A Packet Classification Algorithm for Multiple Look-up Engines

Douglas Low, Jean-Loup Baer
Department of Computer Science and Engineering
University of Washington
{douglas, baer}@cs.washington.edu

Technical Report UW-CSE-2005-02-02

February 2005

Abstract— **We present a packet classification algorithm suitable for implementation on programmable network processors (NP). The current trend for NPs is to have several microengines with limited local storage capabilities. The proposed algorithm is therefore decomposable into several modules, both in terms of computation and of the indexing data structure for classification purposes. We show how this decomposition can lead to an efficient pipelined process and how the decomposition of the index structure can result in mapping into local storage the most often accessed and search intensive parts of the index.**

The algorithm is tested on 3 Access Control Lists with between 750 and 2300 rules. We show possible trade-offs between local fast storage and number of memory accesses. We compare storage and memory accesses with the same metrics for a state-of-the-art algorithm using a monolithic data structure.

I. Introduction

Internet Protocol (IP) packet classification is the process of identifying the highest priority rule which matches a packet header from a given set of rules. These rules consist of a tuple of patterns which are matched against several fields of a packet header. IP forwarding, i.e., longest prefix match of a destination address [1] or one-dimensional classification, used to be the main task of core Internet routers. Now multi-dimensional classification is required to support applications such as differentiated services [2], [3] and load balancing, which provide guarantees on packet transmission throughput and latency, and firewalls, which deal with network security. The growing speed of communication and the dictum that there should be "no queuing before header processing" [3] requires efficient algorithms to perform these applications. Software implementations of these algorithms are favored over hardware approaches, since Application Specific Integrated Circuits (ASICs) lack flexibility and Ternary Content Addressable Memories (TCAMs) [4] lack scalability.

In this paper we present a packet classification algorithm suitable for implementation on programmable network processors (NP), which is the architecture of emerging router platforms. The common structure of an NP architecture is a multiprocessor on a chip, with a control processor and multiple microengines. Our algorithm uses the control processor to preprocess data structures needed for packet classification, while the microengines handle the various parts of the classification process. The microengines have a limited amount of control store and fast local storage (SRAM). The local storage may be shared with other microengines and direct communication is possible with a subset of all the microengines. There is also slower memory to store the set of packet classification rules. To this end, this paper concentrates on designing a packet classification algorithm which may be decomposed into parallel or pipelined stages, with each stage assigned to a microengine. During a stage a microengine accesses relatively small data structures which fit into fast local storage and accesses slow memory a bounded number of times. The worst-case performance for packet classification is the worst-case time of the slowest stage.

We formally define in Section II the set of rules, also called a classifier, the problem that packet classification solves, the performance metrics of interest and conditions we place on the solution. We discuss previous work in Section III and note that although the packet classification problem can be posed in terms of multidimensional computational geometry, the bounds on storage and speed require using heuristics which take advantage of the struc-

ture of the sets of rules. We examine three rule sets to exemplify typical structures observed.

Our algorithm is presented in Section IV. Similar to several other algorithms, we treat the prefix matches used for source and destination address fields differently from the rest of the fields, namely source and destination ports and protocol. We have two new contributions for address prefix matches. The first contribution is in our handling of rules with different prefix lengths and wildcards. Dividing rules into "long" prefixes, "short" prefixes and wildcards allows for significant memory savings. The second contribution is how we prune the set of possible rule matches by performing partial matches on source and destination address fields in alternation, which generates a set of small data structures (hash tables). We describe the preprocessing of the set of rules to construct our index data structure and the classification process for an incoming packet. We sketch how to update the index in real-time. Building the data structure takes an appreciable amount of time as compared to classifying a packet but this is mitigated by only needing to build the data structure once.

In Section V we show how our algorithm lends itself to mapping onto several microengines. Using three rule sets mentioned above, we derive the amount of local storage required as well as the timing of the search operation, in terms of the numbers of fast and slow memory accesses. In Section VI we compare storage and memory accesses with the Extended Grid of Tries (EGT) algorithm, which is monolithic.

We summarize our contributions and results in Section VII and suggest further areas for study.

II. Definitions, Metrics and Criteria

A. Definition

A classifier is a set of N rules, $S = R_1, R_2, \ldots, R_N$, where each rule has k fields or *dimensions* corresponding to k fields in a packet header. For each dimension a rule prescribes either an exact match, a prefix match, a range match or a wildcard (any value is a match). Each rule has a priority and an associated action. Table I shows an example classifier, although for brevity we omit the action for a rule and the priority is implicit in the rule number, with lower rule numbers having higher priority.

The packet classification process consists of finding the rule R_i of highest priority that matches the packet header in all dimensions. Then the action associated with the rule is taken.

In this paper we consider only 5-dimensional classifiers where the first two fields are the source and destination addresses with exact or prefix matches, the third and fourth fields are source and destination ports with either exact or range matches, and the last field is a protocol number which requires an exact match.

B. Performance Metrics and Solution Criteria

Packet classification is a real-time task, i.e. there are constraints on the time that can be allotted to it. The performance metric of choice is the worst-case classification time for an "adversary" packet. The processor time to search data structures that are used as indices for the dimensions of the classifier is often negligible compared to the time to access them, thus worst-case time is generally translated into *worst-case number of memory accesses*. However the number of memory accesses can be reduced by increasing the size of the data structures while reducing their depth, i.e. worst-case series of memory accesses. In order to keep the data structures of reasonable size e.g. the size of a second or third level cache, the memory usage cannot be unduly increased. *Memory size* is also an important performance metric.

We shall see in the next section that most of the efficient software algorithms require extensive preprocessing and often rather complex data structures. An important criterion for some packet classifiers is that it should be possible to update rules quickly e.g. to block packets from particular IP addresses when under a denial of service attack. An attribute of the packet classifier is thus *online updates*.

Our algorithm is geared towards network processors that have several microengines with associated fast SRAM on-chip and slower DRAM off-chip. The algorithm should take advantage of this architecture i.e. it should be *modular*. Such modularity encompasses decomposition into several processes which can be performed concurrently and/or in a pipelined manner. A second aspect of modularity is that the microengines assigned to the processes only need to access a portion of the fast memory. Similarly we should not require complete interconnection between the microengines. Buses connecting subsets of the microengines and banks of fast memory should be sufficient, thus avoiding costly structures such as crossbars or multistage interconnection networks.

Current classifiers can contain up to a few thousand rules. It has been projected that this number could increase by one order of magnitude. *Scalability* of the algorithm is a criterion which cannot be neglected.

III. Review of Previous Work and Characteristics of Rule Sets

A. Previous Work

Several researchers [5], [6] noted that packet classification resembles the computational geometry problem in

TABLE I

AN EXAMPLE OF A PACKET CLASSIFIER. THE "ANY" VALUE IS A WILDCARD FOR A DIMENSION. THE ACTION FOR A RULE IS OMITTED FOR BREVITY AND THE PRIORITY IS IMPLICIT IN THE RULE NUMBER – LOWER RULE NUMBERS HAVE HIGHER PRIORITY.

Rule Number	Source Address Prefix	Destination Address Prefix	Source Port Range	Destination Port Range	Protocol
R0	0000*	01*	0 - 1023	1024 - 65535	17
R1	0000*	01*	1024 - 4999	1024 - 8000	8
R2	0000*	01*	5000 - 65535	8001 - 65535	8
R3	0000*	111*	Any	0 - 1023	17
R4	0000*	111*	Any	1024 - 65535	1
R5	0011*	01*	Any	6000 - 7000	8
R6	01*	1*	Any	7001 - 8000	8
R7	10*	0*	Any	8001 - 9000	17

k-dimensional space stated as: Given a point (packet) and a set of N k-dimensional objects (the packet classifier), find the object that the point belongs to. If the objects are non-overlapping, a slightly easier problem than packet classification, the best bounds for $k > 3$ are $O(\log N)$ time with $O(N^k)$ space or $O(\log N^{k-1})$ time with $O(N)$ space. For classifiers of 1,000 rules of 10 bytes each (an underestimate) and five dimensions, either too much space (on the order of 10 million Gigabytes) would be used, or the execution time would be too slow (on the order of 10,000 memory accesses). Thus there is a need to either use special hardware or employ heuristics which take into account the characteristics (semantics) of the rule sets.

1) Hardware Solutions: Many current classifiers of limited size are based on Ternary Content Addressable Memory (TCAM), which stores patterns of the form (*value, mask*) by having each "bit" of the pattern be either 0, 1 or "don't care" (X). An input *in* has a match in the TCAM if the logical intersections *value* $\wedge$ *mask* and *in* $\wedge$ *mask* are equal. A row (k patterns) of the TCAM represents a rule.

The great advantage of TCAMs is speed – all rules are checked in parallel. There are however several drawbacks. Each TCAM cell uses 16 transistors whereas an SRAM cell uses 6, making TCAMs significantly more expensive. TCAMs support only prefix and exact matches, therefore ranges must be expanded into equivalent sets of prefixes. There is no theoretical difficulty in doing so [7] but this significantly increases the number of rules, requiring more chip area. Since all rules are matched in parallel, the chip consumes up to a factor of 100 more power than an SRAM of similar capacity [4]. Online updates of TCAM-based classifiers are difficult since either the rules have to be sorted in priority order or a priority encoder is required, which is difficult to modify online. These considerations make TCAMs more suitable for IP lookup which only requires prefix matches for the destination address. Current research on how to partition TCAMs to reduce the power consumption and exploration of new circuits to perform range matches are underway [4].

Lakshman and Stiliadis have developed an algorithm well suited to ASIC implementation [6]. Queries for each dimension are converted into range matches. The total range of a dimension is partitioned into non-overlapping intervals and the rules that match the query for each interval are encoded into a priority ordered bitmap vector. The correct interval in each dimension is determined by binary search. Each dimension can be searched in parallel, producing a bit vector of the matching rules in that dimension. However the final step of the algorithm that involves finding the most significant bit in the intersection of the k bit vectors has $O(N)$ time in the worst case. Moreover the priority encoding in the bit vectors prevents easy online updating.

2) Software Solutions: An excellent survey of software solutions to the packet classification problem up to 2001 was performed by Gupta and McKeown [5]. We review some recent approaches, citing previous papers where relevant.

A number of solutions are based on trie search, a highly successful technique for IP lookup [1]. Extension to several dimensions is however fairly complex if one wants to avoid backtracking when searching for a matching rule. In the case where only source and destination address fields are considered, an example of decomposition [8] which we will be using, backtracking can be avoided by judicious insertion of "jump" pointers which guide the search in the second dimension trie when a leaf is reached. The

original grid of tries approach [9] has been recently extended using a path compression technique [10]. This Extended Grid of Tries algorithm (EGT) with path compression (EGT-PC) improves worst-case performance with significant savings in memory usage. However online updates are not feasible and it appears to be quite a challenge to split the algorithm into several stages for pipelining.

The aggregate bitvector algorithm developed by Baboescu et al. [11], [9] follows the approach of Lakshman and Stiliadis [6] described in the previous section, although it is intended to be a software solution. This algorithm uses trie searches on each dimension to generate bit vectors of matching rules, rather than binary interval searches. The aggregate bitvector exploits the sparseness of the rule match bit vectors by maintaining a bitmap recording which groups of bits are not all zero. The logical intersection needs only to be computed on the groups of bits which are not all zero. Nonetheless the worst-case time is still $O(N)$.

B. Characteristics of Access Control Lists

The classifiers which we examined are obtained from router access control lists (ACLs). They are ACLs 1 to 3 from a study by Kounavis et al. [12]. Following the approach taken by this study, we classify rules first by source and destination address prefix pairs, then examine the remaining fields. We found it useful to divide rules into the following categories:

1) Long source and long destination address prefixes
2) Long source and short destination address prefixes
3) Short source and long destination address prefixes
4) Short source and destination address prefixes
5) Source address wildcard
6) Destination address wildcard
7) Source and destination address wildcards

The criteria for considering address prefixes to be short are (a) there are few of them so that they can be classified quickly in the search process and (b) there is a gap between their length and the length of the shortest of the long rules. As we will see later, limiting the number of short prefixes reduces the amount of memory used by our algorithm which extends short prefixes to build the address indexing data structure, thus creating extra prefixes and potentially duplicating rule matches. Rules with wildcards as the source and/or destination address prefix will match any packet in these dimensions, therefore storing these rules with the other rules wastes storage and they do not require the generality of the algorithm described in the next sections. Table II shows the number of rules in each category for the ACLs.

TABLE II
NUMBER OF RULES OF ACLS 1, 2 AND 3 IN EACH RULE CATEGORY.

Rule category	ACL1	ACL2	ACL3
Long src and long dst	601	186	1824
Long src and short dst	54	151	87
Short src and long dst	72	159	71
Short src and short dst	22	12	35
Wildcard src	2	41	198
Wildcard dst	2	33	172
Wildcard src and dst	1	25	12
Total number of rules	754	607	2399

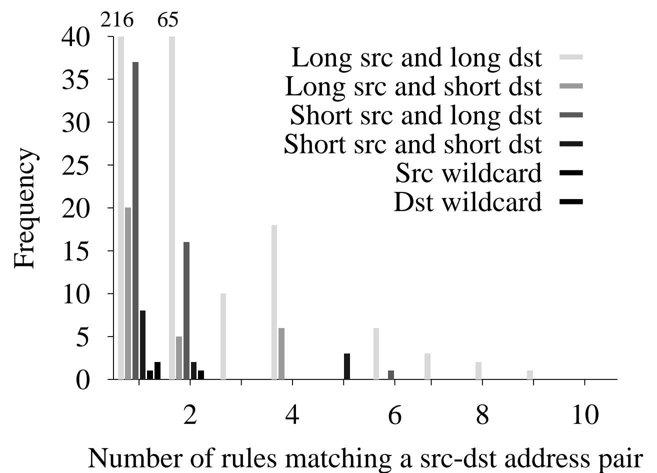

Fig. 1. The distribution of the frequency of rules matching a particular source and destination address prefix for ACL1. The 2 outliers not shown are from the long source and destination address prefixes category at 34 and 37 rule matches, each with frequency 1.

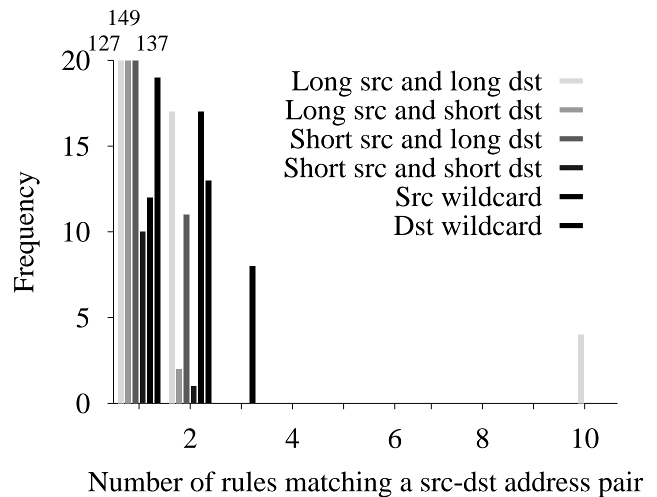

Fig. 2. The distribution of the frequency of rules matching a particular source and destination address prefix for ACL2. There are no outliers.

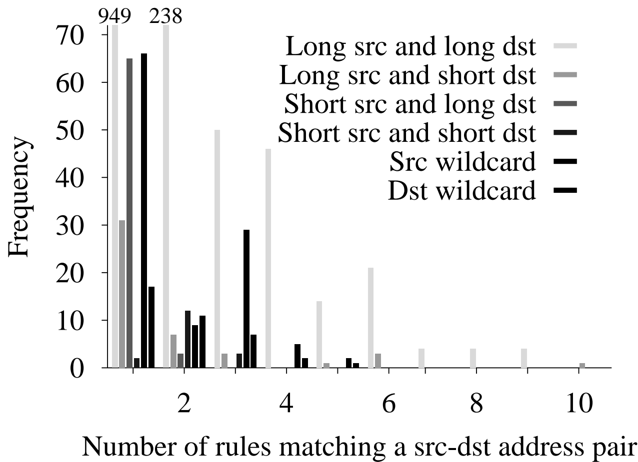

Fig. 3. The distribution of the frequency of rules matching a particular source and destination address prefix for ACL3. There are 5 outliers not shown. 2 of the outliers are from the long source and destination address prefixes category, at 13 and 14 rule matches, with frequency 2 and 1 respectively. 3 of the outliers are from the wildcard destination address prefixes category, at 16, 17 and 18 rule matches, with frequency 9, 5 and 2 respectively.

The distribution of the frequency of rules matching a particular source and destination address prefix are shown in Figures 1 to 3. Out of the 601 long source and long destination address prefix rules in ACL1, 216 of them are uniquely determined by the source and destination address dimension pair. Previous papers [10], [9] note that after source and destination address patterns have been matched, the number of rules remaining to be differentiated between is small (4 or less). We did not observe this behavior in the ACLs we studied. Although in general the number of remaining rules to be matched is less than 10, there are notable outliers. For instance ACL1 has 2 cases where there are 34 and 37 matches respectively for the long source and long destination address prefix category. ACL3 has outliers from two rule categories, 13 and 14 rule matches for the long source and long destination address prefixes and 16, 17 and 18 rule matches for the wildcard destination address prefixes. A more efficient method than a linear search of all these rules is required to classify a packet. These outliers are the motivation for the port range search data structure, which is described in Section IV.

IV. Classification Algorithm

In this section, we first show how to build the data structures that allow us to classify incoming packets. Preprocessing the ACL to facilitate a fast classification is crucial. As important, for our purposes, is that the data structures can be easily decomposed in small units, say of the order of 512 to 1KB.

A. Preprocessing

We first build a data structure to perform a pruning of candidate rules based on the prefix matches of source and destination addresses. The outcome of this first step is either: (1) a number of candidate rules less than some threshold, say 5, or (2) all of the bits of the source and destination addresses have been exhausted.

In case (2), we continue the pruning with a range search on one of the port dimensions. If condition (1) is again not met, we then go on the the second port dimension. This procedure yields a set of candidate rules which due to condition (1) may require some or all dimensions of the packet to be checked. A rule can appear in many different final rule sets, so to save memory, we store the rule numbers instead of the entire rule. The rules themselves are stored in a separate data structure (i.e. ACL) which is not used during a packet classification except during this final stage of the search.

An abstract view of the source and destination address pruning process is shown in Figure 4 for the ACL of Table I. If we were to select a threshold of 3 for condition (1), i.e., we stop the pruning when we have only 1 or 2 candidate rules left to be checked, we would have to perform a port range search only in one case (the leftmost path in Figure 4).

For the range match on the port dimensions and the search of the candidate rules, we use conventional data structures – an array of non-overlapping intervals in the range case and an array in the second. The originality of the algorithm lies in the pruning for the prefix matches. As can be seen in Figure 4, we perform the pruning by building a trie with interleaving pattern matching between bits of the source and destination addresses.

We now give a more concrete representation of the address search data structure.

1) Address Search Structure: We store the children of the internal nodes of the address search structure in hash tables. The size and fill factor of the hash tables can be determined so that we have structures of the size we desire and we can limit the number of comparisons required in a hash table look-up (close to 1). The goal is to ensure the modularity of the data structures so that they can fit into the fast memory of microengines. In Figure 4 the first level of source nodes would in fact be a single hash table of 8 elements with a fill factor of 0.75. On the leftmost path the next hash table would be 4 elements with the same 0.75 fill factor if we want to have sizes that are power of 2 (otherwise 3 elements would be OK). We use open addressing in our hash table implementation since it does not need additional memory to store linked lists to resolve hash collisions and is as fair as chaining when the

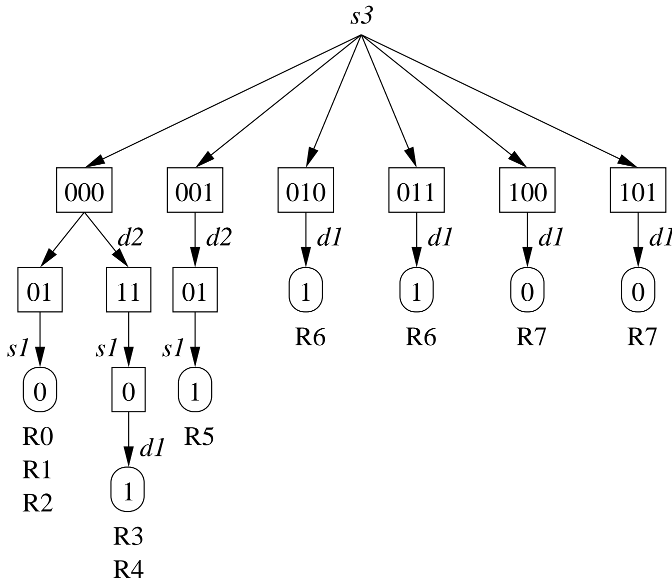

Fig. 4. Address classification data structure for the classifier in Table I. *s#* and *d#* represent # source and destination address bits matched respectively. Rectangles represent internal nodes, circles represent leaves which link to the next stage of the packet classification algorithm.

fill factor is less than 0.75 [13].

When a set of rules is being considered during the multi-bit trie construction, the short address prefixes need to be extended so that all of them have the same length for the k-bit lookup at a trie node. Recall from Section III-B that to limit the amount of extension of rules, we filter out rules with short source or destination address prefixes and build separate indices for them. We also do this for those rules with wildcards for source and/or destination address prefixes.

We record the smallest priority p of the rules in each hash table and port interval search array. When we perform classification (c.f. Section IV-B), we compare the priority k of the best rule match found so far with p and stop the search if $k < p$.

2) Optimization: To find the required child of a trie node, we perform a hash table look-up using a fixed number of bits (Section IV-A.1). We extend short address prefixes, e.g., 00 extended to 4 bits results in the 4 prefixes 0000, 0001, 0010 and 0011. A straightforward implementation of the data structure duplicates subtries. Instead we can replace each copy of a subtrie with a pointer to a single entry. Note, however, that the same number of hash table entries are still required. To determine which subtries would be redundant and should therefore be merged together, we build a DAG (directed acyclic graph) using the address prefixes as keys. Each leaf node in the DAG points to a list of rules with that particular address prefix. Our algorithm processes prefixes in order of increasing length and for each prefix length l, we perform two steps: (1) insert all prefixes with length l (duplicates lead to rule lists with multiple entries) and (2) extend all prefixes (i.e. the leaf nodes) to length $l + 1$. In step (2) the two new children of a leaf node point to the same rule list as their parent did. Applying the DAG building algorithm to the ACL in Table I, we obtain the data structure in Figure 5.

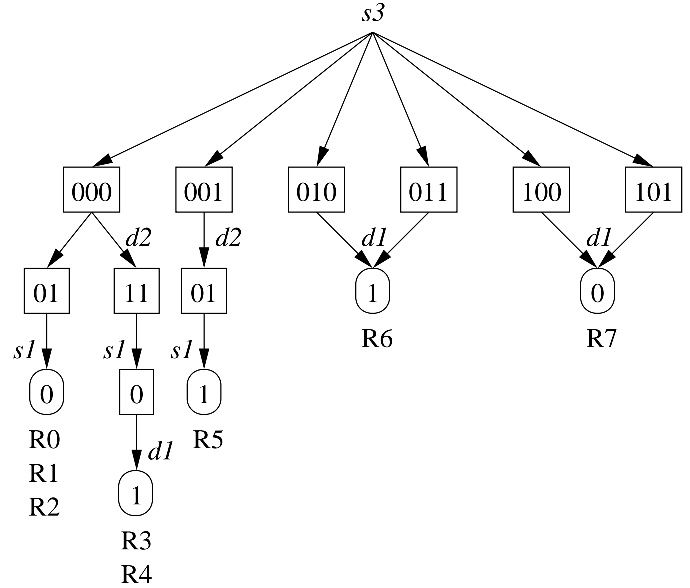

Fig. 5. Address search data structure for the classifier in Table I, which is optimized to remove redundant subtries for R6 and R7. *s#* or *d#* represent # source or destination address bits matched respectively. Rectangles represent internal nodes, circles represent leaves which link to the next stage of the packet classification algorithm.

The DAG cannot eliminate all redundant nodes, it can only eliminate redundant children of a node. To illustrate this point, consider the rule set in Table III; its corresponding address search data structure in Figure 6 has redundant (R8, R9) leaves.

Our data structure building algorithm skips the port interval arrays and goes directly to the candidate rule set search if the rules do not have distinct values in the source and/or destination port dimensions. Figure 7 shows the port interval arrays built for the rules in Table I. Rules R3 to R7 have wildcards for the source port dimension and rules R5 to R7 are all single rules, as are R0 to R2 after the source port search is performed. For sake of example we show the port range search structures here, even though in some cases the number of rules is less than the threshold of 3 used in the example and a candidate rule set search would actually be used in that case.

We begin pruning the source and destination wildcard address rule category using the source port field because the address search structure does not distinguish between rules in this category.

TABLE III

AN EXAMPLE OF A PACKET CLASSIFIER WHICH DEMONSTRATES THE LIMITATION OF THE PREFIX DAG IN REMOVING REDUNDANT NODES. THE "ANY" VALUE IS A WILDCARD FOR A DIMENSION. THE ACTION FOR A RULE IS OMITTED FOR BREVITY AND THE PRIORITY IS IMPLICIT IN THE RULE NUMBER – LOWER RULE NUMBERS HAVE HIGHER PRIORITY.

Rule Number	Source Address Prefix	Destination Address Prefix	Source Port Range	Destination Port Range	Protocol
R8	0*	0*	1024 - 1024	Any	8
R9	0*	0*	Any	8000 - 8000	Any
R10	00*	00*	5000 - 8000	Any	17
R11	01*	1*	Any	Any	8

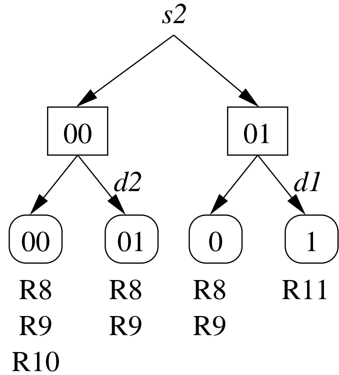

Fig. 6. Address search data structure for the rules in Table III. The redundant nodes containing the rules R8 and R9 cannot be eliminated using a prefi x DAG. *s#* and *d#* represent # source and destination address bits matched respectively. Rectangles represent internal nodes, circles represent leaves which link to the next stage of the packet classifi cation algorithm.

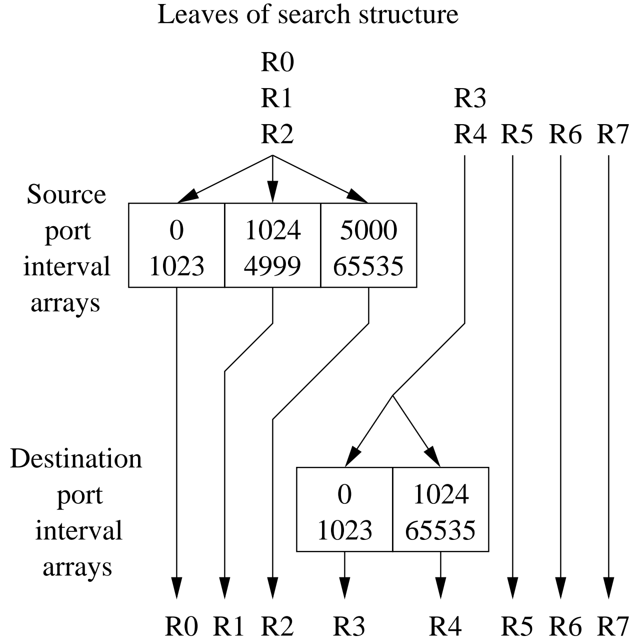

Fig. 7. Source and destination port classifi cation structures for the classifi er in Table I, which are optimized to remove searches over rules whose port dimension values are identical. Rules R3 and R4 have identical source port dimension values, while R5 to R7 are all single rules, as are R0 to R2 after the source port classifi cation is performed.

B. Packet Classification

The packet classification algorithm searches for the best matching rule (lowest rule number / highest priority) in all seven categories of rules. These rule categories are searched in the order given in Section III-B. The non-wildcard categories are searched in three successive stages: (1) an address search, (2) a port range search and (3) a candidate rule set search. The source and destination address wildcard categories have similar search stages, although the first stage does not search the wildcarded address dimension. The source and destination address wildcards category search uses only stages (2) and (3).

The address search involves a multi-bit trie (or DAG) look-up. Each node in the address search structure stores both the number of address bits to use in the hash table lookup and whether they are from the source or destination address dimension. The port range search is a binary search performed on non-overlapping intervals. The candidate rule set search is a linear search through rule numbers but due to condition (1) in Section IV-A, the incoming packet might not have been matched against all dimensions of the rules. Therefore the set indicates which dimensions of the packet remain to be checked.

For sake of brevity in the following examples using the classifier in Table I, assume that addresses are 5 bits in length. Suppose we have a packet with source address 00001 (in binary), destination address 01100 (in binary), source port 256, destination port 6000 and protocol 17. Starting from the root node in Figure 5, the first three source bits of the packet (000) determine that the leftmost child is to be traversed to. The first two bits of the destination address of the packet (01) determine that the leftmost

child is the next node to be traversed to. The fourth source bit of the packet (0) selects the only child of the node at this point, which is a leaf of the search structure. We then move to the port range search structure in Figure 7, starting from the top-left node labelled with R0 R1 R2. The source port of the packet (256) is found in the first interval of the source port search array. The destination port of the packet (6000) is found in the first (and only) interval of the destination port search array. At this point only the protocol needs to be compared in a linear search of an array of rules. We obtain a match for the protocol value 17, therefore the packet matches rule R0. The remaining rule category search structures i.e., those described in Section III-B besides the "long-long" one, would be searched to determine if there were a matching rule with a higher priority, although in this case we only have one search structure.

Consider now a packet with source address 10101 (in binary), destination address 11100 (in binary), source port 128, destination port 50 and protocol 7. Starting from the root node in Figure 5, the first three source bits of the packet (101) determine that the rightmost child is to be traversed to. The first bit of the destination address of the packet (1) does not correspond to a child of the current node, so the search fails to find a matching rule in Table I for the packet. The other rule category search structures would be searched to find a matching rule. Only if those searches all fail is there in fact no matching rule for the packet.

C. Update

The data structure that we propose lends itself well to on-line updates. Its modularity is such that when changes are made to either one node of the trie or a port range structure or a linear array, only that part of the structure needs to be locked temporarily to prevent the classifier reading partially updated data.

Deleting a rule does not present any difficulty. It is sufficient to search for the rule and delete it from the final rule sets to which it belongs. When inserting a rule, we first check the search path to determine where the rule should be inserted. If the search path for the addresses prefix matches does not exist, a new path will have to be created from the node where this new path diverges from old ones. In that case, some of the optimizations of Section IV-A.2 might have to be undone. If the path already exists, then we might have to create a new level in the trie if the number of rules in the final set exceeds the threshold, and/or modify the range structure.

Note that we have not addressed the priority factor in the insertion. At this point we assume that rules that are not of either lowest or highest priority compared to those that have the same source-destination address fields won't be inserted on-line. This is consistent with current practice.

V. Mapping to Microengines

In this section we describe the network processor architecture that we developed our algorithm for, the amount of memory used to hold the indices of the ACLs we studied and derive upper bounds on the worst-case time to perform packet classification.

A. Network Processor Architecture

The NP architecture we studied is a multiprocessor on a chip, with a control processor and multiple microengines. We use the control processor to build the index structures, while the microengines handle the classification process. The microengines have a limited amount of control store and a two level memory hierarchy, consisting of a small amount (say 8KB per microengine) of fast local storage (SRAM) and a larger (megabytes), slower off-chip memory (DRAM). We consider the best case where local memory is shared between all microengines, although decomposed index structures would allow for more limited sharing of local memory.

One possible arrangement of microengines to classify packets would be a pipeline, as shown in Figure 8. Each category of rules is processed by a separate microengine or group of microengines sharing the same data, except for the three categories of rules with wildcards in the source and/or destination addresses. Since there are comparatively few of these rules, these categories are handled together by a single microengine and the address prefix search is restricted to one dimension. Recall from Section IV-A.1 that rule priorities are stored in the classification structures so that when a rule match is found, the remaining structures do not have to be searched if they do not contain rules of higher priority. To maximize the benefit of this short-circuiting search, the pipeline stages are arranged so that rules which have longer source-destination address pairs and presumably higher priority are in earlier stages than rules with shorter source-destination address pairs and lower priority.

B. Memory Usage

An ACL uses around 20 bytes per rule; ACLs 1, 2 and 3 occupy 15KB, 12KB and 47KB of memory respectively. ACLs are stored in slow memory since there are relatively few accesses to them as compared to the indices.

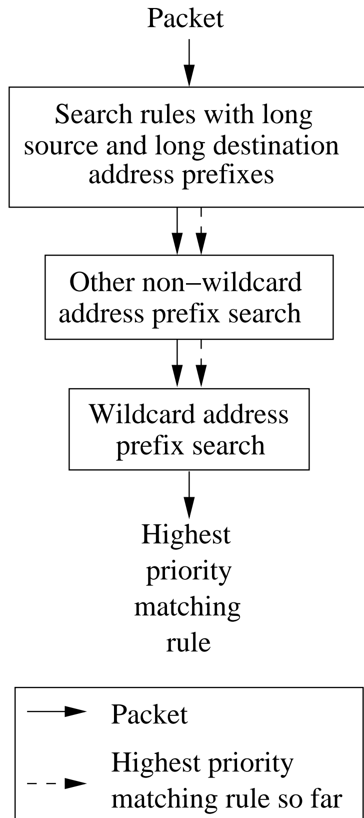

Fig. 8. Packet classification using a pipeline of microengines. Each category or groups of categories of rules is processed by a separate microengine. The other non-wildcard address prefix search stage encompasses all three long-short, short-long and short-short rule categories, since the data structures for these categories tend to be small and able to all fit into a single microengine fast memory. Likewise the wildcard address prefix stage encompasses the remaining rule categories.

The tuning parameter we use to control the amount of memory used to build the indices for the ACLs is the maximum size of a hash table in the address search structure. Larger hash table sizes lead to a wider and shorter address search structure at the expense of creating extra hash table entries from extending short address prefixes. The port interval arrays do not have direct tuning parameters, their sizes and numbers depend on the results of rules classified by the address search structure. We can control the maximum size for a candidate rule set but to keep searches efficient, this size should be on the order of 4 rules. Thus reasonable values for this tuning parameter have negligible effect on the memory used by the indices. The total memory used by the index for each rule category as the maximum hash table size varies from 8 to 128 is shown in Figures 9 to 11. Overall memory usage is shown in Table IV.

Although the port interval arrays represent only a small portion of the required storage, the binary search on them must be efficient, therefore they have highest priority for being stored in fast local storage. The address search structure has the next highest priority for fast memory. If it does not entirely fit, the lowest parts of the structure can be stored in slow memory because they are accessed less frequently than the top levels. In any remaining fast memory we store the candidate rule sets, otherwise we store these data structures in slow memory, since there are fewer accesses to these data structures.

The memory used by the index data structures for the source and destination address wildcard rules is unaffected by the maximum hash table size because an address search structure is not built for these rules. Instead the candidate rule matches are pruned using a source port interval array. Since the memory used by this category is both limited (under 200 bytes) and independent of the maximum hash table size, we exclude these results from the figures.

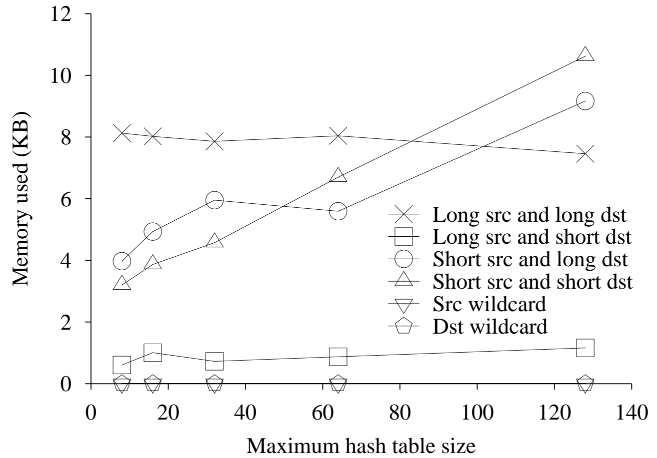

Fig. 9. ACL1 memory usage versus maximum hash table size.

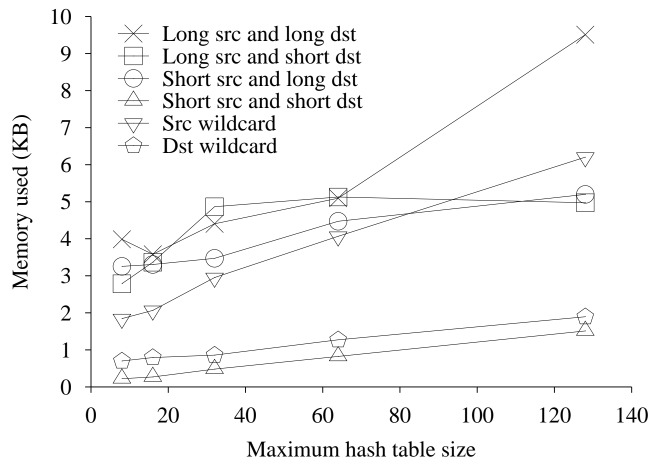

Fig. 10. ACL2 memory usage versus maximum hash table size.

We can see that separating rules into categories lets

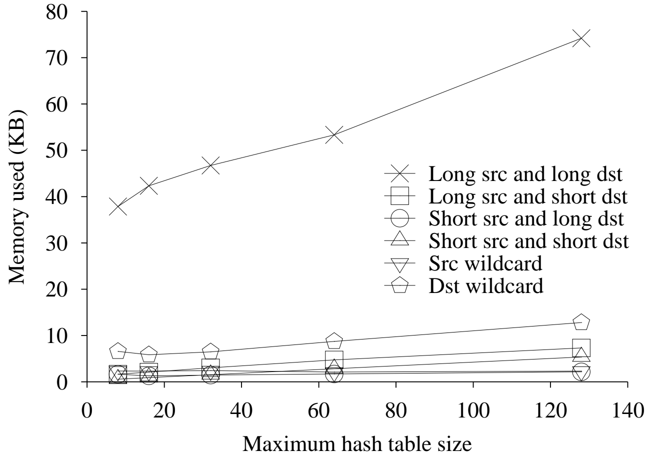

Fig. 11. ACL3 memory usage versus maximum hash table size.

TABLE IV

OVERALL MEMORY USAGE FOR ACL1, ACL2 AND ACL3 INDEX STRUCTURES. THE TOTALS ARE NOT BROKEN DOWN BY RULE CATEGORY.

	ACL1	ACL2	ACL3
Hash tables	8856	11560	41280
Port arrays	1716	478	6508
Candidate rule arrays	4548	3994	16514
Total	15120	16032	64302

us choose different maximum hash table sizes for each search structure, which enables greater flexibility in controlling memory usage. In ACL 1 we could choose a maximum hash table size of 128 for the long source and long destination address rule category and 64 for the rest of the categories.

Assume for the following discussion that we select a maximum hash table size of 64. There is no problem in fitting all of the indices for ACLs 1 and 2 into the fast memory of 3 microengines (24KB). However ACL3 requires on the order of 64KB of memory to store its indices. In particular the long source and long destination address classification index occupies about 50KB of memory and requires placing parts of the index into slower memory according to the priorities discussed earlier in this section. Storing parts of the index in slower memory has implications for worst-case performance, which we explain in the following section.

C. Bounds on Classification Time

The worst-case time required to classify an incoming packet is bounded by the number of memory references required in each pipeline stage in Figure 8. It is preferable to have 1 or 2 more fast memory accesses and use a smaller hash table size to reduces overall memory usage and avoid storing parts of the index in slow memory. Slow memory takes at least 5 times as long as fast memory to access, therefore the increased memory references is offset by the fact that these references are to fast memory. The aim is to balance the memory references performed in each pipeline stage, which are bounded by the sum of the worst-case address search structure memory references and the remaining memory references to the port arrays and candidate rule sets.

The worst-case address search structure memory references required for each ACL as the maximum hash table size is varied from 8 to 128 is shown in Figures 12 to 14. We choose a maximum hash table size which minimizes both the worst-case memory references and memory used. Very small hash tables (16 entries or less) tend to be fuller for a given fill factor, leading to worse performance because of greater numbers of hash table look-up collisions and thus memory references, although they are to fast memory. Our performance analysis might not work as well with small hash tables, since we use open addressing for our hash tables on the assumption that collisions are rare.

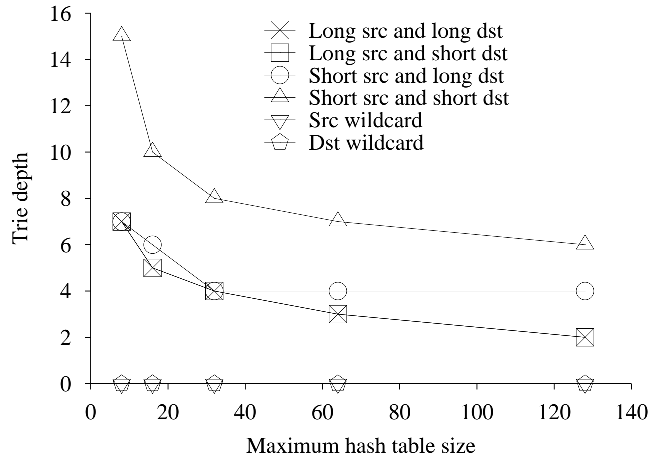

Fig. 12. ACL1 maximum trie depth versus maximum hash table size.

The worst-case number of memory references for the port array searches for ACLs 1 to 3 are shown in Table V. The candidate rule sets require in the worst-case 5 memory references to search. The port array and the candidate rule set search memory references are unaffected by the maximum hash table size tuning parameter.

The following worst-case memory reference analysis assumes a maximum hash table size of 64. The number of memory references ACL1 requires to search the indices are 14 for the long source and long destination addresses,

TABLE V

WORST-CASE NUMBER OF MEMORY REFERENCES FOR OUR ALGORITHM TO THE PORT ARRAYS FOR ACLS 1, 2 AND 3.

Rule category	ACL1	ACL2	ACL3
Long source and long destination	6	3	4
Long source and short destination	0	0	3
Short source and long destination	2	0	0
Short source and short destination	3	0	0
Source wildcard	0	0	2
Destination wildcard	0	0	6
Source and destination wildcards	0	4	5

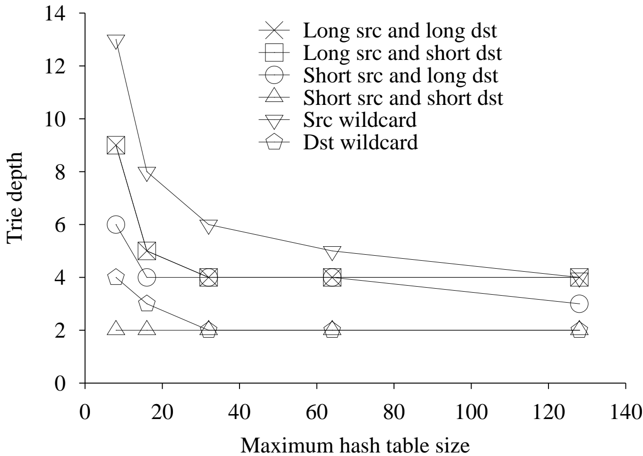

Fig. 13. ACL2 maximum trie depth versus maximum hash table size.

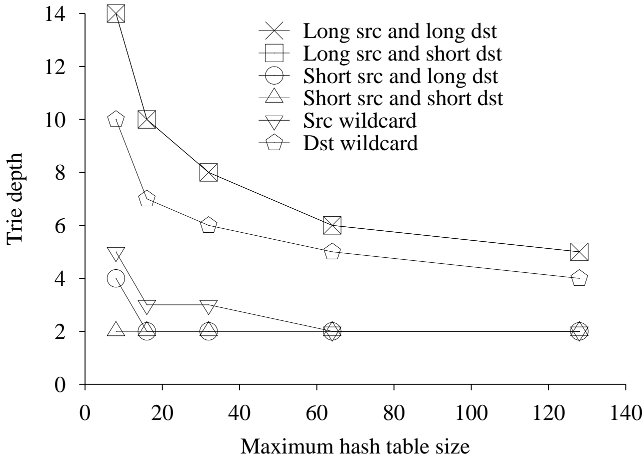

Fig. 14. ACL3 maximum trie depth versus maximum hash table size.

34 for the other non-wildcard addresses and 15 for the wildcard addresses. ACL2 requires 12, 25 and 26 memory references and ACL3 requires 15, 28 and 32 memory references, respectively.

As mentioned in the previous section, all the indices for ACLs 1 and 2 can easily fit into 24KB of fast memory. Since each individual index occupies at most 8KB of fast memory, one microengine can be assigned to a pipeline stage handling the processing of a single index. To balance the number of memory references made per pipeline stage, we would divide those stages which make greater numbers of memory references into smaller substages, and assign a separate microengine to each substage.

While ACL3 appears superficially to have the same order of worst-case memory references as ACLs 1 and 2, the first stage of the pipeline for ACL3 accesses a far larger index. If the fast memory is globally shared, the larger index can simply be stored using additional microengines to provide the required fast memory. If the fast memory is only locally accessible, the top levels of the address search structure of the index would be stored in fast memory and the lower levels would be stored in slow memory. Therefore about 3 of the 15 total memory accesses may be to slower memory. If slow memory takes on the order of 5 times as long to access as fast memory, these 15 memory accesses take time equivalent to 27 fast memory accesses. Therefore the pipeline stages are balanced in worst-case performance.

As an alternative to storing the lowest level of the address search structure in slow memory is to devote several microengines to the classification process. Each microengine is responsible for classifying a packet using only a portion of the index, the partitioning of the index and the required communication between microengines of partially classified packets is a topic that we are currently investigating.

Using a maximum hash table size of 32 for ACL3 would reduce overall memory used by about 7KB while incurring an additional 1 or 2 address search structure memory references for four of the seven rule categories. Going to a maximum hash table size of 16 to reduce memory usage further is not worthwhile due to the hash table performance degradation discussed earlier in this section.

In practice the deepest path in the address search structure does not align with the largest port arrays and candidate rule sets. Furthermore only an incoming packet which fails to match any rules induces worst-case behavior. Rule matches in early stages of the classification pipeline allow later searches to be skipped.

VI. Comparison with EGT

It is interesting to compare the memory requirements and the number of memory references that are achieved by our algorithm with some state of the art monolithic algorithm like EGT[1] Recall that our algorithm has for goals to generate modular data structures and to be easily pipelined. Thus even if the monolithic approach were superior in a single workstation environment, our algorithm could be more efficient in a network processor setting. As it turns out, on the ACLs that we tested, our algorithm is competitive with EGT (and EGT-PC) even in an environment for which it is not primarily intended.

Tables IV and VI show the breakdown of memory usage by our algorithm and EGT. The candidate rule lists in EGT are dynamically built during the search of the source and destination IP address fields and then the remaining fields are searched linearly. In contrast, once the destination and source IP address fields have been searched, our algorithm uses binary search of port arrays. It would be non-trivial to extend EGT to have this capability as it would require matching a dynamically generated and arbitrary length candidate rule list to a statically built search structure.

The memory footprint of EGT is approximately 6 times that of our algorithm and even with path compression, for which Baboescu et al. [10] report reduces memory used to around 1/3, EGT-PC still occupies twice the memory. Moreover our index data structures are decomposable and can be distributed between the memories of several micro-engines.

TABLE VI

Memory used by EGT index structures for ACL1 and ACL3.

	ACL1	ACL3
Trie nodes	142676	413720
Rule nodes	3016	9596
Total	145692	423316

[1]We are unable to run the original EGT code found in [10] on ACL2. We were not able either to generate the correct data structures using EGT-PC. We hope to fi x these problems by publication time.

We used ClassBench [14] to generate packet traces to evaluate the performance of EGT and our algorithm, which is shown in Tables VII and VIII. The packet traces for ACL1 and ACL3 contain 7548 and 67824 packets respectively. Even examining trie node references only, which gives an advantage to EGT due to the aforementioned port array search which our algorithm uses, EGT requires 5 times as many memory references for ACL1 and 2.5 times for ACL3 as our algorithm. Path compression can reduce the number of trie node references made by EGT by a factor of 2 to 3. Therefore as a first approximation our algorithm makes fewer trie node references than EGT with path compression on ACL1 and is on par with ACL3.

Examining non-trie references, which path compression does not reduce, on ACL1 EGT makes around 130,000 compared to 180,000 for our algorithm, and on ACL3, 2.2 million compared to 1 million. Our algorithm benefits significantly from being able to perform port array searches for ACL3. Also our algorithm is intended to be pipelined, therefore memory references can be made concurrently.

TABLE VII

Memory references made by EGT when classifying ACL1 and ACL3 packet traces.

	ACL1	ACL3
Trie	708810	4294358
Non-trie	132432	2208374
Candidate rule	80592	1311746
ACL	51840	896628
Total	841242	6502732

TABLE VIII

Memory references made by our algorithm when classifying ACL1 and ACL3 packet traces.

	ACL1	ACL3
Trie	132550	1833623
Non-trie	181073	1005782
Port array	8886	497392
Candidate rule	105176	327834
ACL	67011	180556
Total	313623	2839405

The previous paragraph discussed throughput, however many classification algorithms are geared to worst case performance.

On ACL1 the worst case number of memory references made by EGT for a packet is 168, of which 90 are to the trie. On ACL3 the values are 204 and 84 respectively. EGT-PC uses up to a factor of 3 less trie memory references. The worst case number of trie references for ACL1 would be reduced by 60 and for ACL3 by 56, thus the total worse case memory references could be reduced to around 108 and 148. These memory references determine the worst case processing time per packet.

On ACL1 the worst case number of memory references made by our algorithm for a packet is 71, while on ACL3 it is 97. This is better than what is achieved by EGT-PC. Furthermore our algorithm is pipelined and the memory references are made concurrently to private fast memory or to banked slower memory. Assuming that there is no buffering in the pipeline, the minimum delay between packets being sent into the processing pipeline must be at least the time taken by the longest stage. This time is less than the time taken to process a packet, which depends on the total number of memory references required, because processing on packets occurs concurrently. For ACL1 the packet with the worst case number of memory references is distinct to the packet which has the stage with the longest processing time. That particular packet uses 60 memory references in total, 34 of which occur in the longest stage, which is the one handling rules with long source and destination address prefixes. For ACL3 the packet with the worst case number of memory references is the same as the one with the longest processing stage, which is again the one handling long source and destination address prefixes. 53 of the 97 total memory references are made by the longest processing stage.

In summary, on limited testing, our algorithm performs better than EGT and as well as EGT-PC, both in terms of throughput and worse case memory accesses, when run in a workstation environment. It is superior both in performance relative to EGT-PC on a workstation and ease of mapping the index data structures to small local memory modules when run on network processors.

VII. Conclusion

Our packet classification algorithm is suitable for implementation on programmable network processors which have a multiprocessor on a chip architecture. Such an architecture contains a control processor, which we use to preprocess the data structures needed for packet classification, and several microengines, which we use to handle the various parts of the classification process. The microengines have a limited amount of control store and fast local storage, plus there is a slower global memory for additional storage.

The prefix matches used for source and destination address fields in a packet are treated differently from the matches performed on source and destination port and protocol fields. This approach is common to several other packet classification algorithms. We have two new contributions for address prefix matches. The first contribution is in our handling of rules with different prefix lengths and wildcards. Dividing rules into "long" prefixes, "short" prefixes and wildcards allows for significant memory savings. The second contribution is how we prune the set of possible rule matches by performing partial matches on source and destination address fields in alternation.

We show how our algorithm lends itself to mapping onto several microengines arranged in a processing pipeline. Each microengine accesses relatively small data structures which fit into fast local storage and accesses slow memory a bounded number of times. We derive this amount of local storage required as well as the timing of the search operation, in terms of the numbers of fast and slow memory accesses.

We compare our algorithm with EGT which is monolithic. Even though our algorithm is not intended for execution on a single processor, it has a smaller memory footprint than EGT and makes less memory references when classifying packets.

We are currently examining how larger rule sets impact the local data structures. In particular we need to determine how to decompose the address search structure and distribute it amongst the fast local memories of the microengines processing a pipeline stage.

References

[1] Vrini Srinivasan and George Varghese, "Fast address lookups using controlled prefix expansion," *ACM TOCS*, vol. 17, no. 1, pp. 1–40, Feb. 1999.

[2] "Differentiated services (diffserv)," http://www.ietf.org/html.charters/diffserv-charter.html, 2001.

[3] V. Kumar, T. Lakshman, and D. Stiliadis, "Beyond best effort: Architectures for the differentiated services of tomorrow's internet," *IEEE Communications Magazine*, vol. 36, no. 5, pp. 152–164, May 1998.

[4] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended tcams," in *Proc. of 11th IEEE ICPM'03*, 2003.

[5] Pankaj Gupta and Nick McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, Mar. 2001.

[6] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. of ACM SIGCOMM'98*, 1998, pp. 191–202.

[7] Anja Feldmann and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proc. INFOCOM 2000*, 2000, pp. 193–202.

[8] T. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. INFOCOM 2000*, 2000, pp. 1213–1222.

[9] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proc. of ACM SIGCOMM'98*, 1998, pp. 203–214.

[10] Florin Baboescu, Sumeet Singh, and George Varghese, "Packet classification for core routers: Is there an alternative to CAMs?," in *Proc. of INFOCOM 2003*, 2003.

[11] Florin Baboescu and George Varghese, "Scalable packet classification," in *Proc. of ACM SIGCOMM'01*, 2001, pp. 199–210.

[12] Michael E. Kounavis, Alok Kumar, Harrick Vin, Raj Yavatkar, and Andrew T. Campbell, "Directions in packet classification for network processors," in *Second Workshop on Network Processors (NP2)*, 2003.

[13] Donald Knuth, *The Art of Computer Programming. Vol 3, Searching and Sorting*, Addison-Wesley, Reading, Ma, 1973.

[14] David E. Taylor and Jonathan S. Turner, "Classbench: A packet classification benchmark," in *Proc. of IEEE INFOCOM 2005*, 2005.