Packet Classification on Multiple Fields

Pankaj Gupta and Nick McKeown
Computer Systems Laboratory, Stanford University
Stanford, CA 94305-9030
{pankaj, nickm}@stanford.edu†

Abstract

Routers classify packets to determine which flow they belong to, and to decide what service they should receive. Classification may, in general, be based on an arbitrary number of fields in the packet header. Performing classification quickly on an arbitrary number of fields is known to be difficult, and has poor worst-case performance. In this paper, we consider a number of classifiers taken from real networks. We find that the classifiers contain considerable structure and redundancy that can be exploited by the classification algorithm. In particular, we find that a simple multi-stage classification algorithm, called RFC (recursive flow classification), can classify 30 million packets per second in pipelined hardware, or one million packets per second in software.

1 Introduction

There are a number of network services that require packet classification, such as routing, access-control in firewalls, policy-based routing, provision of differentiated qualities of service, and traffic billing. In each case, it is necessary to determine which flow an arriving packet belongs to so as to determine — for example — whether to forward or filter it, where to forward it to, what class of service it should receive, or how much should be charged for transporting it. The categorization function is performed by a *flow classifier* (also called a packet classifier) which maintains a set of rules, where each flow obeys at least one rule. The rules classify which flow a packet belongs to based on the contents of the packet header(s). For example, a flow could be defined by particular values of source and destination IP addresses, and by particular transport port numbers. Or a flow could be simply defined by a destination prefix and a range of port values. As we shall see, a number of different types of rules are used in practice. This paper describes a method for fast packet classification based on an almost arbitrary set of rules. We focus here only on the problem of identifying the class to which a packet belongs. The actions taken for each class (e.g. packet scheduling in an output queue, routing decisions, billing [2][3][4][8][11][12][13]) while interesting in its own right, is not the topic of this paper.

The most well-known form of packet classification is used to route IP datagrams. In this case, all of the packets destined to the set of addresses described by a common prefix may be considered to be part of the same flow. Upon arrival to a router, the header of each packet is examined to determine the Network-layer destination address, which identifies the flow to which the packet belongs. Until recently, longest-prefix matching for routing lookups could not be done at high speeds. Now that several fast routing lookup algorithms have been developed (e.g. [1][5][7][9][16]), attention has turned to the more general problem of packet classification.

† This work was funded by the Center for Integrated Systems at Stanford University and the Alfred P. Sloan Foundation.

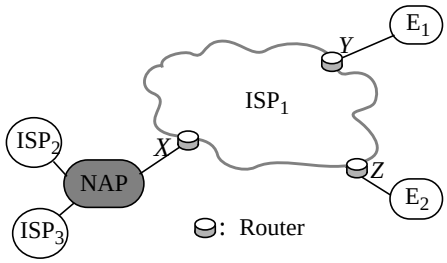


Figure 1: Example network of an ISP (ISP_1) connected to two enterprise networks (E_1 and E_2) and to two other ISP networks across a NAP.

To help illustrate the variety of packet classifiers, we start with some examples of how packet classification can be used by an Internet Service Provider (ISP) to provide different services. Figure 1 shows ISP_1 connected to three different sites: two enterprise networks E_1 and E_2 and a Network Access Point (NAP) which is in turn connected to ISP_2 and ISP_3. ISP_1 provides a number of different services to its customers, as shown in Table 1.

Table 1:

Service	Example
Packet Filtering	Deny all traffic from ISP_3 (on interface *X*) destined to E_2.
Policy Routing	Send all voice-over-IP traffic arriving from E_1 (on interface *Y*) and destined to E_2 via a separate ATM network.
Accounting & Billing	Treat all video traffic to E_1 (via interface *Y*) as highest priority and perform accounting for the traffic sent this way.
Traffic Rate Limiting	Ensure that ISP_2 does not inject more than 10Mbps of email traffic and 50Mbps of total traffic on interface *X*.
Traffic Shaping	Ensure that no more than 50Mbps of web traffic is injected into ISP_2 on interface *X*.

Table 2 shows the categories or classes that the router at interface *X* must classify an incoming packet into. Note that the classes specified may or may not be mutually exclusive (i.e. they may be overlapping or non-overlapping), for example the first and second class in Table 2 overlap. When this happens, we will follow the convention in which rules closer to the top of the list take priority, with the default rule appearing last.

Table 2:

Class	Relevant Packet Fields:
Email and from ISP_2	Source Link-layer Address, Source Transport port number
From ISP_2	Source Link-layer Address
From ISP_3 and going to E_2	Source Link-layer Address, Destination Network-Layer Address
All other packets	—

2 The Problem of Packet Classification

Packet classification is performed using a packet classifier, also called a policy database, flow classifier, or simply a classifier. A classifier is a collection of rules or policies. Each rule specifies a *class*[†] that a packet may belong to based on some criterion on F fields of the packet header, and associates with each class an identifier, *classID*. This identifier uniquely specifies the action associated with the rule. Each rule has F components. The i^{th} component of rule R, referred to as $R[i]$, is a regular expression on the i^{th} field of the packet header (in practice, the regular expression is limited by syntax to a simple address/mask or operator/number(s) specification). A packet P is said to *match* a particular rule R, if $\forall i$, the i^{th} field of the header of P satisfies the regular expression $R[i]$. It is convenient to think of a rule R as the ***set*** *of all packet headers which could match R*. When viewed in this way, two distinct rules are said to be either partially overlapping or non-overlapping, or that one is a subset of the other, with corresponding set-related definitions. We will assume throughout this paper that when two rules are not mutually exclusive, the order in which they appear in the classifier will determine their relative priority. For example, in a packet classifier that performs longest-prefix address lookups, each destination prefix is a rule, the corresponding next hop is its action, the pointer to the next-hop is the associated *classID*, and the classifier is the whole forwarding table. If we assume that the forwarding table has longer prefixes appearing before shorter ones, the lookup is an example of the packet classification problem.

2.1 Example of a Classifier

All examples used in this paper are classifiers from real ISP and enterprise networks. For privacy reasons, we have sanitized the IP addresses and other sensitive information so that the relative structure in the classifiers is still preserved.[‡] First, we'll take a look at the data and its characteristics. Then we'll use the data to evaluate the packet classification algorithm described later.

An example of some rules from a classifier is shown in Table 3. We collected 793 packet classifiers from 101 different ISP and enterprise networks and a total of 41,505 rules. Each network provided up to ten separate classifiers for different services.[††]

Table 3:

Network-layer Destination (addr/mask)	Network-layer Source (addr/mask)	Transport-layer Destination	Transport-layer Protocol
152.163.190.69/0.0.0.0	152.163.80.11/0.0.0.0	*	*
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	eq www	udp
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	range 20-21	udp
152.168.3.0/0.0.0.255	152.163.200.157/0.0.0.0	eq www	tcp
152.163.198.4/0.0.0.0	152.163.160.0/0.0.3.255	gt 1023	tcp
152.163.198.4/0.0.0.0	152.163.36.0/0.0.0.255	gt 1023	tcp

We found the classifiers to have the following characteristics:

1) The classifiers do not contain a large number of rules. Only 0.7% of the classifiers contain more than 1000 rules, with a mean of 50 rules. The distribution of number of rules in a classifier is shown in Figure 2. The relatively small number of rules per classifier should not come as a surprise: in most networks today, the rules are configured manually by network operators, and it is a non-trivial task to ensure correct behavior.
2) The syntax allows a maximum of 8 fields to be specified: source/destination Network-layer address (32-bits), source/destination Transport-layer port numbers (16-bits for TCP and UDP), Type-of-service (TOS) field (8-bits), Protocol field (8-

† For example, each rule in a flow classifier is a flow specification, where each flow is in a separate class.

‡ We wanted to preserve the properties of set relationship, e.g. inclusion, among the rules, or their fields. The way a 32-bit IP address *p0.p1.p2.p3* has been sanitized is as follows: (a) A random 32-bit number *c0.c1.c2.c3* is first chosen, (b) a random permutation of the 256 numbers 0...255 is then generated to get *perm[0..255]* (c) Another random number S between 0 and 255 is generated: these randomly generated numbers are common for all the rules in the classifier, (d) The IP address with bytes: *perm[(p0 ^ c0 + 0 * s) % 256], perm[(p1 ^ c1 + 1 * s) % 256], perm[(p2 ^ c2 + 2 * s) % 256]* and *perm[(p3 ^ c3 + 3 * s) % 256]* is then returned as the sanitized transformation of the original IP address, where ^ denotes the exclusive-or operation.

†† In the collected data, the classifiers for different services are made up of one or more ACLs (*access control lists*). An ACL rule has only two types of actions, "deny" or "permit". In this discussion, we will assume that each ACL is a separate classifier, a common case in practice.

bits), and Transport-Layer protocol flags (8-bits) with a total of 120 bits. 17% of all rules had 1 field specified, 23% had 3 fields specified and 60% had 4 fields specified.†

3) The Transport-layer protocol field is restricted to a small set of values: in all the packet classifiers we examined, it contained only TCP, UDP, ICMP, IGMP, (E)IGRP, GRE and IPINIP or the wildcard '*', i.e. the set of all transport protocols.
4) The Transport-layer fields have a wide variety of specifications. Many (10.2%) of them are *range* specifications (e.g. of the type *gt 1023*, i.e. greater than 1023, or *range 20-24*). In particular, the specification 'gt 1023' occurs in about 9% of the rules. This has an interesting consequence. It has been suggested in literature (for example in [17]) that to ease implementation, ranges could be represented by a series of prefixes. In that case, this common range would require six separate prefixes (1024-2047, 2048-4095, 4096-8191, 8192-16383, 16384-32767, 32768-65535) resulting in a large increase in the size of the classifier.
5) About 14% of all the classifiers had a rule with a non-contiguous mask (10.2% of all rules had non-contiguous masks). For example, a rule which has a Network-layer address/mask specification of 137.98.217.0/8.22.160.80 has a non-contiguous mask (i.e. *not* a simple prefix) in its specification. This observation came as a surprise. One suggested reason is that some network operators choose a specific numbering/addressing scheme for their routers. It tells us that a packet classification algorithm cannot always rely on Network-layer addresses being prefixes.
6) It is common for many different rules in the same classifier to share a number of field specifications (compare, for example, the entries in Table 3). These arise because a network operator frequently wants to specify the same policy for a pair of communicating groups of hosts or subnetworks (e.g. deny every host in group1 to access any host in group2). Hence, given a simple address/mask syntax specification, a separate rule is commonly written for each pair in the two (or more) groups. We will see later how we make use of this observation in our algorithm.
7) We found that 8% of the rules in the classifiers were redundant. We say that a rule R is redundant if one of the following condition holds:
(a) There exists a rule T appearing earlier than R in the classifier such that R is a subset of T. Thus, no packet will ever match R‡ and R is redundant. We call this *backward redundancy*; 4.4% of the rules were backward redundant.
(b) There exists a rule T appearing after R in the classifier such that (i) R is a subset of T, (ii) R and T have the same actions and (iii) For each rule V appearing in between R and T in the classifier, either V is disjoint from R, or V has the same action as R. We call this, *forward redundancy*; 3.6% of the rules were forward redundant. In this case, R can be eliminated to obtain a new smaller classifier. A packet matching R in the original classifier will match T in the new classifier, but with the same action.

† If a field is not specified, the wildcard specification is assumed. Note that this is affected by the syntax of the rule specification language.

‡ Recall that rules are prioritized in order of their appearance in the classifier.

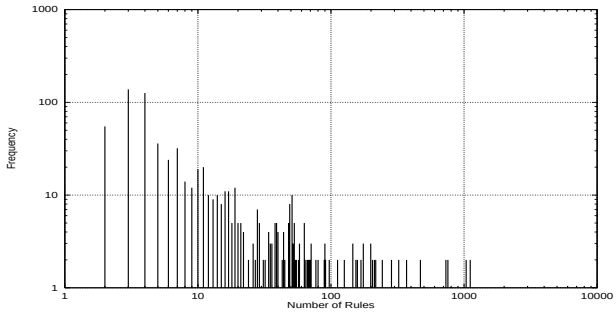


Figure 2: The distribution of the total number of rules per classifier. Note the logarithmic scale on both axes.

3 Goals

In this section, we highlight our objectives when designing a packet-classification scheme:

1) The algorithm should be fast enough to operate at OC48c linerates (2.5Gb/s) and preferably at OC192c linerates (10Gb/s). For applications requiring a deterministic classification time, we need to classify 7.8 million packets/s and 31.2 million packets/s respectively (assuming a minimum length IP datagram of 40 bytes). In some applications, an algorithm that performs well in the *average* case may be acceptable using a queue prior to the classification engine. For these applications, we need to classify 0.88 million packets/s and 3.53 million packets/s respectively (assuming an average Internet packet size of 354-bytes [18]).
2) The algorithm should ideally allow matching on arbitrary fields, including Link-layer, Network-layer, Transport-layer and — in some exceptional cases — the Application-layer headers.†† It makes sense for the algorithm to optimize for the commonly used header fields, but it should not preclude the use of other header fields.
3) The algorithm should support general classification rules, including prefixes, operators (like range, less than, greater than, equal to, etc.) and wildcards. Non-contiguous masks may be required.
4) The algorithm should be suitable for implementation in both hardware and software. Thus it needs to be fairly simple to allow high speed implementations. For the highest speeds (e.g. for OC192c at this time), we expect that hardware implementation will be necessary and therefore the scheme should be amenable to pipelined implementation.
5) Even though memory prices have continued to fall, the memory requirements of the algorithm should not be prohibitively expensive. Furthermore, when implemented in software, a memory-efficient algorithm can make use of caches. When implemented in hardware, the algorithm can benefit from faster on-chip memory.
6) The algorithm should scale in terms of both memory and

†† That is why packet-classifying routers have been called "layerless switches".

speed with the size of the classifier. An example from the destination routing lookup problem is the popular multiway trie [16] which can, in the worst case require enormous amounts of memory, but performs very well and with much smaller storage on real-life routing tables. We believe it to be important to evaluate algorithms with realistic data sets.

7) In this paper, we will assume that classifiers change infrequently (e.g. when a new classifier is manually added or at router boot time). When this assumption holds, an algorithm could employ reasonably static data structures. Thus, a preprocessing time of several seconds may be acceptable to calculate the data structures. Note that this assumption may not hold in some applications, such as when routing tables are changing frequently, or when fine-granularity flows are dynamically or automatically allocated.

Later, we describe a simple heuristic algorithm called RFC† (Recursive Flow Classification) that seems to work well with a selection of classifiers in use today. It appears practical to use the classification scheme for OC192c rates in hardware and up to OC48c rates in software. However, it runs into problems with space and preprocessing time for big classifiers (more than 6000 rules with 4 fields). We describe an optimization which decreases the storage requirements of the basic RFC scheme and enables it to handle a classifier with 15,000 rules with 4 fields in less than 4MB.

4 Previous Work

We start with the simplest classification algorithm: for each arriving packet, evaluate each rule sequentially until a rule is found that matches all the headers of the packet. While simple and efficient in its use of memory, this classifier clearly has poor scaling properties; time to perform a classification grows linearly with the number of rules.

A hardware-only algorithm could employ a ternary CAM (content-addressable memory). Ternary CAMs store words with three-valued digits: '0', '1' or 'X' (wildcard). The rules are stored in the CAM array in the order of decreasing priority. Given a packet-header to classify, the CAM performs a comparison against all of its entries in parallel, and a priority encoder selects the first matching rule. While simple and flexible, CAMs are currently suitable only for small tables; they are too expensive, too small and consume too much power for large classifiers. Futhermore, some operators are not directly supported, and so the memory array may be used very inefficiently. For example, the rule '*gt 1023*' requires six array entries to be used. But with continued improvements in semiconductor technology, large ternary CAMs may become viable in the future.

A solution called '*Grid of Tries*' was proposed in [17]. In this scheme, the trie data structure is extended to two fields. This is a good solution if the filters are restricted to only two fields, but is not easily extendible to more fields. In the same paper, a general solution for multiple fields called '*Crossproducting*' is described. It takes about 1.5MB for 50 rules and for bigger classifiers, the authors propose a caching technique (on-demand crossproducting) with a non-deterministic classification time.

† Not to be confused with "Request For Comments"

Another recent proposal [15] describes a scheme optimized for implementation in hardware. Employing bit-level parallelism to match multiple fields concurrently, the scheme is reported to support up to 512 rules, classifying one million packets per second with an FPGA device and five 1M-bit SRAMs. As described, the scheme examines five header fields in parallel and uses bit-level parallelism to complete the operation. In the basic scheme, the memory storage is found to scale quadratically and the memory bandwidth linearly with the size of the classifier. A variation is described that decreases the space requirement at the expense of higher execution time. In the same paper, the authors describe an algorithm for the special case of two fields with one field including only intervals created by prefixes. This takes *O(number-of-prefix-lengths + logn)* time and *O(n)* space for a classifier with *n* rules.

There are several standard problems in the field of computational geometry that resemble packet classification. One example is the point location problem in multidimensional space, i.e. the problem of finding the enclosing region of a point, given a set of regions. However, the regions are assumed to be non-overlapping (as opposed to our case). Even for non-overlapping regions, the best bounds for *n* rules and *F* fields, for $F > 3$, are *O(logn)* in time with $O(n^F)$ space; or $O(log^{F-1}n)$ time and *O(n)* space [6]. Clearly this is impractical: with just 100 rules and 4 fields, n^F space is about 100MB; and $log^{F-1}n$ time is about 350 memory accesses.

5 Proposed Algorithm RFC (Recursive Flow Classification)

5.1 Structure of the Classifiers

As the last example above illustrates, the task of classification is extremely complex in the worst case. However, we can expect there to be *structure* in the classifiers which, if exploited, can simplify the task of the classification algorithm.

To illustrate the structure we found in our dataset, let's start with an example in just two dimensions, as shown in Figure 3. We can represent a classifier with two fields (e.g. source and destination prefixes) in 2-dimensional space, where each rule is represented by a rectangle. Figure 3a shows three such rectangles, where each rectangle represents a rule with a range of values in each dimension. The classifier contains three explicitly defined rules, and the default (background) rule. Figure 3b shows how three rules can overlap to create five regions (each region is shaded differently), and Figure 3c shows three rules creating seven regions. A classification algorithm must keep a record of each region and be able to determine the region to which each newly arriving packet belongs. Intuitively, the more regions the classifier contains, the more storage is required, and the longer it takes to classify a packet.

Even though the number of rules is the same in each figure, the task of the classification algorithm becomes progressively harder as it needs to distinguish more regions. In general, it can be shown that the number of regions created by *n*-rules in *F* dimensions can be as much as $O(n^F)$.

We analyzed the structure in our dataset to determine the number of overlapping regions, and we found it to be considerably smaller than the worst case. Specifically for the biggest classifier with 1734 rules, we found the number of distinct overlapping regions in four

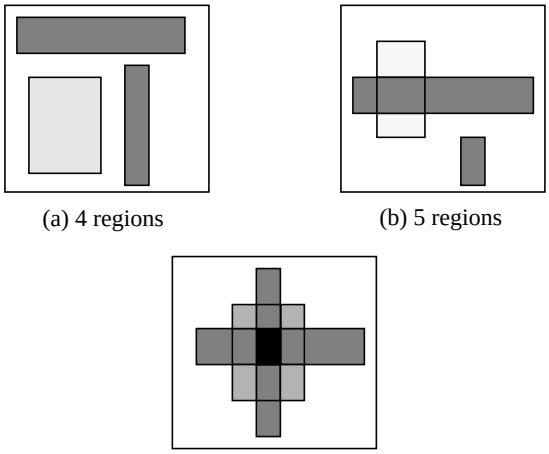


Figure 3: Some possible arrangements of three rectangles.

dimensions to be 4316, compared to a worst possible case of approximately 10^{13}. Similarly we found the number of overlaps to be relatively small in each of the classifiers we looked at. As we will see, our algorithm will exploit this structure to simplify its task.

5.2 Algorithm

The problem of packet classification can be viewed as one of mapping S bits in the packet header to T bits of *classID*, (where $T = \log N$, and $T \ll S$) in a manner dictated by the N classifier rules. A simple and fast (but unrealistic) way of doing this mapping might be to pre-compute the value of *classID* for each of the 2^S different packet headers. This would yield the answer in one step (one memory access) but would require too much memory. The main aim of *RFC* is to perform the same mapping but over several stages, as shown in Figure 4. The mapping is performed recursively; at each stage the algorithm performs a *reduction*, mapping one set of values to a smaller set.

The RFC algorithm has *P phases*, each phase consisting of a set of parallel memory lookups. Each lookup is a reduction in the sense that the value returned by the memory lookup is shorter (is expressed in fewer bits) than the index of the memory access. The algorithm, as illustrated in Figure 5, operates as follows:

1) In the first phase, *F* fields of the packet header are split up into multiple chunks that are used to index into multiple memories in parallel. For example, the number of chunks equals 8 in Figure 5; and Figure 6 shows an example of how the fields of a packet may be split across each memory. Each of the parallel lookups yields an output value that we will call *eqID*. (The reason for the identifier *eqID* will become clear shortly). The contents of each memory are chosen so that the result of the lookup is narrower than the index i.e. requires fewer bits.
2) In subsequent phases, the index into each memory is formed by combining the results of the lookups from earlier phases. For example, the results from the lookups may be concatenated; we will consider another way to combine them later.
3) In the final phase, we are left with one result from the lookup. Because of the way the memory contents have been pre-computed, this value corresponds to the *classID* of the packet.

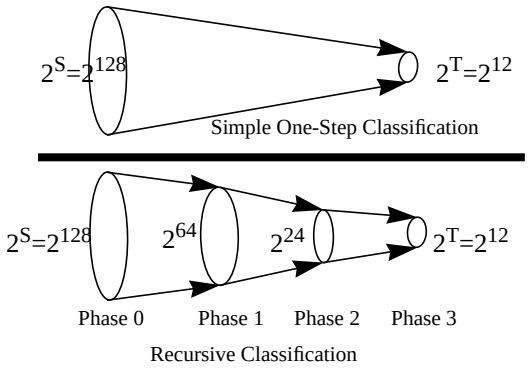


Figure 4: Showing the basic idea of Recursive Flow Classification. The reduction is carried out in multiple phases, with a reduction in phase *I* being carried out recursively on the image of the phase *I-1*. The example shows the mapping of 2^S bits to 2^T bits in 3 phases.

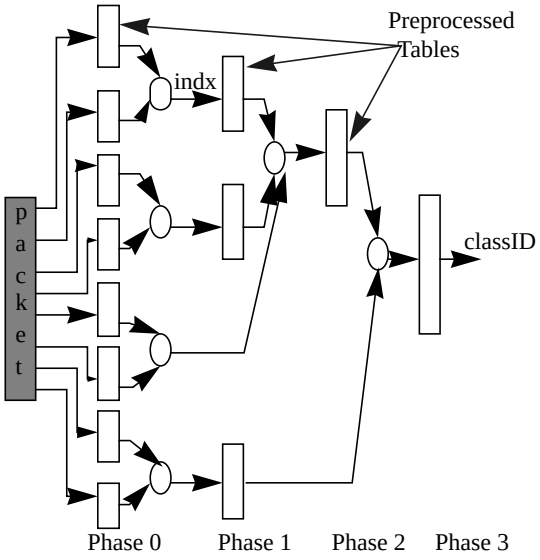


Figure 5: The packet flow in RFC.

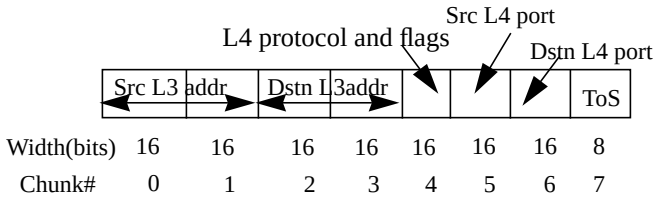


Figure 6: Example chopping of the packet header into chunks for the first RFC phase. L3 refers to Network-layer and L4 refers to Transport-layer fields

For the scheme to work, the contents of each memory are preprocessed. To illustrate how the memories are populated, we consider a simple example based on the classifier in Table 3. We'll see how the

24-bits used to express the Transport-layer Destination and Transport-layer Protocol (chunk #6 and #4 respectively) are reduced to just three bits by Phases 0 and 1 of the RFC algorithm. We start with chunk #6 that contains the 16-bit Transport-layer Destination. The corresponding column in Table 3 partitions the possible values into four sets: (a) {www=80} (b) {20,21} (c) {>1023} (d) {all remaining numbers in the range 0-65535}; which can be encoded using two bits 00_b through 11_b. We call these two bit values the "equivalence class IDs" (*eqIDs*). So, in Phase 0 of the RFC algorithm, the memory corresponding to chunk #6 is indexed using the 2^{16} different values of chunk #6. In each location, we place the *eqID* for this Transport-layer Destination. For example, the value in the memory corresponding to "chunk #6 = 20" is 00_b, corresponding to the set {20,21}. In this way, a 16-bit to two-bit reduction is obtained for chunk #6 in Phase 0. Similarly, the eight-bit Transport-layer Protocol column in Table 3 consists of three sets: (a) {tcp} (b) {udp} (c) {all remaining numbers in the range 0-255}, which can be encoded using two-bit *eqID*s. And so chunk #4 undergoes an eight-bit to two-bit reduction in Phase 0.

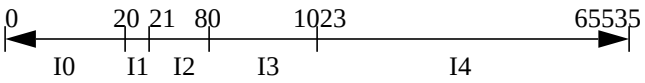
In the second phase, we consider the combination of the Transport-layer Destination and Protocol fields. From Table 3 we can see that the five sets are: (a) {({80}, {udp})} (b) {({20-21}, {udp})} (c) {({80}, {tcp})} (d) {({gt 1023}, {tcp})} (e) {all remaining crossproducts}; which can be represented using three-bit *eqID*s. The index into the memory in Phase 1 is constructed from the two two-bit *eqIDs* from Phase 0 (in this case, by concatenating them). Hence, in Phase 1 we have reduced the number of bits from four to three. If we now consider the combination of both Phase 0 and Phase 1, we find that 24 bits have been reduced to just three bits.

In what follows, we will use the term "Chunk Equivalence Set" (CES) to denote a set above, e.g. each of the three sets: (a) {tcp} (b) {udp} (c) {all remaining numbers in the range 0-255} is said to be a CES because if there are two packets with protocol values lying in the same set and have otherwise identical headers, the rules of the classifier do not distinguish between them.

Each CES can be constructed in the following manner:

First Phase (Phase 0): Consider a fixed chunk of size *b* bits, and those component(s) of the rules in the classifier corresponding to this chunk. Project the rules in the classifier on to the number line $\left[0, 2^b - 1\right]$. Each component projects to a set of (not necessarily contiguous) intervals on the number line. The end points of all the intervals projected by these components form a set of non-overlapping intervals. Two points in the same interval always belong to the same equivalence set. Also, two intervals are in the same equivalence set if exactly the same rules project onto them. As an example consider chunk #6 (Destination port) of the classifier in Table 3. The end-points of the intervals (*I0..I4*) and the constructed equivalence sets (*E0..E3*) are shown in Figure 7. The RFC table for this chunk is filled with the corresponding *eqID*s. Thus, in this example, *table(20)* = 00_b, *table(23)* = 11_b etc. The pseudocode for computing the *eqID*s in Phase 0 is shown in Figure 19 in the Appendix.

To facilitate the calculation of the *eqIDs* for subsequent RFC phases, we assign a class bitmap (CBM) for each CES indicating



E0 = {20,21} E2 = {1024-65535}
E1 = {80} E3 = {0-19,22-79,81-1023}

Figure 7: An example of computing the four equivalence classes E0...E3 for chunk #6 (corresponding to the 16-bit Transport-Layer destination port number) in the classifier of Table 3.

which rules in the classifier contain this CES for the corresponding chunk. This bitmap has one bit for each rule in the classifier. For example, E0 in Figure 7 will have the CBM 101000_b indicating that the first and the third rules of the classifier in Table 3 contain E0 in chunk #6. Note that the class bitmap is *not* physically stored in the lookup table: it is just used to facilitate the calculation of the stored *eqIDs* by the preprocessing algorithm.

Subsequent Phases: A chunk in a subsequent phase is formed by a combination of two (or more) chunks obtained from memory lookups in previous phases, with a corresponding CES. If, for example, the resulting chunk is of width *b* bits, we again create equivalence sets such that two *b*-bit numbers that are not distinguished by the rules of the classifier belong to the same CES. Thus, (20,udp) and (21,udp) will be in the same CES in the classifier of Table 3 in Phase 1. To determine the new equivalence sets for this phase, we compute all possible intersections of the equivalence sets from the previous phases being combined. Each distinct intersection is an equivalence set for the newly created chunk. The pseudocode for this preprocessing is shown in Figure 20 of the Appendix.

5.3 A simple complete example of RFC

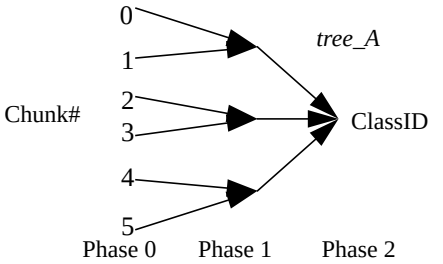
Realizing that the preprocessing steps are involved, we present a complete example of a classifier, showing how the RFC preprocessing is performed to determine the contents of the memories, and how a packet can be looked up as part of the RFC operation. The example is shown in Figure 22 in the Appendix. It is based on a 4-field classifier of Table 6, also in the Appendix.

6 Implementation Results

In this section, we consider how the RFC algorithm can be implemented and how it performs. First, we consider the complexity of preprocessing and the resulting storage requirements. Then we consider the lookup performance to determine the rate at which packets can be classified.

6.1 RFC Preprocessing

With our classifiers, we choose to split the 32-bit Network-layer source and destination address fields into two 16-bit chunks each. These are chunks #0,1 and #2,3 respectively. As we found a maximum of four fields in our classifiers, this means that Phase 0 of RFC has six chunks: chunk #4 corresponds to the protocol and protocol-flags field and chunk #5 corresponds to the Transport-layer



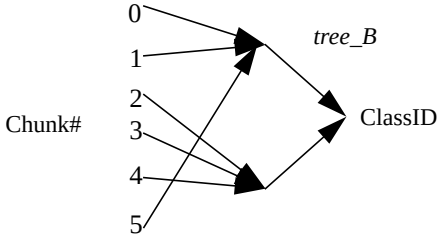
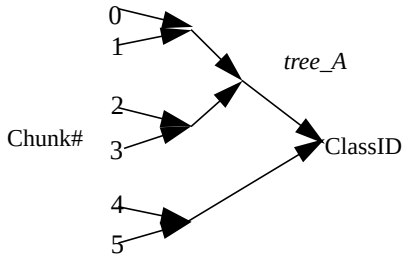


Figure 8: Two example reduction trees for P=3 RFC phases.



Phase 0 Phase 1 Phase 2 Phase 3

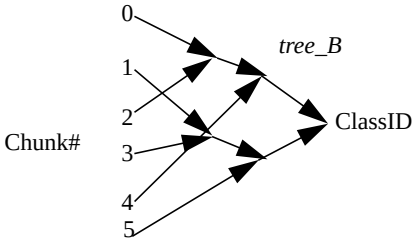


Figure 9: Two example reduction trees for P=4 RFC phases.

Destination field.

The performance of RFC can be tuned with two parameters: (i) The number of phases, *P*, that we choose to use, and (ii) Given a value of *P*, the reduction tree used. For instance, two of the several possible reduction trees for P=3 and P=4 are shown in Figure 8 and Figure 9 respectively. (For P=2, there is only one reduction tree possible). When there is more than one reduction tree possible for a given value of P, we choose a tree based on two heuristics: (i) we combine those chunks together which have the most "correlation" e.g. we combine the two 16-bit chunks of Network-layer source address in the earliest phase possible, and (ii) we combine as many chunks as we can without causing unreasonable memory consumption. Following these heuristics, we find that the "best" reduction tree for P=3 is tree_B in Figure 8, and the "best" reduction tree for P=4 is tree_A in Figure 9.

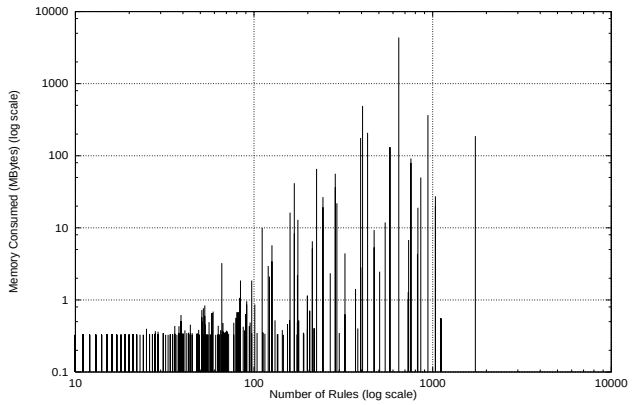


Figure 10: The RFC storage requirements in Megabytes for two Phases using the classifiers available to us. This special case of RFC is identical to the Crossproducting method of [17].

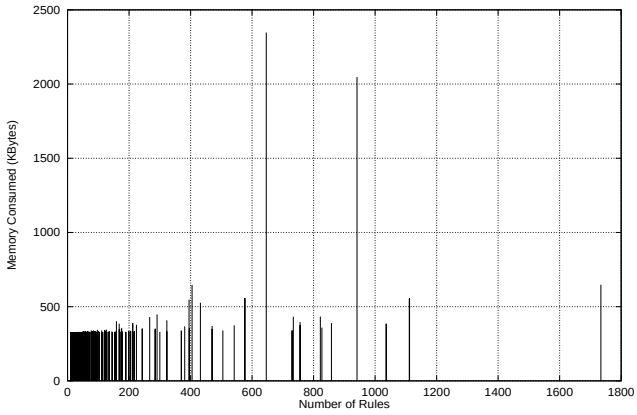


Figure 11: The RFC storage requirements in Kilobytes for three phases using the classifiers available to us. The reduction tree used is *tree_B* in Figure 8.

Now, let us look at the performance of RFC using our set of classifiers. Our first goal is to keep the total amount of memory reasonably small. The memory requirements for each of our classifiers is plotted in Figure 10, Figure 11 and Figure 12 for P=2, 3 and 4 phases respectively. The graphs show how the memory usage increases with the number of rules in each classifier. For practical purposes, it is assumed that memory is only available in widths of 8, 12 or 16 bits. Hence, an *eqID* requiring 13 bits is assumed to occupy 16 bits in the RFC table.

As we might expect, the graphs show that as we increase the number of phases from three to four, we require a smaller total amount of memory. However, this comes at the expense of two additional memory accesses, illustrating the trade-off between memory consumption and lookup time in RFC. Our second goal is to keep the preprocessing time small. And so in Figure 13 we plot the preprocessing time required for both three and four phases of RFC.†

The graphs indicate that, for these classifiers, RFC is suitable if

† The case P=2 is not plotted: it was found to take hours of preprocessing time because of the unwieldy size of the RFC tables.

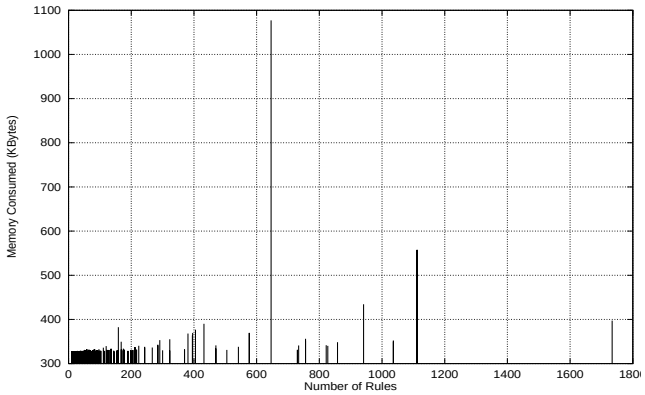


Figure 12: The RFC storage requirements in Kilobytes for four phases using the classifiers available to us. The reduction tree used is *tree_A* in Figure 9.

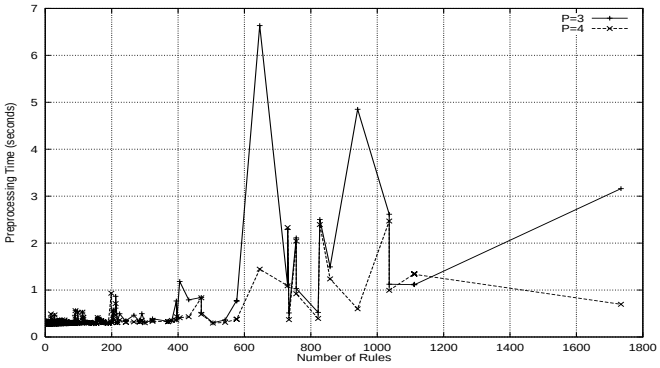


Figure 13: The preprocessing times for three and four phases in seconds, using the set of classifiers available to us. This data is taken by running the RFC preprocessing code on a 333MHz Pentium-II PC running the Linux operating system.

(and only if) the rules change relatively slowly; for example, not more than once every few seconds. Thus, it may be suitable in environments where rules are changed infrequently, for example if they are added manually, or when a router reboots.

For applications where the tables change more frequently, it may be possible to make incremental changes to the tables. This is a subject requiring further investigation.

Finally, note that there are some similarities between the RFC algorithm and the bit-level parallel scheme in [15]; each distinct bitmap in [15] corresponds to a CES in the RFC algorithm. Also, note that when there are just two phases, RFC corresponds to the crossproducting method described in [17].

6.2 RFC Lookup Performance

The RFC lookup operation can be performed both in hardware and in software.† We will discuss the two cases separately, exploring the

† Note that the RFC preprocessing is always performed in software.

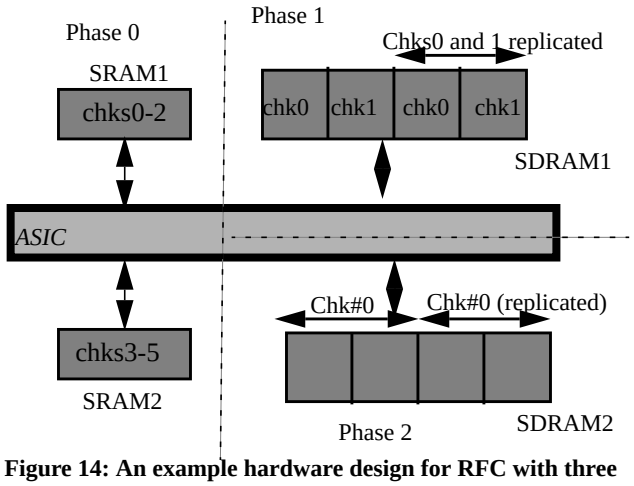


Figure 14: An example hardware design for RFC with three phases. The latches for holding data in the pipeline and the on-chip control logic are not shown.This design achieves OC192 rates in the worst case for 40Byte packets, assuming that the phases are pipelined with 4 clock cycles (at 125MHz clock rate) per pipeline stage.

lookup performance in each case.

Hardware

An example hardware implementation for the tree *tree_B* in Figure 8 (three phases) is illustrated in Figure 14 for four fields (six chunks in Phase 0). This design is suitable for all the classifiers in our dataset, and uses two 4Mbit SRAMs and two 4-bank 64Mbit SDRAMs [19] clocked at 125 MHz.‡ The design is pipelined such that a new lookup may begin every four clock cycles. The pipelined RFC lookup proceeds as follows:

1) **Pipeline Stage 0: Phase 0** *(Clock cycles 0-3)*: In the first three clock cycles, three accesses are made to the two SRAM devices in parallel to yield the six *eqID*s of Phase 0. In the fourth clock cycle, the *eqID*s from Phase 0 are combined to compute the two indices for the next phase.
2) **Pipeline Stage 1: Phase 1***(Clock cycles 4-7)*: The SDRAM devices can be accessed every two clock cycles, but we assume that a given bank can be accessed again only after eight clock cycles. By keeping the two memories for Phase 1 in different banks of the SDRAM, we can perform the Phase 1 lookups in four clock cycles. The data is replicated in the other two banks (i.e. two banks of memory hold a fully redundant copy of the lookup tables for Phase 1). This allows Phase 1 lookups to be performed on the next packet as soon as the current packet has completed. In this way, any given bank is accessed once every eight clock cycles.
3) **Pipeline Stage 2: Phase 2** *(Clock cycles 8-11)*: Only one lookup is to be made. The operation is otherwise identical to Phase 1.

Hence, we can see that approximately 30 million packets can be

‡ These devices are in production in industry at the time of writing this paper. In fact, even bigger and faster devices are available today - see [19]

classified per second (to be exact, 31.25 million packets per second with a 125MHz clock) with a hardware cost of approximately $50.† This is fast enough to process minimum length packets at the OC192c rate.

Software

Figure 21 (Appendix) provides pseudocode to perform RFC lookups. When written in 'C', approximately 30 lines of code are required to implement RFC. When compiled on a 333Mhz Pentium-II PC running Windows NT we found that the worst case path for the code took $(140clks + 9 \cdot t_m)$ for three phases, and $(146clks + 11 \cdot t_m)$ for four phases, where t_m is the memory access time.‡ With $t_m = 60ns$, this corresponds to $0.98\mu s$ and $1.1\mu s$ for three and four phases respectively. This implies RFC can perform close to one million packets per second in the worst case for our classifiers. The average lookup time was found to be approximately 50% faster than the worst case; Table 4 shows the average time taken per lookup for 100,000 randomly generated packets for some classifiers.

Table 4:

Number of Rules in Classifier	Average Time per lookup (ns)
39	587
113	582
646	668
827	611
1112	733
1734	621

The pseudocode in Figure 21 calculates the indices into each memory using multiplication/addition operations on *eqIDs* from previous phases. Alternatively, the indices can be computed by simple concatenation. This has the effect of increasing the memory consumed because the tables are then not as tightly packed. Given the simpler processing, we might expect the classification time to decrease at the expense of increased memory usage. Indeed the memory consumed grows approximately two-fold on the classifiers. Surprisingly, we saw no significant reduction in classification times. We believe that this is because the processing time is dominated by memory access time as opposed to the CPU cycle time.

† Under the assumption that SDRAMs are now available at $1.50 per megabyte, and SRAMs are $12 for a 4Mbyte device.

‡ The performance of the lookup code was analyzed using VTune[20], an Intel performance analyzer for processors of the Pentium family.

6.3 Larger Classifiers

As we have seen, RFC performs well on the real-life classifiers available to us. But how will RFC perform with larger classifiers that might appear in the future? Unfortunately, it is difficult to accurately predict the memory consumption of RFC as a function of the size of the classifier: the performance of RFC is determined by the structure present in the classifier. With pathological sets of rules, RFC could scale geometrically with the number of rules. Fortunately, such cases do not seem to appear in practice.

To estimate how RFC might perform with future, larger classifiers, we synthesized large artificial classifiers. We used two different ways to create large classifiers (given the importance of the structure, it did not seem meaningful to generate rules randomly):

1) A large classifier can be created by concatenating the classifiers belonging to the same network, and treating the result as a single classifier. Effectively, this means merging together the individual classifiers for different services. Such an implementation is actually desirable in scenarios where the designer may not want more than one set of RFC tables for the whole network. In such cases, the *classID* obtained would have to be combined with some other information (such as classifier ID) to obtain the correct intended action. By only concatenating classifiers from the same network, we were able to create classifiers up to 3,896 rules. For each classifier created, we performed RFC with both three and four phases. The results are shown in Figure 15.
2) To create even larger classifiers, we concatenated all the classifiers of a few (up to ten) different networks. The performance of RFC with four phases is plotted as the 'Basic RFC' curve in Figure 18. We found that RFC frequently runs into storage problems for classifiers with more than 6000 rules. Employing more phases does not help as we must combine at least two chunks in every phase, and end up with one chunk in the final phase.†† An alternative way to process large classifiers would be to split them into two (or more) parts and construct separate RFC tables for each part. This would of course come at the expense of doubling the number of memory accesses.‡‡

7 Variations

Several variations and improvements of RFC are possible. First, it should be easy to see how RFC can be extended to process a larger number of fields in each packet header.

Second, we can possibly speed up RFC by taking advantage of available fast lookup algorithms that find longest matching prefixes in one field. Note that in our examples, we use three memory accesses each for the source and destination Network-layer address

†† With six chunks in Phase 0, we could have increased the number of phases to a maximum of six. However we found no appreciable improvement by doing so.

‡‡ Actually, for Phase 0, we need not lookup memory twice for the same chunk if we use wide memories. This would help us access the contents of both the RFC tables in one memory access.

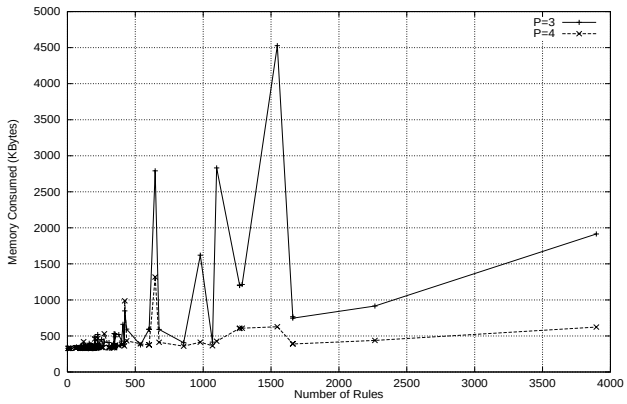


Figure 15: The memory consumed by RFC for three and four phases on classifiers created by merging all the classifiers of one network.

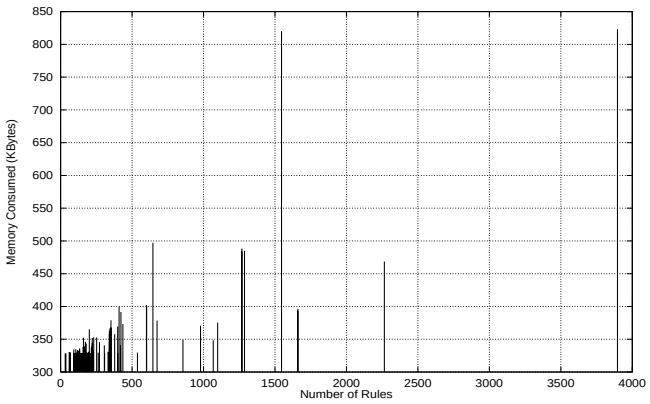


Figure 16: The memory consumed with three phases with the adjGrp optimization enabled on the large classifiers created by concatenating all the classifiers of one network

lookups during the first two phases of RFC. This is necessary because of the considerable number of non-contiguous address/ mask specifications. In the event that only prefixes are present in the specification, one can use a more sophisticated and faster technique for looking up in one dimension e.g. one of the methods described in [1][5][7][9] or [16].

Third, we can employ a technique described below to reduce the memory requirements when processing large classifiers.

7.1 Adjacency Groups

Since the size of the RFC tables depends on the number of chunk equivalence classes, we focus our efforts on trying to reduce this number. This we do by merging two or more rules of the original classifier as explained below. We find that each additional phase of RFC further increases the amount of compaction possible on the original classifier.

First we define some notation. We call two distinct rules *R* and *S*, with *R* appearing first, in the classifier to be *adjacent in dimension 'i'* if all of the following three conditions hold: (1) They have the same action, (2) All but the i^{th} field have the exact same specification in the two rules, and (3) All rules appearing in between *R* and *S* in the classifier have either the same action or are disjoint from *R*. Two rules are said to be simply *adjacent* if they are adjacent in some dimension. Thus the second and third rules in the classifier of Table 3 are adjacent† in the dimension corresponding to the Transport-layer Destination field. Similarly the fifth rule is adjacent to the sixth but not to the fourth. Once we have determined that two rules *R* and *S* are adjacent, we merge them to form a new rule *T* with the same action as *R* (or *S*). It has the same specifications as that of *R* (or *S*) for all the fields except that of the i^{th} which is simply the *logical-OR* of the i^{th} field specifications of *R* and *S*. The third condition above ensures that the relative priority of the rules in between *R* and *S* will not be affected by this merging.

An *adjacency group* (adjGrp) is defined recursively as: (1) Every rule in the original classifier is an adjacency group, and (2) Every merged rule which is created by merging two or more adjacency groups is an adjacency group.

We compact the classifier as follows. Initially, every rule is in its own adjGrp. Next, we combine adjacent rules to create a new smaller classifier. One simple way of doing this is to iterate over all fields in turn, checking for adjacency in each dimension. After these iterations are completed, the resulting classifier will have no more adjacent rules. We do similar merging of adjacent rules after each RFC phase. As each RFC phase collapses some dimensions, groups which were not adjacent in earlier phases may become so in later stages. In this way, the number of adjGrps and hence the size of the classifier keeps on decreasing with every phase. An example of this operation is shown in Figure 17.

Note that there is absolutely no change in the actual lookup operation: the equivalence IDs are now simply pointers to bitmaps which keep track of adjacency groups rather than the original rules. To demonstrate the benefits of this optimization for both three and four phases, we plot in Figure 16 the memory consumed with three phases on the 101 large classifiers created by concatenating all the classifiers belonging to one network; and in Figure 18, the memory consumed with four phases on the even larger classifiers created by concatenating all the classifiers of different networks together. The figures show that this optimization helps reduce storage requirements. With this optimization, RFC can now handle a 15,000 rule classifier with just 3.85MB. The reason for the reduction in storage is that several rules in the same classifier commonly share a number of specifications for many fields.

However, the space savings come at a cost. For although the classifier will correctly identify the action for each arriving packet, it cannot tell which rule in the original classifier it matched. Because the rules have been merged to form adjGrps, the distinction between each rule has been lost. This may be undesirable in applications that maintain matching statistics for each rule.

† Adjacency can be also be looked at this way: treat each rule with F fields as a boolean expression of F (multi-valued) variables. Initially each rule is a conjunction i.e. a logical-AND of these variables. Two rules are defined to be adjacent of they are adjacent vertices in the F-dimensional hypercube created by the symbolic representation of the F fields.

Table 5:

Scheme	Pros	Cons
Sequential Evaluation	Good storage requirements. Works for arbitrary number of fields.	Slow lookup rates.
Grid of Tries[17]	Good storage requirements and fast lookup rates for two fields. Suitable for big classifiers.	Not easily extendible to more than two fields. Not suitable for non-contiguous masks.
Crossproducting[17]	Fast accesses. Suitable for multiple fields. Can be adapted to non-contiguous masks.	Large memory requirements. Suitable without caching for small classifiers up to 50 rules.
Bit-level Parallelism[15]	Suitable for multiple fields. Can be adapted to non-contiguous masks.	Large memory bandwidth required. Comparatively slow lookup rate. Hardware only.
Recursive Flow Classification	Suitable for multiple fields. Works for non-contiguous masks. Reasonable memory requirements for real-life classifiers. Fast lookup rate.	Large preprocessing time and memory requirements for large classifiers (i.e. having more than 6000 rules without adjacency group optimization).

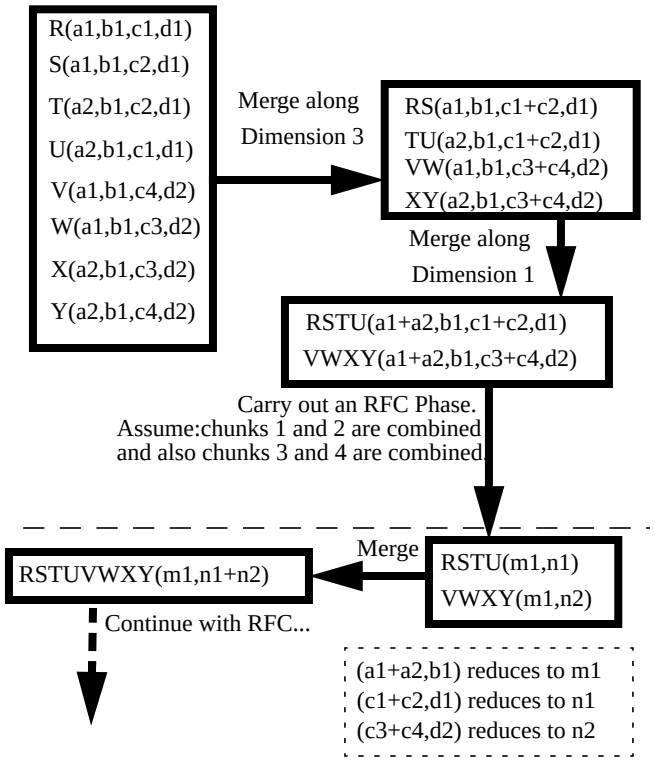


Figure 17: Example of Adjacency Groups. Some rules of a classifier are shown. Each rule is denoted symbolically by *RuleName(FieldName1, FieldName2,...)*. The '+' denotes a logical OR. All rules shown are assumed to have the same action.

8 Comparison with other packet classification schemes

Table 5 shows a qualitative comparison of some of the schemes for doing packet classification.

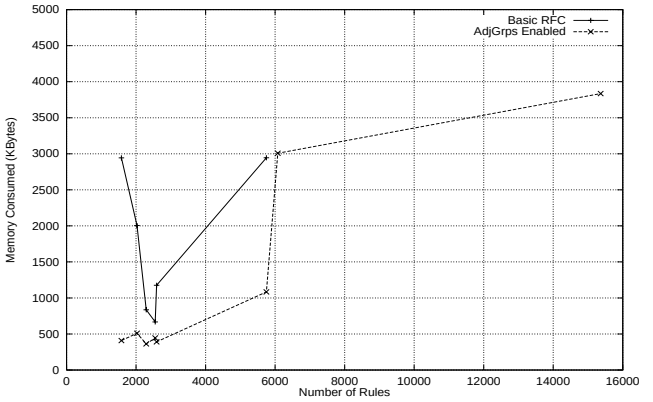


Figure 18: The memory consumed with four phases with the adjGrp optimization enabled on the large classifiers created by concatenating all the classifiers of a few different networks. Also shown is the memory consumed when the optimization is not enabled (i.e. the basic RFC) scheme. Notice the absence of some points in the Basic RFC curve. For those classifiers, the basic RFC takes too much memory/preprocessing time.

9 Conclusions

It is relatively simple to perform packet classification at high speed using large amounts of storage; or at low speed with small amounts of storage. When matching multiple fields (dimensions) simultaneously, it is difficult to achieve both high classification rate and modest storage in the worst case. We have found that real classifiers (today) exhibit considerable amount of structure and redundancy. This makes possible simple classification schemes that exploit the structure inherent in the classifier. We have presented one such algorithm, called RFC which appears to perform well with the selection of real-life classifiers available to us. For applications in which the tables do not change frequently (for example, not more than once every few seconds) a custom hardware implementation can achieve OC192c rates with a memory cost of less than $50, and a software implementation can achieve OC48c rates. RFC was found to consume too much storage for classifiers with four fields

and more than 6,000 rules. But by further exploiting structure and redundancy in the classifiers, a modified version of RFC appears to be practical for up to 15,000 rules.

10 Acknowledgments

We would like to thank Darren Kerr at Cisco Systems for help in providing the classifiers used in this paper. We also wish to thank Andrew McRae at Cisco Systems for independently suggesting the use of bitmaps in storing the colliding rule set, and for useful feedback about the RFC algorithm.

11 References

[1] A. Brodnik, S. Carlsson, M. Degermark, S. Pink. “Small Forwarding Tables for Fast Routing Lookups,” Proc. ACM SIGCOMM 1997, pp. 3-14, Cannes, France.

[2] Abhay K. Parekh and Robert G. Gallager, “A generalized processor sharing approach to flow control in integrated services networks: The single node case,” IEEE/ACM Transactions on Networking, vol. 1, pp. 344-357, June 1993.

[3] Alan Demers, Srinivasan Keshav, and Scott Shenker, “Analysis and simulation of a fair queueing algorithm,” Internetworking: Research and Experience, vol. 1, pp. 3-26, January 1990.

[4] Braden et al., “Resource ReSerVation Protocol (RSVP) - Version 1 Functional Specification,” RFC 2205, September 1997.

[5] B. Lampson, V. Srinivasan, and G. Varghese, "IP lookups using multiway and multicolumn search," in Proceedings of the Conference on Computer Communications (IEEE INFOCOMM), (San Francisco, California), vol. 3, pp. 1248-1256, March/April 1998.

[6] M.H. Overmars and A.F. van der Stappen, “Range searching and point location among fat objects,” in Journal of Algorithms, 21(3), pp. 629-656, 1996.

[7] M. Waldvogel, G. Varghese, J. Turner, B. Plattner. “Scalable High-Speed IP Routing Lookups,” Proc. ACM SIGCOMM 1997, pp. 25-36, Cannes, France.

[8] M. Shreedhar and G. Varghese, “Efficient Fair Queuing Using Deficit Round-Robin,” IEEE/ACM Transactions on Networking, vol. 4,3, pp. 375-385, 1996.

[9] P. Gupta, S. Lin, and N. McKeown, “Routing lookups in hardware at memory access speeds,” in Proceedings of the Conference on Computer Communications (IEEE INFOCOMM), (San Francisco, California), vol. 3, pp. 1241-1248, March/April 1998.

[10] P. Newman, T. Lyon and G.Minshall, “Flow labelled IP: a connectionless approach to ATM”, Proceedings of the Conference on Computer Communications (IEEE INFOCOMM), (San Francisco, California), vol. 3, pp 1251-1260, April 1996.

[11] R. Guerin, D. Williams, T. Przygienda, S. Kamat, and A. Orda, “QoS routing mechanisms and OSPF extensions,” Internet Draft, Internet Engineering Task Force, March 1998. Work in progress.

[12] Richard Edell, Nick McKeown, and Pravin Varaiya, “Billing Users and Pricing for TCP”, IEEE JSAC Special Issue on Advances in the Fundamentals of Networking, September 1995.

[13] Sally Floyd and Van Jacobson, “Random early detection gateways for congestion avoidance,” IEEE/ACM Transactions on Networking, vol. 1, pp. 397-413, August 1993.

[14] Steven McCanne and Van Jacobson, “A BSD Packet Filter: A New Architecture for User-level Packet Capture,” in Proc. of Usenix Winter Conference, (San Diego, California), pp. 259-269, Usenix, January 1993.

[15] T.V. Lakshman and D. Stiliadis, “High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching”, Proc. ACM SIGCOMM, pp. 191-202, September1998.

[16] V.Srinivasan and G.Varghese, “Fast IP Lookups using Controlled Prefix Expansion”, in Proc. ACM Sigmetrics, June 1998.

[17] V.Srinivasan, S.Suri, G.Varghese and M.Waldvogel, “Fast and Scalable Layer4 Switching”, in Proc. ACM SIGCOMM, pp. 203-214, September 1998.

[18] http://www.nlanr.net/NA/Learn/plen.970625.hist.

[19] http://www.toshiba.com/taec/nonflash/components/memory.html.

[20] http://developer.intel.com/vtune/analyzer/index.htm.

12 Appendix

```
/* Begin Pseudocode */
/* Phase 0, Chunk j of width b bits*/
for each rule rl in the classifier
begin
   project the i^th component of rl onto the number line (from 0 to 2^b-1),
   marking the start and end points of each of its constituent intervals.
endfor
/* Now scan through the number line looking for distinct equivalence
classes*/
bmp := 0; /* all bits of bmp are initialised to '0' */
for n in 0..2^b-1
   begin
   if (any rule starts or ends at n)
   begin
      update bmp;
      if (bmp not seen earlier)
      begin
         eq := new_Equivalence_Class();
         eq->cbm := bmp;
      endif
   endif
   else eq := the equivalence class whose cbm is bmp;
table_0_j[n] = eq->ID; /* fill ID in the rfc table*/
endfor
/* end of pseudocode */
```

Figure 19: Pseudocode for RFC preprocessing for chunk *j* of Phase 0

.

Table 6:

Rule#	Chunk#0 (Src L3 bits 31..16)	Chunk#1 (Src L3 bits 15..0)	Chunk#2 (Dst L3 bits 31..16)	Chunk#3 (Dst L3 bits 15..0)	Chunk#4 (L4 protocol) [8 bits]	Chunk#5 (Dstn L4) [16 bits]	Action
(0)	0.83/0.0	0.77/0.0	0.0/0.0	4.6/0.0	udp (17)	*	permit
(1)	0.83/0.0	1.0/0.255	0.0/0.0	4.6/0.0	udp	range 20 30	permit
(2)	0.83/0.0	0.77/0.0	0.0/255.255	0.0/255.255	*	21	permit
(3)	0.0/255.255	0.0/255.255	0.0/255.255	0.0/255.255	*	21	deny
(4)	0.0/255.255	0.0/255.255	0.0/255.255	0.0/255.255	*	*	permit

```
/* Begin Pseudocode */

/* Assume that the chunk #i is  formed from combining m distinct chunks c1, c2, ..., cm of phases p1,p2, ..., pm where p1, p2, ..., pm < j */
indx := 0; /* indx runs through all the entries of the RFC table table_j_i */
listEqs := nil;
for each CES, c1eq, of chunk c1
for each CES, c2eq, of chunk c2
.....
for each CES, cmeq of chunk cm
begin
 intersectedBmp := c1eq->cbm & c2eq->cbm & ... & cmeq->cbm;/* bitwise ANDing */
 neweq := searchList(listEqs, intersectedBmp);
 if (not found in listEqs)
 begin
  /* create a new equivalence class */
  neweq := new_Equivalence_Class();
  neweq->cbm := bmp;
  add neweq to listEqs;
 endif
 /* Fill up the relevant RFC table contents.*/
 table_j_i[indx] := neweq->ID;
 indx++;
endfor
/* end of pseudocode */
```

Figure 20: Pseudocode for RFC preprocessing for chunk *i* of Phase *j*, *j*>0)

```
/* Begin Pseudocode */

for (each chunk, chkNum of phase 0)
  eqNums[0][chkNum] = contents of appropriate rfctable at memory address pktFields[chkNum];
for (phaseNum = 1..numPhases-1)
for (each chunk, chkNum, in Phase phaseNum)
begin
 /* chd stores the number and description about this chunk's parents chkPrants[0..numChkParents]*/
 chd = parent descriptor of (phaseNum, chkNum);
 indx = eqNums[phaseNum of chkParents[0]][chkNum of  chkParents[0]];
 for (i=1..chd->numChkParents-1)
  begin
    indx = indx * (total #equivIDs of chd->chkParents[i]) + eqNums[phaseNum of chd->chkParents[i]][chkNum of chd->chkParents[i]];
    /*** Alternatively: indx = (indx << (#bits of equivID of chd->chkParents[i])) ^ (eqNums[phaseNum of chkParents[i]][chkNum of chkParents[i]] ***/
  endfor
 eqNums[phaseNum][chkNum] = contents of appropriate rfctable at address indx.
endfor
return eqNums[0][numPhases-1]; /* this contains the desired classID */
/* end of pseudocode */
```

Figure 21: Pseudocode for the RFC Lookup operation with the fields of the packet in *pktFields*.

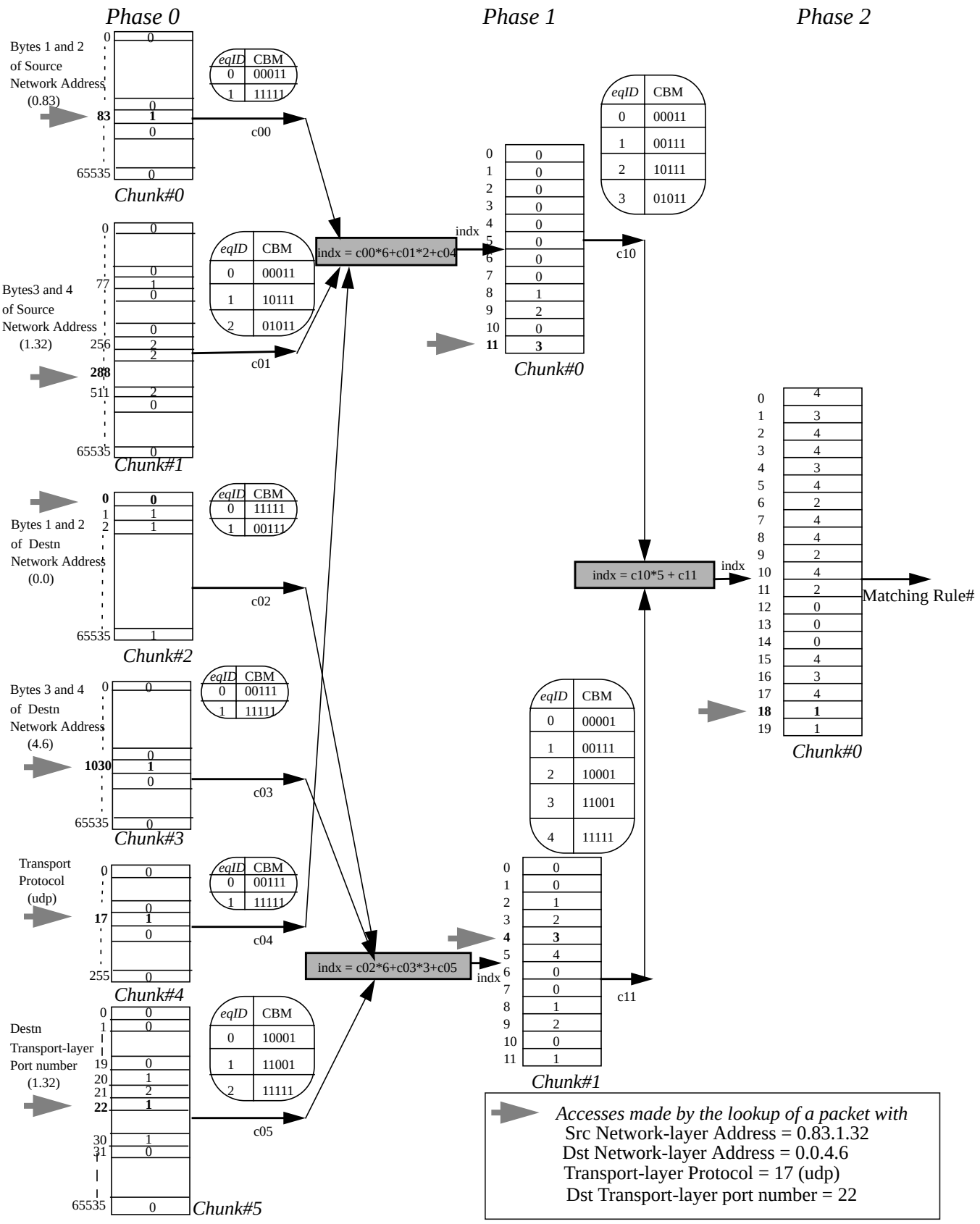


Figure 22: This figure shows the contents of RFC tables for the example classifier of Table 6. The sequence of accesses made by the example packet have also been shown using big gray arrows. The memory locations accessed in this sequence have been marked in bold.