High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching

T.V. Lakshman and D. Stiliadis

Bell Laboratories
101 Crawfords Corner Rd.
Holmdel, NJ 07733
{lakshman, stiliadi }@bell-labs.com

Abstract

The ability to provide differentiated services to users with widely varying requirements is becoming increasingly important, and Internet Service Providers would like to provide these differentiated services using the same shared network infrastructure. The key mechanism, that enables differentiation in a connectionless network, is the packet classification function that parses the headers of the packets, and after determining their context, classifies them based on administrative policies or real-time reservation decisions. Packet classification, however, is a complex operation that can become the bottleneck in routers that try to support gigabit link capacities. Hence, many proposals for differentiated services only require classification at lower speed edge routers and also avoid classification based on multiple fields in the packet header even if it might be advantageous to service providers. In this paper, we present new packet classification schemes that, with a worst-case and traffic-independent performance metric, can classify packets, by checking amongst a few thousand filtering rules, at rates of a million packets per second using range matches on more than 4 packet header fields. For a special case of classification in two dimensions, we present an algorithm that can handle more than 128K rules at these speeds in a traffic independent manner. We emphasize worst-case performance over average case performance because providing differentiated services requires intelligent queueing and scheduling of packets that precludes any significant queueing before the differentiating step (i.e., before packet classification). The presented filtering or classification schemes can be used to classify packets for security policy enforcement, applying resource management decisions, flow identification for RSVP reservations, multicast look-ups, and for source-destination and policy based routing. The scalability and performance of the algorithms have been demonstrated by implementation and testing in a prototype system.

1 Introduction

The transformation of the Internet into an important commercial infrastructure has significantly changed user expectations in terms of performance, security, and services. Internet Service Providers, while using a shared backbone infrastructure, would like to provide different services to different customers based on different service pricing or based on widely varying customer requirements. For providing this differentiated service, service providers need mechanisms for isolating traffic from different customers, for preventing unauthorized users from accessing specific parts of the network, and for providing customizable performance and bandwidth in accordance with customer expectations and pricing. In addition, service providers need mechanisms that allow routing decisions to be made not just based on destination addresses and the shortest path to it, but also based on contracts between service providers or between a service provider and a customer [14]. Consequently, routers (or packet forwarding engines in general) used in both enterprise and backbone environments should be able to provide network managers with the proper mechanisms that will allow the provisioning of these features.

Forwarding engines must be able to identify the context of packets and must be able to apply the necessary actions so as to satisfy the user requirements. Such actions may be the dropping of unauthorized packets, redirection of packets to proxy servers, special queueing and scheduling actions, or routing decisions based on a criteria other than the destination address. In the paper, we use interchangeably the terms *packet filtering* or *packet classification* to denote the mechanisms that support the above functions.

Specifically, the packet filtering mechanisms should parse a large portion of the packet header, including information on the transport protocols, before a forwarding decision is made*. The parsing results in the incoming packet being classified using a set of rules that have been defined by network management software or real-time reservation protocols such as RSVP.

Packet filtering functionality is required for example when a router is placed between an enterprise network and a core backbone network. The router must have the ability to block all unauthorized accesses that are initiated from the public network and are destined to the enterprise network. On the other hand, if accesses are initiated from a remote site of the enterprise network, they can be forwarded into the intranet and this requires filtering ability. If this level of security is not enough, another policy requirement might be that authorized access attempts from the public network be forwarded to an application level proxy server that will

*Forwarding based on transport level information is also referred to as layer-4 forwarding.

authenticate the access. Clearly, filtering mechanisms are very useful at the edge of an enterprise network. In an edge network node, the router might need to identify the traffic that is initiated from specific customers, and either police it or shape it to meet a predefined contract. Indeed, these are the actions that are required by some of the differentiated services model proposals that are being considered for standardization by the IETF [10].

It is evident that most filter rules naturally apply to a whole range of addresses, port numbers, or protocols, and not just to single predefined hosts or applications. Aggregation, for instance of addresses, is not only required because customers are usually allocated blocks of addresses, but also because it is necessary to keep the network manageable. Therefore, the specification of the packet classification policies must allow aggregations in their definitions. This means that packet classification algorithms must be be able to process rules that define combinations of ranges of values. If the algorithms can only handle exact values and do not support aggregation, preprocessing is required to translate the ranges to exact values. This is infeasible since ranges can grow exponentially with length of the packet field on which the ranges are defined.

A trend worth noting is that even though packet filtering was thought of as a tool necessary only at the network access points and mainly for firewall or security applications, it is now becoming apparent that it is a valuable tool for performing traffic engineering and meeting the new service requirements of the commercial Internet. Filtering policies that use the full information of the packet header can be defined for distributing the available system or network resources. The main consequence of these new uses is that all packet classification actions must be performed at wire-speed, i.e., the forwarding engines must have enough processing power to be able to process every arriving packet without queueing since without header processing it is not possible to differentiate packets to provide differentiated services.

The main contributions of this paper are algorithms that use multi-dimensional range matching that enable Gigabit routers to provide wire-speed packet filtering and classification in a traffic independent manner (i.e. we do not rely on traffic dependent caching or average case results to achieve fast execution times). To our knowledge, our proposed schemes are the first schemes that allow thousands of filter rules to be processed at speeds of millions of packets per second with range matches on 5 or more packet fields in a traffic independent manner. Specifically, we present three algorithms: The first algorithm takes advantage of bit-level parallelism which combined with very elementary bit-operations results in a scheme that supports a large number of filter rules. The second algorithm extends the performance of the first algorithm by making efficient use of memory. It provides a means for balancing the the time-space tradeoff in implementation, and allows optimization for a particular system taking into account the available time for packet processing, the available memory, and the number of filter rules to be processed. Furthermore, the algorithm allows on-chip memory to be used in an efficient and traffic independent manner for reducing worst-case execution time. This is unlike typical caching schemes which are heavily traffic dependent and only improve average case performance. The performance metric for all our schemes is worst-case execution time, simple operations to make it amenable to hardware implementation if necessary, and space requirements which are feasible with current memory technology and costs. The implementation simplicity, scalability and performance of our filtering have been demonstrated in a prototype router with interfaces operating at a million packets per second.

Our third algorithm considers the special case of filter rules on two fields. This is motivated by important applications such look-ups for multicast traffic forwarding and policy-based routing. To elaborate on this example, when a forwarding engine supports a multicast protocol like PIM (sparse mode or dense mode) [13] or DVMRP [26], the forwarding decision has to be made on both the source address value and the multicast group value. Depending on the protocol, the forwarding engine may have a forwarding entry for a given group value irrespective of source addresses, and also have forwarding entries for a given group value and source subnet. Given the increasing importance of multicast forwarding in the Internet, it would be ideal if a simple algorithm could be used for making multicast forwarding decisions. Since the search for the source addresses may use the same forwarding information base as that used for unicast routing, the same type of CIDR (Classless Inter-Domain Routing) aggregations [15] are likely to be used. CIDR aggregations introduced the notion of prefix in the definition of routing entries. In other words an entry in the forwarding base is defined as a value and a mask. The mask defines the number of bits of the destination address of a packet that can be ignored when trying to match the destination address of the packet to the particular entry of the forwarding base. The bits that can be masked-out are always in the less significant portion of the address. Thus, the values in the forwarding engines can thought as prefixes. For the case of IPv4, prefixes can have a length between 1 and 32 bits. We present a linear space, O(prefix length) scheme which can be used to implement 2-dimensional lookups at rates of millions of packets per second for more than $128K$ entries in the forwarding table. Considering that multicast forwarding tables in the core backbone might include several hundreds of thousands of entries, even a solution that uses $O(n \log n)$ space with a moderate constant or $O(log^2 n)$ time may not be feasible when the number of entries n is that high.

2 Design Goals

We first try to identify the main requirements that a packet classification algorithm must satisfy in order to be useful in practice.

2.1 The Requirement for Real-Time Operation

Traditional router architectures are based on flow-cache architectures to classify packets. The basic idea is that packet arrivals define flows [9, 17], in the sense that if a packet belonging to a new flow arrives, then more packets belonging to that flow can be expected to arrive in the near future. With this expected behavior, the first packet of a flow is processed through a slow path that analyzes the complete header. The header of the packet is then inserted into a cache or hash table together with the action that must be applied to the first packet as well as to all other packets of the flow. When subsequent packets of that flow arrive the corresponding action can be determined from the cache or hash table.

There are three main problems associated with this architecture or any similar cache-based architecture when applied to current Internet requirements:

1. In current backbone routers, the number of flows that are active at a given interface is extremely high. Recent studies have shown that an OC-3 interface might have an average of 256K flows active concurrently [24] †. For this many number of flows, use of hardware caches is extremely difficult, especially if we consider the fact that a fully-associative hardware cache may be required. Caches of such size will most likely be implemented as hash tables since hash tables can be scaled to these sizes. However, the $O(1)$ look-up time of a hash table is an average case result and the worst-case performance of a hash table can be poor since multiple headers might hash into the same location. The number of bits in the packet headers that must be parsed is typically between 100 and 200 bits, and even hash tables are limited to only a couple of million entries. So any hash function that is used must be able to randomly distribute 100 to 200 bit keys of the header to no more than 20-24 bits of hash index. Since there is no knowledge about the distribution of the header values of the packets that arrive to the router, the design a good hash function is not trivial.

2. Due to the large number of flows that are simultaneously active in a router and due to the fact that hash tables generally cannot guarantee good hashing under all arrival patterns, the performance of cache based schemes is heavily traffic dependent. If a large number of new flows arrive at the same time, the slow path of the system that implements the complete header matching can be temporarily overloaded. This will result in queueing of packets before they are processed. But in this case, no intelligent mechanism can be applied for buffering and scheduling of these packets because without header processing there is no information available about the destination of the packets or about any other fields relevant to differentiation. So it is possible that congestion and packet dropping happen due to processing overload and not due to output link congestion.

   To better illustrate this, consider the model in Figure 1. Packets arrive to the interfaces and are placed in a queue for processing. After the packet classification and next-hop lookup operations are performed, they are forwarded to the outgoing interfaces where they are queued for transmission. Clearly, some interfaces may be idle even though there are packets waiting to be transmitted in the input queues. For example, all packets destined for output 1 can be blocked in the slow path processing module behind packets that are destined to other outputs. Output 1 remains idle, although there are packets in the buffers available for transmission. Obviously such a system will suffer from Head-of-Line blocking that will limit the throughput substantially. Note, that the Head-of-Line problem can be diminished, if there is knowledge about the destination of more than one enqueued packet [20]. However, the fundamental problem of the system of Figure 1 is that the destination or the context of the packet is not actually known before the packet is processed. Thus, it is impossible to apply any intelligent queueing mechanisms at the input queues and head-of-line blocking cannot be eliminated.

3. A commercial Internet infrastructure should be robust and should provide predictable performance at all times. Variations in the throughput of forwarding engines based on traffic patterns are undesirable and make network traffic engineering more difficult. In addition, the network should not be vulnerable to attacks from malicious users. A malicious user or group of users discovering the limitations of the hash algorithms or caching techniques, can generate traffic patterns that force the router to slow down and drop a large portion of the packets arriving at a particular interface.

†Note the by active we do not imply that the flow currently has a backlog of packets to be served. The definition of active flows for caching look-up information is different from the definition for scheduling because caching information changes at a slower time scale.

Summarizing, we claim that any packet queueing delays are only acceptable after the classification step is performed, if provisioning of differentiated services and robustness are important. In particular, the queueing delays before the complete processing of a packet can be no larger than the maximum allowed delay for the flow with the minimum delay requirement (which could be extremely small if constant bit rate flows are supported). This *no-queueing before processing* principle applies because it is the header processing (including packet filtering) operation that enables the router to determine the quality-of-service (QoS) level to be accorded to a particular packet. Hence, large queues formed while waiting for the filtering operation can violate quality-of-service for some flows even before the router determines the QoS to be accorded to the flow. The implication that this has on the design of packet filtering algorithms is that it is the worst-case performance of the algorithms that determines the true maximum packet processing rate and not the average case performance (the averaging being done on filter rule combinations and traffic arrival patterns). If average case performance is used to determine supported packet processing speeds, then buffering is needed before filtering. To estimate delays in this undifferentiated-traffic buffer, we need a characterization of the the variance in execution times (which is difficult to determine) and we need to predict traffic patterns at different interfaces. The delay in this pre-filtering buffer can cause QoS to be violated for flows with stringent delay constraints if there is any error in estimating these quantities. Hence, it is preferable to avoid queueing before header processing.

2.2 Criteria for efficient packet classification and system constraints

We can now outline, based on the prior discussion, the criteria that an efficient classification algorithm must meet:

1. The algorithm must be fast enough for use in routers with Gigabit links. Internet Service Providers are envisaged to build networks with link capacities of 2.4 Gigabits/s and more. Any packet classification scheme for use in core networks must be scalable to these speeds.

2. The algorithm must be able to process every packet arriving to the physical interfaces at wire-speed. Recent traffic studies have shown that 75% of the packets are smaller than the typical TCP packet size of 552 bytes. In addition, nearly half the packets are 40 to 44 bytes in length, comprising of TCP acknowledgments and TCP control segments [24]. Since the algorithm cannot use buffering to absorb variation in execution

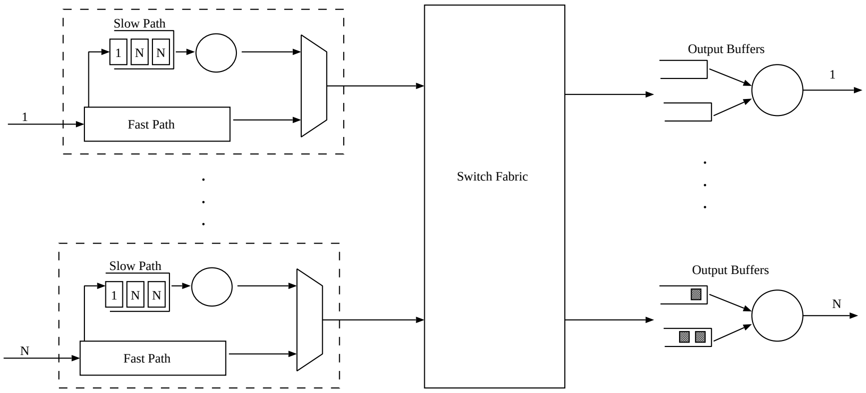

Figure 1: Queueing model of a system that uses a cache-based architecture for packet classification.

times, it must operate at wire-speed when all packets are as small as 44 bytes. This means that the algorithm must have provably small worst-case execution times which are independent of traffic patterns.

3. Classification rules must be based on several fields of the packet header, including among others source and destination IP addresses, source and destination port numbers, protocol type and Type-of-Service. The rules must be able to specify ranges and not just exact values or simple prefixes.

4. For some applications, it might be possible to limit the requirements to only two dimensions and to have ranges defined only as prefixes. This is a more restricted problem than the general classification problem but it has a very useful application in both multicast lookups and in RSVP reservations that use either wild-card filters or CIDR aggregations [15, 28].

5. It is possible that some packets may match more than one rule. The algorithm must allow arbitrary priorities to be imposed on these rules, so that only one of these rules will finally be applicable to the packet.

6. Updates of rules are rare compared to searches in the data structures. In particular, the frequency of updates is in the time-scale of tens of seconds or longer whereas look-ups happen for every processed packet. At a packet processing rate of 10^6 packets per second, the ratio of look-ups to updates is 10^7. Hence, the algorithms can be optimized for lookups even if it means degrading update performance to some extent.

7. Memory accesses are expensive and are the dominant factor in determining the worst-case execution time.

8. Memory is organized in words of size w and the cost of accessing a word is the same as the cost of accessing any subset of bits in a word.

9. Memory cost can be relatively low if technologies such as Synchronous Dynamic RAMs (SDRAMs) are used. These devices can provide very large capacity combined with a high access speed, provided that accesses are sequential. A packet classifier implementation that requires multiple sequential accesses in a high-speed SDRAM memory might be more affordable than an algorithm of lower time-complexity that uses higher-cost, lower capacity memories like SRAMs.

10. For operation at very high speed the algorithm must be amenable to hardware implementation. While we do not preclude software implementations of our proposed algorithms, we are primarily interested in algorithms that are implementable in hardware as well and are not restricted to only a software implementation.

3 Previous Work

The idea for packet filtering, or classification in general, was initiated in [16] and was later expanded in [19, 27]. The architectures and algorithms presented in these papers were targeted mainly for an end-point and their main goal was to isolate packets that are destined to specific protocols or to specific connections. The algorithms used, although they involved a linear parsing of all the filters, were fast enough to operate at end-point link capacities. Obviously these implementations do not scale to very high speeds.

An interesting variation was presented in [1] where the first hardware implementation of packet filters was reported. The implementation, although fast enough to support an OC-12 link, is restricted to only a small number of rules (< 12) and is not general enough for a commercial high-speed router. The implementation uses a pipelined architecture, resulting in O(1) performance using O(N) processing elements for O(N) rules. Clearly, such an algorithm cannot scale to a large number of filter rules since it requires a linear number of processing elements. Moreover, this scheme was designed for rules that required exact matching and not for rules defined as ranges.

The general packet classification problem that we consider can be viewed as a point location problem in multidimensional space. This is a classic problem in Computational Geometry and numerous results have been reported in the literature [5, 6, 11]. The point-location problem is defined as follows: Given a point in a d-dimensional space, and a set of n d-dimensional objects, find the object that the point belongs to. Most algorithms reported in the literature deal with the case of non-overlapping objects or specific arrangements of hyperplanes or hypersurfaces of bounded degree [22]. When considering the general case of $d > 3$ dimensions, as is the problem of packet classification, the best algorithms considering time or space have either an $O(\log^{d-1} n)$ complexity with $O(n)$ space, or an $O(\log n)$ time-complexity with $O(n^d)$ space. [22]. Though algorithms with these complexity bounds are useful in many applications, they are mostly not directly useful for packet filtering. For packet filtering, the algorithms must complete within a specified small amount of time for n, the number of filters, in the range of a few thousands to tens of thousands. So even the algorithms with poly-logarithmic time bounds are not practical for use in a high-speed router.

To illustrate this, let us assume that we would like the router to be able to process 1K rules of 5 dimensions within $1\mu s$ (to sustain a 1 million packets per second throughput). An algorithm with $\log^4 n$ execution time and $O(n)$ space requires 10K memory accesses per packet. This is impractical with any current technology. If we use an $O(\log n)$ time $O(n^4)$ space algorithm, then the space requirement becomes prohibitively large since it is in the range of 1000 Gigabytes.

To the best of our knowledge, there is no algorithm reported in the literature for the general d-dimensional problem, of point-location with non-overlapping object, with lower asymptotic space-time bounds. In addition, our requirements are not for point location given non-overlapping objects, but for point location with overlaps being permitted and prioritization used to pick one object out of many overlapping solutions.

For the special case of two dimensions and non-overlapping rectangles a number of solutions have been reported with logarithmic complexity and near-linear space complexity [12]. However, these algorithms do not consider the special problem related to longest-prefix matches where arbitrary overlaps may be present and overlaps are resolved through the longest prefix priority. An even better solution has been reported in [2] where the time complexity is $O(\log\log N)$. However, the algorithm is based on the stratified trees proposed by van Emde Boas [23, 3] for searches in a finite space of discrete values. The data structures used require a perfect hashing operation in every level of the tree. The preprocessing complexity, without using a randomized algorithm, of calculating the perfect hash is $O(min(hV, n^3))$ where h is the number of hash functions that must be calculated and V is the size of the space. Note, that for 2-dimensional longest-prefix lookups this can result, even for a small number of rules, in executions requiring 2^{32} cycles which is impractical, even if preprocessing is only required once every several seconds.

4 General Packet Classification Algorithms

A simple approach to the problem of multi-dimensional search, as used for packet filtering, is to use decomposable search. Here the idea is to state the original query as the intersection of a few numbers of simpler queries. The challenge then, for instance to obtain a poly-logarithmic solution, is to decompose the problem such that the intersection step does not take time more than the required bound. To achieve these poly-logarithmic execution times, various sophisticated decompositions and query search data structures have been proposed. However, as was pointed out before, even a $\log^4 n$ solution for 5 dimensional packet filtering is not practical for our application where n can be in the thousands. Therefore, we need to employ parallelism of some sort. Moreover, we require simple elemental operations to make the algorithm amenable to hardware implementation. Our cost metric of memory accesses being of unit cost till a word length is exceeded implies that bit level parallelism in the algorithm would give speed-ups. Instead of looking for data structures which give the best asymptotic bounds, we are interested in decomposing the queries such that sub-query intersection can be done fast (as per our memory-access cost metric) for n in the thousands and memory word-lengths that are feasible with current technology.

The first point to note is that our packet-filtering problem involves orthogonal rectangular ranges. This means that a natural decomposition of a k-dimensional query in a k-dimensional orthogonal rectangular range is to decompose it into a set of 1-dimensional queries on 1-dimensional intervals. The problem is that when we do this for our problem, each simple query can generate a solution of $O(n)$ size. This is because we can have arbitrary overlaps and so $O(n)$ ranges may overlap a query point in 1 dimension. Consequently, the intersection step can take time $O(n)$. Nevertheless, this solution is far more practical for our packet filtering problem than other poly-log solutions because we can take advantage of bit-level parallelism.

To summarize, given our constraints (particularly the need for hardware implementation) and cost metrics (in particular memory access cost per bit incrementing only at word boundaries), the number of filter rules being of the order of a few thousands, and the number of dimensions being 5, the simple approach of decomposing the search in each dimension (which can be done in parallel) followed by a linear time combining step is more useful than a sophisticated $O(\log^4 n)$ (n being the number of filter rules) time algorithm.

Below, we describe an algorithm which needs $k*n^2+O(n)$ bits of memory for each dimension, $\lceil\log(2n)\rceil + 1$ comparisons per dimension (which can be done in parallel for each dimension), and $\lceil n/w\rceil$ memory accesses for a pairwise combining operation. We then present a second algorithm which can reduce memory requirements to $O(n\log n)$ bits while increasing the execution time by only a constant (as long as $\log n <= w$ which would certainly be the case). This second algorithm has two other benefits. The constant increase in execution time can be traded off for increased memory, allowing the algorithm to be optimized for the available time and memory budget. Also, it can exploit on-chip memory in a traffic independent manner to speed up worst-case bounds (unlike typical caching schemes which only speed up average-case bounds and are traffic dependent). Specifically, if there is $(k*n^2)/l$ bits of on-chip memory then the number of off-chip memory accesses is $\lceil\log(n)/w\rceil/(2l)\rceil$ where w is the word length. The number of on-chip memory accesses is $\lceil n/w_{on-chip}\rceil$ where $w_{on-chip}$ is the on-chip memory word length.

4.1 Packet Classification based on Bit-Parallelism

Although, the algorithm we will describe has linear time complexity, its use of bit-level parallelism significantly accelerates the filtering operation for any practical implementation. The filtering rules are changed very infrequently in comparison to the frequency of search operations. Hence, extra preprocessing can be used to speed up searches.

We assume that a set of n packet filtering rules in k dimensions are defined. Let $r_m = \{e_{1,m}, e_{2,m}, \ldots, e_{k,m}\}$ denote the set of ranges that define rule r_m in the k dimensions.

The preprocessing part of the algorithm is as follows:

1. For each dimension $j, 1 <= j <= k$, project all intervals $e_{j,i}, 1 <= i <= n$ on the j-axis, by extracting the j^{th} element of every filter rule for all n filter rules. There are a maximum of $2n+1$ non-overlapping intervals that are created on each axis. Let us denote by $P_j, 1 <= j <= k$ the k sets of such intervals.

2. For each interval $i \in P_j, \forall j \in \{1\ldots k\}$, create sets of rules $R_{i,j}, 1 \le i \le 2n+1, 1 \le j \le k$, such that a rule r_m belongs in set $R_{i,j}$ if and only if, the corresponding

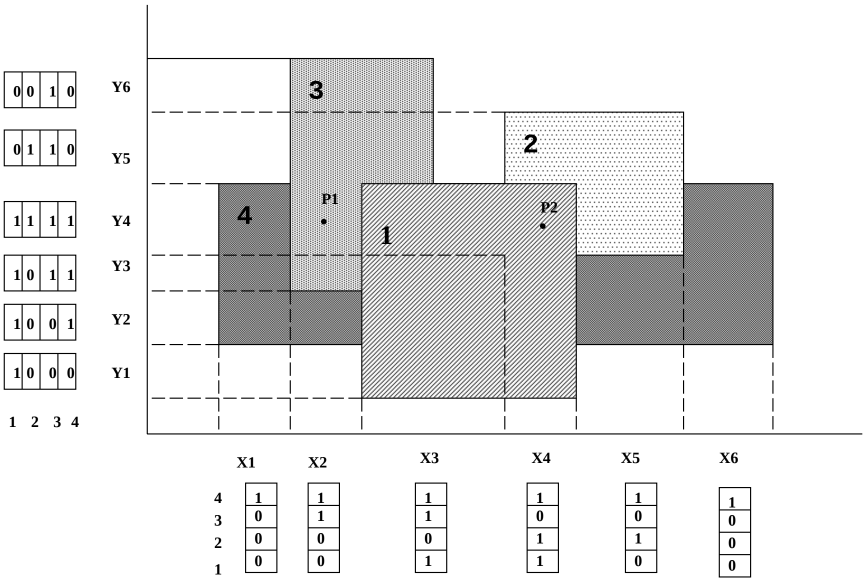

Figure 2: General packet classification using bit-parallelism.

interval i overlaps in the j^{th} dimension with $e_{j,m}$, i.e. iff $i \subseteq e_{(j,m)}$ where $e_{(j,m)}$ is the j^{th} element of rule r_m.

Without loss of generality, we assume that rules are sorted based on their priorities. Assume that a packet with fields $E_1, E_2, \ldots E_k$ arrives to the system. The classification of the packet involves the following steps.

1. For each dimension j, find the interval, say i_j on set P_j that E_j belongs to. This is done using binary search (requiring $\lceil \log(2n+1) \rceil + 1$ comparisons) or using any other efficient search algorithm.

2. Create intersection of all sets $R_{i_j,j}$, $i_j \in \{1, 2, \ldots 2n+1\}$. This can be done by taking the conjunction of the corresponding bit vectors in the bit arrays associated with each dimension and then determining the highest priority entry in the resultant bit vector (see explanation below).

3. The rule corresponding to the highest priority entry must be applied to the arriving packet.

Note, that the on-line processing step involves an intersection among the potential sets of applicable filter rules which are obtained considering only one dimension at a time. These potential solution sets may have cardinality $O(n)$ since we have assumed that rules may have arbitrary overlaps. The intersection step involves examining each of these rules at least once and hence the algorithm has time complexity $O(n)$.

To accelerate the execution time, we can take advantage of bit-level parallelism. Each set $R_{i,j}$ is represented by a bitmap n-bits long which acts as the indicator function for the set. Let $B_j[i,m]$ be a $(2n+1) \times n$ array of bits associated with each dimension j. We can store each set $R_{i,j}$ as a bit vector in column i of the bit array $B_j[i,m], 1 <= m <= n$, where bit $B_j[i,m]$ is set if and only if, the rule r_m belongs in set $R_{i,j}$. The intersection operation is then reduced to a logical-AND among the bitmaps that are retrieved after the search in each direction. To be able to select the highest priority rule, we rely on the rules being ordered based on priorities. The bitmaps are organized in memory into words of width w where each word is the unit of memory access. We can implement the intersection by reading sequentially the words of all dimensions and implementing the logical-AND. The first rule that we find through this process is the highest priority rule. Clearly this takes $\lceil k * n/w \rceil$ memory accesses. By making w large, the worst-case execution time can be reduced further.

Let us consider the example in 2-dimensions, shown in Figure 2, to illustrate the functioning of the algorithm. Rules are represented by 2-dimensional rectangles that can be arbitrarily overlapped. The preprocessing step of the algorithm projects the edges of the rectangles to the corresponding axis. In the example shown, the four rectangles create seven intervals in each axis. In the worst case, the projection results in a maximum of $2n+1$ intervals on each dimension. We next associate a bitmap with each dimension, as is shown in Figure 2. A bit in the bitmap is set, if and only if, the corresponding rectangle overlaps with the interval that the bitmap corresponds to. Note, that because of the method by which the intervals were created, it is not possible for a rectangle to overlap, say, only with half an interval. Assume that the packet represented by point $P1$ arrives to the system. During the first on-line step, we locate the intervals in both axis that contain this point. In the example, these are intervals X2 and Y4 for the X and Y axis respectively. In the second step, we use the bitmaps to locate the highest priority rectangle that covers this point. Note that rectangles are numbered based on their priorities. After the logical-AND of the bitmaps, the first bit that is set in the resulting bitmap is that corresponding to rectangle 3 which is the highest priority rectangle, amongst all those overlapping point P1.

4.1.1 Hardware Implementation

The key point behind the hardware implementation is that the algorithm performs only very simple operations. Since the only hardware elements that are required for the binary search operation (to locate enclosing intervals) are an inte-

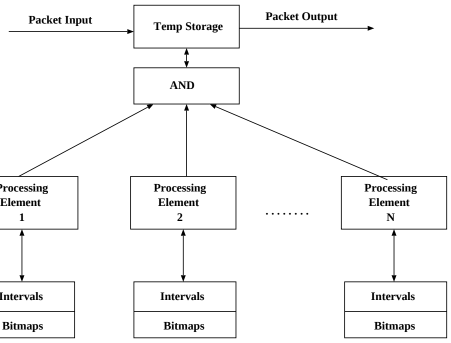

Figure 3: Architecture block diagram of a parallel implementation.

ger comparator and counter, and the only operation for the intersection is a parallel AND operation, the complexity of such a processing element is very low. The straightforward approach is to use a different processing element for each dimension (see Figure 3). Each processing element consists of a single comparator, a state machine and 2 local registers. The processing elements implement the binary search on all intervals in parallel. The result of this search is a pointer to a bitmap for each direction. The second step of the algorithm requires a parallel access to all bitmaps and a logical AND operation amongst them. The first time that this operation results in a non-zero value, the corresponding filter has been located. The algorithm requires one more access to the memory to retrieve the actions that may be associated with this filter.

The algorithm has been implemented in 5 dimensions in a high-speed router prototype using a single FPGA device and five 128 Kbyte Synchronous SRAMs chips supporting up to 512 rules and processing 1 million packets per second in the worst case. This is achievable despite the device being run at a very low speed of 33 MHz. Since we used the same memory chips as those used in the L2-caches of personal computers, the cost of the memories is trivial. The device can be used as a co-processor, next to a general purpose processor that handles all the other parts of IP forwarding or firewall functions.

4.2 Packet Classification based on Incremental Reads

The next algorithm we propose uses incremental reads to reduce the space requirements. The algorithm allows designers to optimize time-complexity and space. Since the dominant factor determining execution times is off-chip memory accesses, the availability of on-chip memory and the use of the proposed algorithm can significantly increase the number of filter rules that can be applied within the given time constraint.

The main idea used in developing the proposed algorithm is the following. Consider a specific dimension j. There are at most $2n+1$ non-overlapping intervals that are projected onto this dimension. Corresponding to each of these intervals there is a bitmap of n bits with the positions of the 1s in this bitmap indicating the filter rules that overlap this interval. The boundary between intervals is a point where some the projections of some filter rules terminate and those of some filter rules start. If we have exactly $2n+1$ intervals, then the set of filter rules that overlap any two adjacent intervals l and m differ by only one rule (i.e., at the boundary between interval l and m either a filter rule's projection starts, or a filter rule's projection ends). This implies that the corresponding bitmaps associated with these two intervals differ in only bit. Hence the second bit map can be reconstructed from the first by just storing, in place of the second bit map, a pointer of size $\log n$ which indicates the position of the single bit which is different between these two bit maps. Carrying this argument further, a single bit map and $2n$ pointers of size $\log n$ can be used to reconstruct any bit map. This cuts the space requirement to $O(n \log n)$ from $O(n^2)$ but increases the number of memory accesses by $\lceil (2n \log n)/w) \rceil$.

The above argument is not changed when more than one filter starts or terminates at a particular boundary point between two adjacent intervals. This is because if a boundary point has more than one, say i, filters terminating or starting at that point then the number of intervals in that dimension is reduced by $i-1$. Hence, we still need only $2n$ pointers (each filter terminating or starting needs exactly one pointer even if they all terminate or start at the same boundary point).

We can now generalize this basic idea by storing $(2n+1)/l$ complete bitmaps instead of just one bitmap. These bitmaps are stored such that at most only $\lceil (2n+1)/2l \rceil$ pointers (pointer are stored such that retrieval starts from the nearest lower or higher position where a complete bitmap is stored) need to be retrieved to construct the bitmap for any interval. The preprocessing phase is as follows:

1. For each dimension $j \in \{1 \dots k\}$ do
2. Determine the, at most $2n+1$, non-overlapping intervals for dimension j by projecting all intervals $e_{j,i}$, $1 <= i <= n$ on the j-axis. This step is the same as the projection step for the algorithm proposed in the previous sections.
3. Generate set of overlapping rules for first interval and store its bit map in storage associated with this interval.
4. For all intervals $i \in \{2 \dots 2n+1\}$ do
   Generate set of overlapping rules for interval i. If this bit map has l or more bits different from most recent bit map which was stored, then store this bit map in storage associated with interval i. Otherwise, increment i. At most $\lceil 2n+1/l \rceil$ bit maps are stored since there are n bits, each bit can change at most twice, and the successive bit maps have at least l bits which are different.
5. Fill in pointers indicating changed bits from previous interval. There are at most $l-1$ intervals between intervals for which bitmaps have been stored. Starting from each bit map, store pointers which indicate the bits which have changed from the preceding interval. "Preceding interval" is the interval with lower index for bit maps in lower half of the $l-1$ intervals between stored bit maps and it is the interval with the higher index for bit maps at the mid-point or upper half of the $l-1$ intervals.

The per-packet processing performed to find the applicable filter rule, if any, is as follows:

1. For each dimension $j \in \{1 \dots k\}$ do
2. Find the interval, say i, on set P_j that E_j belongs to. This is done by binary search (requiring $\lceil \log(2n+1) \rceil + 1$ comparisons) or using any other efficient search algorithm.

3. If this interval has its complete bit map, b_i, stored then retrieve this bit map. This require $\lceil n/w \rceil$ memory accesses. Otherwise, first retrieve the bit map for the interval closest to i with a stored bit map. This bit map has at most $\lceil (l-1)/2 \rceil$ bits different from the retrieved bit map. Fetch the, at most $\lceil (l-1)/2 \rceil$ pointers corresponding to all intervals in between i and the interval whose bit map was received. This case requires $\lceil n/w \rceil + \lceil ((l-1)/2) * (\log n)/w \rceil$ memory accesses. Construct the bit map for i using the pointers in sequence.

4. Create a new bit map as the logical-AND of the retrieved bit maps for all k dimensions. Note that the AND operation can be done progressively as the bit maps are being constructed and does not necessarily require the entire bit map for each dimension to be completely retrieved.

5. The index of the leading 1 in this bit map gives the applicable filter rule.

4.3 Choice of l

One possible criteria for choice of l is to lower the memory requirement from $O(n^2)$. Setting $l = 2n+1$ is an extreme case which reduces memory requirements to $O(n \log n)$ but requires retrieving $n-1$ pointers. Let us present the tradeoff with an example. If we assume that bits of the bitmap are retrieved through pointers, as is the case when $l = 2n+1$, then the total time for retrieving the bitmaps in each direction becomes $\lceil 2n \log n/w \rceil$ which can be much higher than the $\lceil n/w \rceil$ time required by the first algorithm. However, at the expense of higher execution time, the space requirement is reduced substantially.

If on-chip memory is available however, complete bitmaps may be retrieved together with the pointers. The basic assumption behind the utilization of on-chip RAMs is that they be designed to be extremely wide. It is thus possible to increase the data bus width by at least a factor of 4 over an off-chip RAM. Current technologies, such as the ones offered by major ASIC vendors [25, 21], allow large portions of Synchronous Dynamic RAMs (SDRAMs) to be allocated in the same chip as logic. Note, that SDRAMs are ideal for retrieving bitmaps since they offer the best performance when accesses are sequential.

So, let us assume that on-chip memory can be α times as wide as off-chip memory. Let us assume that a full bitmap is kept in the on-chip memory for every l words. The total time required for retrieving this bitmap can be calculated as $t = n/\alpha w$. This time must be equal to the time required to retrieve at least $l/2$ pointers from the off-chip memory, or $t = l \log n/2w$ off-chip memory accesses. From the above two relations, we can easily calculate the optimal value of l for a given technology as

$$l = \frac{2}{\alpha} \frac{n}{\log n}.$$

For example, if we assume $\alpha = 4$ and $n = 8K$ we get $l = 315$ and we will need approximately 64 cycles to complete the operation. This will translate, using a 66MHz clock, to a processing rate of 1 million packets per second. Note, that the total space requirement for the $8K$ filters is 32 Kbytes of on-chip memory and 32 Kbytes of off-chip memory for each dimension. This is definitely within the capabilities of current technology.

5 Classification in Two Dimensions

The 2-dimensional classification problem is of increasing importance in the evolving Internet architecture of the future. Drastically changing user expectations will necessitate the offering of different types of services and service guarantees by the network. Although RSVP, or similar reservation protocols, can offer end-to-end Quality-of-Service guarantees for specific flows, it is not clear whether such a reservation based model can be scaled for operation over high-speed backbones where a very large number of flows can be concurrently active. An alternative approach that has been proposed is route aggregated flows along specific traffic engineered paths. This directed routing is based not only on the destination address of packets, but on the source address as well [18]. RSVP or similar reservation protocols can be used for setting the aggregation and routing rules [18, 4]. However, the key mechanism needed to support such a scheme in a high-speed core router is a 2-dimensional classification or lookup scheme that determines the next hop, and the associated resource allocations, for each packet as a function of both the source and destination addresses. In addition, multicast forwarding requires lookups based on both the source address and multicast groups [13, 26].

The 2-dimensional look-up problem is defined as follows: A query point p is a pair (s, d). For the forwarding problem, these values could correspond to source and destination addresses. For the multicast look-up application, the query point can correspond to the source address of a packet and to a group id that identifies the multicast group that the packet belongs to. A 2-dimensional look-up table consists of pairs (s_i, d_i) where each s_i is a prefix of possible source addresses and each d_i is a contiguous range or point, of possible group identifiers or destination addresses. Each pair defines a rectangle in 2-dimensions. Note that rectangles can have arbitrary overlaps. Given a query point p, the search or look-up problem is to find an enclosing rectangle (if any), say $r_j = (s_j, d_j)$, such that $p = (s, d)$ is contained in r_j, and such that s_j is the most specific (longest) matching prefix of s. The d_j can be ranges including prefix ranges. The matching d_j, when multiple matches occur for a specific s_j due to overlaps in the d direction, is taken to be the one of highest priority. Note that the d dimension allows any contiguous range and is not restricted to prefix ranges only. Therefore, if the d direction corresponds to destination addresses, then ranges in the destination addresses do not have to be in powers of 2 (which would be the case with prefix ranges). This might be particularly useful if destination addresses are concatenated with layer-4 destination ports or some other similar header field (to form a"2 1/2 dimensional" lookup).

We are interested in solutions for use in IP routers. This means that the look-up tables may have as many as 2^{16} entries and perhaps more. Also, we are interested in only worst-case performance of the algorithms since we want to avoid queueing for header processing in order to provide QoS guarantees.

Let n denote the number of entries in the multicast forwarding table. A simple solution that takes only $O(\log n)$ time and $O(n^2)$ space is to have an $n \times n$ array with each entry representing the highest priority rectangle that encloses the point corresponding to the coordinates represented by the entry. The look-up is done with two binary searches. However, this is clearly impractical in space when the number of filtering rules is $n = 2^{16}$. The $O(n^2)$ space is because the same rectangle can be represented in $O(n)$ locations.

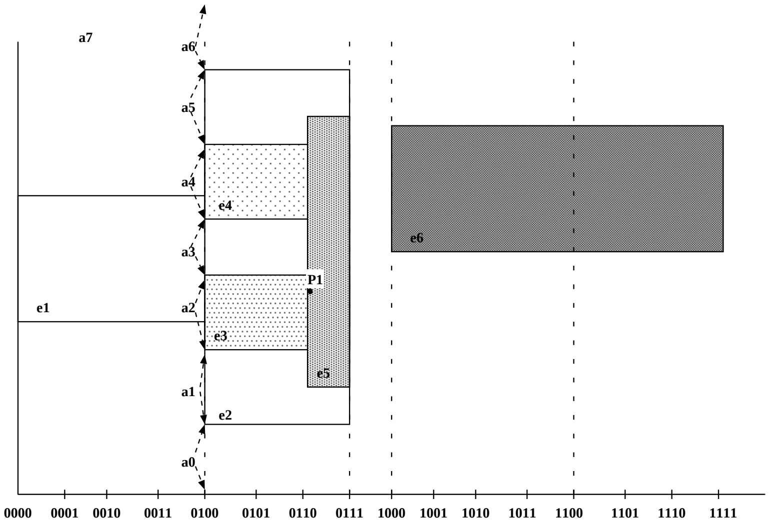

Figure 4: Operation of the 2-dimensional algorithm when one dimension includes only intervals created by prefixes

We would like to maintain the same time complexity while storing all the rectangles using only $O(n)$ space. We cannot directly use known solutions to the problem of rectangular point location since we can have arbitrary overlapping rectangles. In the proposed algorithm, we make use of the fact that in the s dimension our rectangles have lengths which are in powers of 2. This is because these s ranges are always prefix ranges.

The restriction of ranges in one dimension to be prefix ranges provides constraints that can be exploited. To illustrate this consider Figure 4. All prefix range intervals can only have sizes which are powers of two and totally arbitrary overlaps are not possible (two prefixes of the same length do not overlap). also, a range can only start from an even point and terminate at an odd point. Based on these observations, we can split the set of ranges into several distinct cells distinguished by the length of the prefix (or, equivalently the size of the range).

Let the s field be of length ls bits and the d field be of length ld bits. Let $R_1, R_2, \ldots, R_{ls}$ denote subsets of the set R of n rectangles such that subset R_i consists of all rectangles formed from prefixes that are i bits long. Let n_i denote the number of prefixes of length i that are present in the lookup table. Assume that these prefixes in the s dimension are numbered in ascending order of values obtained by extending all prefixes to their maximum length ls by adding sufficient zero bits to each of the specified prefixes. Denote the n_i prefixes of length i by $P_i^1, P_i^2, \ldots, P_i^{n_i}$. With each prefix P_i^j there is an associated set of rectangles, $R_i^j = \{(P_i^j, d_i^1), (P_i^j, d_i^2), \cdots\cdots, \}$, that have P_i^j as one of their sides. Here, the $\{d_i^1, d_i^2, \cdots\cdots, \}$, Are ranges in the d dimension and can overlap. The set of rectangles R_i is the union of sets $R_i^1, R_i^2, \ldots, R_i^{n_i}$ where each of the R_i^j is the set of rectangles associated with the j^{th} prefix of length i. Note that the sets R_i^j are disjoint and that all rectangles in R_i^j that match p have higher priority than rectangles in R_t^j that match p if $i > t$. This is because rectangles in R_i^j are formed with longer prefixes than those in R_t^j since $i > t$.

In the example of Figure 4, the set of rectangles with prefix equal to 1 is $R_1 = \{\{e1\}, \{e6\}\}$. Note that each prefix of length m covers $1/2^m$ of the total s range. There is a total of $n_2 = 1$ prefixes of length 2. The set R_2 of rectangles formed with prefixes of length 2 consists of the rectangles $\{e2, e3, e4\}$. From Figure 4, we see that these rectangles can overlap in the d dimension. There is one prefix of length 3 and one rectangle formed using it. So set R_3 has one set of rectangles and that set contains one rectangle $e5$.

Let us assume that the size of the list of ranges d_i^j is denoted by k_i^j. From each list of ranges consisting of d_i^js, we derive a list of non-overlapping intervals D_i^js. The size of this new set D_i^j is $K_i^j <= 2k_i^j + 1$, i.e. by representing the original k_i^j overlapping intervals as non-overlapping intervals we increase the space by only a constant factor of 2. The purpose of replacing overlapping intervals by non-overlapping intervals is to locate the d from the query point into one of these non-overlapping rectangles during the search procedure and then to find the associated enclosing rectangle. Hence, when many intervals overlap a given interval, the rectangle associated with the interval (during the overlap eliminating phase) is the one with the highest priority that overlaps the interval. In Figure 4, the set of intervals that are created after this overlap elimination for d_2^1 is $D_2^1 = \{a_0, a_1, \ldots a_6\}$. Let us also assume that from the set of rectangles R_2^1, rectangle $e3$ has the highest priority. Then this will be the rectangle associated with interval a_2, although there are other rectangles overlapping this range.

The preprocessing phase of the algorithm is as follows:

1. Store the set of prefixes P_i^j using any efficient trie representation.
2. $i := 1$
3. For each prefix, P_i^j, store the list of non-overlapping intervals D_i^js in sorted sequence using either an array or a binary tree.
4. Repeat for all prefix lengths i

Essentially, in the preprocessing step we perform two operations: First, we separate the rectangles based on the prefix length in the s dimension. Then, for each prefix, we project all its associated rectangles to the corresponding axis to obtain first the overlapping intervals d_i^j in the d dimension. From these we create the non-overlapping sets D_i^j. The non-overlapping intervals are created by a scan of the overlapping intervals from lower to higher co-ordinates in the d dimension. The procedure is:

1. do for all i
2. Sort the set of overlapping intervals d_i^j into ascending sequence of their starting points.
3. If an interval starts or ends, generate D_i^j for previous interval. Store the interval and pointer to actions for highest priority rectangle that overlaps this interval. If newly created interval, and its previously stored adjacent interval point to the same rectangle, merge these two intervals. Since a new interval D_i^j is created, at most, when an overlapping interval begins or terminates, the size of this new set D_i^j is $K_i^j <= 2k_i^j + 1$ where k_i^j is the size of the set of overlapping intervals d_i^j.

At the end of the pre-processing step each rectangle is stored in exactly one location on the s-axis, i.e, it is stored in a structure associated with the the prefix used to form this rectangle. Its d range is in sorted sequence within the structure, with the other entries in this structure being the d intervals of other rectangles formed using the same prefix. The set of rectangles associated with a prefix is stored as a list of non-overlapping intervals and requires space only proportional to the size of the set. Only $O(n)$ space is needed to store all the rectangles since each rectangle appears only in one set and therefore the size of the union of all sets is $O(n)$.

The look-up algorithm operates in the following phases.

1. i := 1; solution := nil
2. Let $P_i = \{P_i^1, P_i^2, \ldots, P_i^{n_i}\}$ be the set of all prefixes of length i. Find the prefix P_i^j of length i which matches s determined by the query point. If match P_i^j is found then search in the structure associated with P_i^j to find the non-overlapping interval D_i^m that contains d given by the query point. Solution = rectangle associated with (P_i^j, D_i^m). This is the best solution among all prefixes searched so far.
3. Repeat till all prefix lengths have been searched.

The total execution time of the algorithm as presented is $O(ls \log n)$. This is because the number of iterations of the algorithm in the worst case is equal to the number of possible prefix lengths ls. If use a trie structure (note that the use of other data structures is not precluded) to determine prefixes of each length then total cost in time for determining prefixes is $O(ls)$. The list of D_i^js can be of size $O(n)$. Hence, $O(\log n)$ time is needed to search each list for a matching entry (we will later show this factor can be eliminated for all but one list by cascading the search through these multiple lists). The search could have been started from the longest possible prefix so as to improve average execution times but since we are only interested in worst-case performance this optimization is not used.

Consider the example of Figure reflpexample. Assume that a packet with header $P1 = (0110, 0101)$ arrives. We first find a matching prefix (0) of length 1 and search for enclosing rectangles formed with this prefix. We search the d dimension and we find that rectangle $e1$ is a first candidate solution. Note that rectangles $e1, e6$ are the only rectangles in the set of rectangles with prefixes of length equal to 1. Next, we search using prefixes of length 2. The matching prefix is (01). We now get rectangle $e3$ as a better candidate since the d field of the arriving packet overlaps with the range $a2$, and this rectangle is formed with a longer prefix and $e3$ has higher priority than other rectangles formed with prefixes of equal or lower length. $e3$ becomes the best solution found until now. Finally, we locate a matching prefix (011) of length 3. We search among rectangles formed with this prefix and get $e5$ as the best solution.

The time-complexity of $O(ls \log n)$ obtained with this algorithm can be too large for use in a high-speed router. Assume that the the number of possible prefix lengths $ls = 32$ and that the number of table entries $n = 2^{18} = 256K$. This requires 576 memory accesses in the worst case (the measure of execution time for our algorithms). Hence, the time is prohibitively high. In the next section, we show how the $\log n$ factor can be eliminated, and so reducing the number of memory accesses in the above example to about 50 which makes it practical for high-speed implementation.

5.1 Eliminating the $O(\log n)$ factor in execution time

With a trie implementation, the space requirement of the above look-up scheme is $O(n)$. Furthermore, the order of search of the sets from the lists $R_1, R_2, \cdots, etc.$ is in increasing order of prefix lengths, i.e., a set from R_1 is searched before searching a list from R_2 and so on. The search proceeds in levels with sets belonging to R_1 being on the first level, those in R_2 being on the second level and so on. Let the number of non-overlapping intervals in all of R_1 be N_1, in all of R_2 be N_2 etc. The bottom most level R_{ls} has N_{ls} non-overlapping intervals. Note that the number of overlapping intervals at each level can be $O(n)$. Suppose that we had a serendipitous arrangement of intervals where only the size of R_1 is $O(n)$ and for all $R_i, i > 1$ the sizes are $O(1)$. Clearly, in this case the worst case execution time is $O(ls + \log n)$. Of course, we cannot count on such a good arrangement of intervals. However, we can make this happen by introducing some "virtual" intervals using a technique used in computational geometry to speed up searches in multiple ordered lists [7, 8]

When we perform a search on the list at level say i, the information we get is that the given d lies in an interval D_i^j. When we next search the lists at level $i + 1$, instead of searching through all the intervals, we can use the information learned in the previous search and search amongst only those intervals that fall in the range given by D_i^j. While this may improve the average case performance, unfortunately, the worst case is not affected by this heuristic. This is because at level $i+1$ there may be $O(n/ls)$ intervals which fall within the range determined by D_i^j and this can happen at all levels. Hence, an $O(\log(n/ls)) = O(\log n)$ search may be needed at every level.

Now suppose we introduce "virtual intervals" at levels $i < ls$ in the following manner. There are N_{ls} intervals at level ls. Let us also denote by $y_1^{ls}, y_2^{ls} \ldots$, the boundary points that demarcate the N_{ls} intervals in the d dimension at level ls. There are $2 * N_{ls}$ such points at most. We replicate every other point at level ls to level $ls - 1$, i.e $2 * N_{ls}$ points are moved to level $ls-1$. The points that were propagated together with the points defining the original intervals, define the intervals at level $ls - 1$. These are stored as non-overlapping intervals at level $ls - 1$. Next we take all the intervals now at level $ls - 1$ and their associated points and

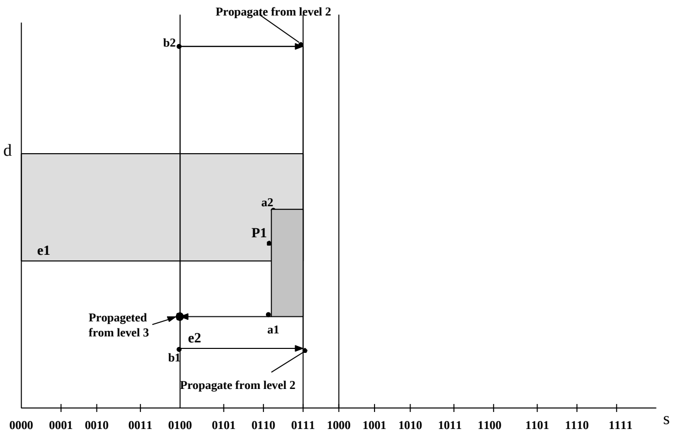

Figure 5: Operation of the 2-dimensional algorithm when one dimension includes only intervals created by prefixes and the propagation technique is used.

replicate every other point and move them as virtual points to level $ls-2$. We repeat this process till we reach the root level. Note that the propagation is only used to speed up the search. At each level, the rectangles associated with each interval are as described in the preprocessing described before. We can ignore the virtual intervals and points that result from propagation as far as the association of rectangles to intervals is concerned.

Note that this propagation process only increases the space requirements by a constant factor, i.e, the total space requirement is still $O(n)$. It can be shown that the maximum amount of virtual intervals created (and hence extra space) is when $N_{ls} = n$, in which case the number of boundary points at level ls is $2n$. The extra space due to the propagations is then

$$2(n + \frac{n}{2} + \frac{n}{4} + \ldots) <= 4n \qquad (1)$$

However, by increasing the space by a constant factor, we gain the advantage that we can search the multiple lists in a more efficient manner. We search the level 1 list as before taking $O(\log n)$ time in the worst case. This results in locating the given d in some interval D_1^j. This interval can possibly be a virtual interval propagated up from the level 2. Now that we have localized d to the interval D_1^j, the search in level 2 need only search in the range given by $D^j{}_1$. Because every other point has been propagated up from level 2, only 2 intervals can fall in range D_1^j to which d has been localized. Hence, the search at level 2 can be done in $O(1)$ time. In general, in moving from level i to level $i+1$, the propagation of intervals ensures that there is enough information gained in the search at level i that the search at level $i+1$ takes only $O(1)$ time. Hence, the worst case execution time of the look-up algorithm is $O(ls + \log n)$.

To illustrate the algorithm, let us consider the example of Figure 5. For illustration purposes, we restrict ourselves to only three squares. We start from the rectangles with the longest prefix and we propagate only point $a1$. As a result, on the axis corresponding to prefix of size two, there are now three points. We propagate points $b1$ and $b2$ only. There are four points now in the axis for prefixes of length 1. Assume that a packet with header $P1$ arrives. During the search operation, we start from the prefixes of length 1 and locate rectangle $e1$ as a candidate solution. When we move to prefixes of size 2, however, we use a pointer to the set of intervals that were possibly propagated. Note, that from the propagation we have lost the information of whether P1 has a d dimension smaller or larger than the point $a1$. But it can only be one of the two solutions. There are two candidate intervals that are retrieved and only one corresponds to the incoming d value. Thus, we use the pointer associated with this interval to continue our search on the prefixes of length three. The final solution can be now retrieved.

6 Concluding Remarks

Packet filtering or classification, using multiple packet-header fields and allowing range matches, has been considered a difficult operation to implement at high-speeds and with a large number of filter rules. However, it is a very useful primitive in connectionless networks for associating a policy-defined context with each incoming packet, so as to permit packet handling using various policy-based routing, security, and differentiated services actions. We presented three new schemes for packet classification. The first two are for implementing generalized packet filters allowing range matches in many dimensions. The schemes allow processing of thousands of filter rules at the rates of millions of packets per second using simple hardware technology and moderate clock speeds. These processing rates are based on traffic and filter-rule-pattern independent worst-case bounds, unlike cache-oriented schemes which are heavily traffic dependent. We are interested in only worst-case performance of the schemes since we want to avoid queueing for header processing in order to use the packet classifier for providing differentiated services and QoS. The third scheme is for the special case of two-dimensional lookups where the ranges in one direction are restricted to being prefix ranges. For this case, we present an algorithm which in the worst case requires only $O(\text{number-of-prefix-lengths} + \log n)$. This scheme allows 2-dimensional classification to be performed with hundreds of thousands of entries at speeds sufficient for operation in network backbones. This two-dimensional lookup has many applications including the important one of supporting multicast. In the multicast case, since the group identifier range is either a specific group identifier or a wildcard range, our algorithm needs only 2 memory accesses beyond what is needed for the longest prefix match needed for unicast forwarding. The ability to filter on thousands of rules in many dimensions, and hundreds of thousands of rules in two dimensions, widens the range of options feasible for evolving the current best-effort Internet to the Internet of the future, capable of providing customized differentiated services. Specifically, our algorithms demonstrate that there may be no need to restrict filtering to the edges or to very simple operations such as using only the Type-of-Service bits in the IP packet header. Contrarily, the whole network, including the backbone, can participate in the enforcement of policies.

Acknowledgements

The authors would like to thank the anonymous reviewers for the their detailed and insightful comments. The authors would also like to thank K. J. Singh and B. Suter who implemented the five-dimensional algorithm in the Bell Labs Router prototype.

References

[1] M.L. Bailey, B.Gopal, M.Pagels, L.L.Peterson, and P.Sarkar. PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First Symposium on Operating Systems Design adn Implementation*, November 1994.

[2] M. De Berg, M. van Kreveld, and J. Snoeyink. Two- and three-dimensional point location in rectangular subdivisions. *Journal of Algorithms*, 18:256–277, 1995.

[3] P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[4] J. Boyle. RSVP Extensions for CIDR Aggregated Data Flows. In *Internet Draft*, http://www.internic.net/internet-drafts/draft-ietf-rsvp-cidr-ext-01.txt, 1997.

[5] B. Chazelle. How to search in history. *Information and Control*, 64:77–99, 1985.

[6] B. Chazelle and J. Friedman. Point location among hyperplanes and unidirectional ray shooting. *Computational Geometry: Theory and Applications*, 4:53–62, 1994.

[7] B. Chazelle and L.J. Guibas. Fractional cascading. i. a data structuring technique. *Algorithmica*, 1(2):133–62, 1986.

[8] B. Chazelle and L.J. Guibas. Fractional cascading. ii. applications. *Algorithmica*, 1(2):163–191, 1986.

[9] K.C. Claffy. *Internet Traffic Characterization*. PhD thesis, University of California, San Diego, 1994.

[10] D. Clark. Service Allocation Profiles. In *Internet Draft*, http://www.internic.net/internet-drafts/draft-clark-diff-svc-alloc-00.txt, 1997.

[11] K.L. Clarkson. New applications of random sampling in computational geometry. *Discrete & Computational Geometry*, 2:195–222, 1987.

[12] H. Edelsbrunner, L.J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15:317–340, 1986.

[13] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast - sparse mode : Protocol specification. In *RFC 2117*, June 1997.

[14] D. Estrin, J. Postel, and Y. Rekhter. Routing arbiter architecture. In *ConneXions*, volume 8, pages 2–7, August 1994.

[15] V. Fuller et. al. Classless Inter-Domain Routing. In *RFC1519, ftp://ds.internic.net/rfc/rfc1519.txt*, June 1993.

[16] J.C.Mogul, R.F.Rashid, and M.J.Accetta. The packet filter: An efficient mechanism for user level network code. Technical Report 87.2, Digital WRL, 1987.

[17] K.Claffy, C. Polyzos, and H.W.Braun. Application of sampling methodologies to network traffic characterization. In *Proceedings of ACM SIGCOMM'93*, pages 194–203, September 1993.

[18] T. Li and Y. Rekhter. Provider Architecture for Differentiated Services and Traffic Engineering (PASTE). In *Internet Draft*, http://www.internic.net/internet-drafts/draft-li-paste-00.txt, 1998.

[19] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical Conference Proceedings*, pages 259–269, Winter 1994.

[20] N. McKeown, V Anantharam, and J. Walrand. Achieving 100% throughput in an input-queued switch. In *Proceedings of INFOCOM'96*, pages 296–302, March 1996.

[21] Mitsubishi, http://www.mitsubishichips.com/eram/eram.htm. *eRAM - Memory and Logic on a chip*, 1997.

[22] M. H. Overmars and A.F. van der Stappen. Range searching and point location among fat objects. *Journal of Algorithms*, 21(3):629–656, 1996.

[23] P. Van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings of 16th IEEE Conference on Foundations of Computer Science*, pages 75–84, 1975.

[24] K. Thomson, G.J. Miller, and R. Wilder. Wide-area traffic patterns and characteristics. *IEEE Network*, December 1997.

[25] Toshiba America Electronic Components. *CMOS dRAMASIC Families*, 1997.

[26] D. Waitzman, C. Partridge, and S. Deering. Distance Vector Multicast Routing Protocol. In *RFC1075, ftp://ds.internic.net/rfc/rfc1075.txt*, June 1993.

[27] M. Yuhara, B.N. Bershad, C.Maeda, J.Eliot, and B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Technical Conference Proceedings*, Winter 1994.

[28] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7(5):8–18, September 1993.