown in the figure). Totally, each entry requires $b_{NLTSS} = \lceil (2W + 2 + g)/36 \rceil \times 36$ bits. For NLTMC$_g$, as shown in Figure 4, we require a W-bit prefix, two more bits to specify prefix type, and g entries to specify the sub-prefix in each of the g subsets. Each of these g entries consumes 6 bits, totally requiring $b_{NLTMC_g} = \lceil (34 + 6g)/36 \rceil \times 36$ bits. After taking into account the overall expansion (specified by α_g), the off-chip memory per item can be given as

$$m_f = \frac{\#rules \times \alpha_g \times 144 + \#prefixes \times b}{\#rules \times 8} \quad (6)$$

Again, $\alpha_g = 1$ for NLTSS. The resulting on-chip and off-chip memory consumption is shown in Table 2. We consider only the NLTMC$_{16}$ configuration because the configuration with more number of subsets results in wider LPM entries which degrades the throughput (to be discussed next). Table 1 shows that the difference in the average rule set expansion is not significant with 16 subsets and 32 subsets (1.42 vs. 1.2). However, the $g = 16$ configuration results in a larger throughput due to shorter LPM entries.

It can be observed that NLTMC consumes more on-chip and off-chip memory compared to NLTSS because NLTMC expands the rule set due to crossproducts and NLTSS doesn't. The average memory consumption for NLTMC$_{16}$ is 45 bytes (on-chip + off-chip) and for NLTSS it is 32 bytes.

6.2 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 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 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. Since a single rule fits in 144 bits (four 36-bit words), two clock cycles are required to lookup a rule into the SRAM. If a packet matches p rules in a rule set then, with a pipelined implementation, a packet can be classified in time $max\{4\tau_{lpm}, 2p\}$. Typically, p is ≤ 6 as noted in [7] [5]. We will evaluate the throughput for different values of p.

Choice of algorithm: With the 300 MHz, 36-bit wide DDR-SRAM, the throughput can be expressed as

$$R = \frac{300 \times 10^6}{max\{4\tau_{lpm}, 2p\}} \quad packets/second \quad (7)$$

where τ_{lpm} is either $\lceil b_{NLTSS}/72 \rceil$ or $\lceil b_{NLTMC_g}/72 \rceil$, depending upon which algorithm is chosen. The throughput is shown in the Table 2.

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

Table 2: The performance algorithms. m_o and m_f 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.

Let's consider the case of NLTSS. When $p \leq 4$, the $max\{4\tau_{lpm}, 2p\} = 4\tau_{lpm}$ due to which 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$_{16}$ 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. 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. While NLTMC shows better through-

put for all configurations compared to NLTSS, it also incurs the associated rule expansion cost as discussed before. Therefore, there is a trade-off between the throughput and memory consumption. Depending upon the requirement, appropriate algorithm and configuration can be chosen.

Secondly, a packet can match p rules when there are p overlapping rules that cover the common region in the 5-dimensional space. Although [7] and [5] report that the number of matching rules can be as high as 8, the *instances* of such overlaps are usually very few. A simple solution to improve throughput in such cases is to simply take out the overlapping rules and keep it in the on-chip memory so that p stays below a given threshold. In each rule set we experimented with, we encountered only one to two instances where there was an overlap beyond four. Therefore, it is very easy to achieve the throughput of $p \leq 4$ by removing this overlap.

7. CONCLUSIONS

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 Crossproduct 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 an average 32 to 45 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 the rate of 38 Million packets per second (OC-192 is equivalent to 31 Mpps with 40-byte packets).

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

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

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

[8] 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.

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

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

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

[12] George Varghese. Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices. 2005.

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

[14] 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.