Survey and Taxonomy of Packet Classification Techniques

DAVID E. TAYLOR

Applied Research Laboratory
Washington University in Saint Louis

Packet classification is an enabling function for a variety of Internet applications including quality of service, security, monitoring, and multimedia communications. In order to classify a packet as belonging to a particular flow or set of flows, network nodes must perform a search over a set of filters using multiple fields of the packet as the search key. In general, there have been two major threads of research addressing packet classification, algorithmic and architectural. A few pioneering groups of researchers posed the problem, provided complexity bounds, and offered a collection of algorithmic solutions. Subsequently, the design space has been vigorously explored by many offering new algorithms and improvements on existing algorithms. Given the inability of early algorithms to meet performance constraints imposed by high speed links, researchers in industry and academia devised architectural solutions to the problem. This thread of research produced the most widely-used packet classification device technology, Ternary Content Addressable Memory (TCAM). New architectural research combines intelligent algorithms and novel architectures to eliminate many of the unfavorable characteristics of current TCAMs. We observe that the community appears to be converging on a combined algorithmic and architectural approach to the problem. Using a taxonomy based on the high-level approach to the problem and a minimal set of running examples, we provide a survey of the seminal and recent solutions to the problem. It is our hope to foster a deeper understanding of the various packet classification techniques while providing a useful framework for discerning relationships and distinctions.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Associative memories*; *Cache memories*; C.2.0 [**Computer-Communication Networks**]: General—*Data communications*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Packet-switching networks*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network management*; E.1 [**Data**]: Data Structures—*Lists, stacks, and queues; Trees*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Packet classification, flow identification

1. INTRODUCTION

Packet classification is an enabling function for a variety of Internet applications including quality of service (QoS), security, monitoring, and multimedia communications. Such applications typically operate on packet flows or sets of flows. Therefore,

Author's address: D. E. Taylor, Applied Research Laboratory, Washington University in Saint Louis, and Exegy Inc., St. Louis, MO; e-mail: david@exegy.com.

Table I. Example Filter Set of 16 Filters Classifying on Four Fields (Each filter has an associated flow identifier (*Flow ID*) and priority tag (*PT*) where † denotes a nonexclusive filter; wildcard fields are denoted with ∗)

Filter				Action	
SA	DA	Prot	DP	FlowID	PT
11010010	*	TCP	[3:15]	0	3
10011100	*	∗	[1:1]	1	5
101101*	001110*	∗	[0:15]	2	$8^{\dagger}$
10011100	01101010	UDP	[5:5]	3	2
∗	*	ICMP	[0:15]	4	$9^{\dagger}$
100111*	011010*	*	[3:15]	5	$6^{\dagger}$
10010011	*	TCP	[3:15]	6	3
*	*	UDP	[3:15]	7	$9^{\dagger}$
11101100	01111010	*	[0:15]	8	2
111010*	01011000	UDP	[6:6]	9	2
100110*	11011000	UDP	[0:15]	10	2
010110*	11011000	UDP	[0:15]	11	2
01110010	*	TCP	[3:15]	12	$4^{\dagger}$
10011100	01101010	TCP	[0:1]	13	3
01110010	*	*	[3:3]	14	3
100111*	011010*	UDP	[1:1]	15	4

network nodes must classify individual packets traversing the node in order to assign a flow identifier, *FlowID*. Packet classification entails searching a table of filters which binds a packet to a flow or set of flows and returning the *FlowID* for the highest priority filter or set of filters which match the packet. Note that filters are also referred to as rules in some of the packet classification literature. Likewise, a *FlowID* is synonymous with the action applied to the packet. At a minimum, filters contain multiple match conditions that specify an exact packet header or set of headers and the associated *FlowID* for packets matching all the conditions. The type of match conditions are typically prefixes for IP address fields, an exact value or wildcard for the transport protocol number and flags, and ranges for port numbers. An example filter table is shown in Table I. In this simple example, filters contain match conditions for four packet headers fields: 8-bit source and destination addresses, transport protocol, and a 4-bit destination port number.

Filter priority may be implied by the order of filters in the filter set. Note that the filters in Table I contain an explicit priority tag *PT* and a nonexclusive flag denoted by †. Priority tags allow filter priority to be independent of filter ordering. Nonexclusive flags allow filters to be designated as either exclusive or nonexclusive. Packets may match only one exclusive filter, allowing quality of service and security applications to specify a single action for the packet. Packets may also match several nonexclusive filters, providing support for transparent monitoring and usage-based accounting applications. Note that a parameter may control the number of nonexclusive filters, *r*, returned by the packet classifier. Like exclusive filters, the priority tag is used to select the *r* highest priority nonexclusive filters. Consider an example search through the filter set in Table I for a packet with the following header fields: source address 10011100, destination address 01101010, protocol UDP, destination port 1. This packet matches the filters with *FlowID* 5 and 15 with priority tags 5 and 4, respectively. Since both are exclusive filters, *FlowID* 15 is applied to the packet, assuming that the highest priority level is 0. Given that no nonexclusive filters match the packet, we return only one *FlowID*. Any packet classification technique that supports multiple matches can support nonexclusive filters. However, techniques that employ precomputation to encode priority into the search may preclude their use.

Due to the complexity of the search, packet classification is often a performance bottleneck in network infrastructure; therefore, it has received much attention in the research community. In general, there have been two major threads of research addressing this problem, algorithmic and architectural. A few pioneering groups of researchers posed the problem, provided complexity bounds, and offered a collection of algorithmic solutions [Gupta and McKeown 1999a, 199b; Lakshman and Stiliadis 1998; Srinivasan et al. 1998]. Subsequently, the design space has been vigorously explored by many offering new algorithms and improvements on existing algorithms [Baboescu and Varghese 2001; Feldmann and Muthukrishnan 2000; Woo 2000]. Given the inability of early algorithms to meet the performance constraints discussed in Section 1.1, researchers in industry and academia devised architectural solutions to the problem. This thread of research produced the most widely-used packet classification device technology, Ternary Content Addressable Memory (TCAM) [Kempke and McAuley 1998; Gibson et al. 2000; McAulay and Francis 1993; Montoye 1994].

Some of the most promising algorithmic research embraces the practice of leveraging the statistical structure of filter sets to improve average performance [Gupta and McKeown 1999a; Baboescu and Varghese 2001; Baboescu et al. 2003; Gupta and McKeown 1999b; Singh et al. 2003]. Several algorithms in this class are amenable to high-performance hardware implementation. We discuss these observations in more detail and provide motivation for packet classification on larger numbers of fields in Section 2. New architectural research combines intelligent algorithms and novel architectures to eliminate many of the unfavorable characteristics of current TCAMs [Spitznagel et al. 2003]. We observe that the community appears to be converging on a combined algorithmic and architectural approach to the problem [Spitznagel et al. 2003; Taylor and Turner 2005b; van Lunteren and Engbersen 2003]. In order to lend structure to our discussion, we develop a taxonomy in Section 3 that frames each technique according to its high-level approach to the problem. The presentation of this taxonomy is followed by a survey of the seminal and recent solutions to the packet classification problem. Throughout our presentation, we attempt to use a minimal set of running examples to provide continuity to the presentation and highlight the distinctions among the various solutions.

1.1. Constraints

Computational complexity is not the only challenging aspect of the packet classification problem. Increasingly, traffic in large ISP networks and the Internet backbone travels over links with transmission rates in excess of one billion bits per second (1 Gb/s). Current generation fiber optic links can operate at over 40 Gb/s. The combination of transmission rate and packet size dictate the throughput, the number of packets per second, routers must support. A majority of Internet traffic utilizes the Transmission Control Protocol which transmits 40 byte acknowledgment packets. In the worst case, a router could receive a long stream of TCP acknowledgments, therefore conservative router architects set the throughput target based on the input link rate and 40 byte packet lengths. For example, supporting 10 Gb/s links requires a throughput of 31 million packets per second, per port. Modern Internet routers contain tens to thousands of ports. In such high-performance routers, route lookup and packet classification is performed on a per-port basis.

Many algorithmic solutions to the route lookup and packet classification problems provide sufficient performance on average. Most techniques suffer from poor performance for a pathological search. For example, a technique might employ a decision tree where most paths through the tree are short, however, one path is significantly long. If a sufficiently long sequence of packets that follows the longest path through the tree arrives at the input port of the router, then the throughput is determined by the worst-case search performance. It is this set of worst-case assumptions that imposes the so-called "wire speed requirement" for route lookup and packet classification solutions. In essence, solutions to these search problems are almost always evaluated based on the time it takes to perform a pathological search. In the context of networks that provide performance guarantees, engineering for the worst case logically follows. In the context of the Internet, the protocols make no performance guarantees and provide best-effort service to all traffic. Furthermore, the switching technology at the core of routers cannot handle pathological traffic. Imagine a sufficiently long sequence of packets in which all the packets arriving at the input ports are destined for the same output port. When the buffers in the router ports fill up, it will begin dropping packets. Thus, the wire speed requirement for Internet routers does not logically follow from the high-level protocols or the underlying switching technology; it is largely driven by network management and marketing concerns. Quite simply, it is easier to manage a network with one less source of packet losses and it is easier to sell an expensive piece of network equipment when you don't have to explain the conditions under which the search engines in the router ports will begin backlogging. It is for these reasons that solutions to the route lookup and packet classification problems are typically evaluated by their worst-case performance.

Achieving tens of millions of lookups per second is not the only challenge for route lookup and packet classification search engines. Due to the explosive growth of the Internet, backbone route tables have swelled to over 100k entries. Likewise, the constant increase in the number of security filters and network service applications causes packet classification filter sets to increase in size. Currently, the largest filter sets contain a few thousand filters. However dynamic resource reservation protocols could cause filter sets to swell into the tens of thousands. Scalability to larger table sizes is a crucial property of route lookup and packet classification solutions; it is also a critical concern for search techniques whose performance depends upon the number of entries in the tables.

As routers achieve aggregate throughputs of trillions of bits per second, power consumption becomes an increasingly critical concern. Both the power consumed by the router itself and the infrastructure to dissipate the tremendous heat generated by the router components significantly contribute to the operating costs. Given that each port of high-performance routers must contain route lookup and packet classification devices, the power consumed by search engines is becoming an increasingly important evaluation parameter.

1.2. Avoiding the Problem

One way to accelerate IP packet forwarding is to avoid performing IP lookups and packet classification via emulated circuit switching. Protocols such as IP-Switching and MPLS/Tag-Switching attempt to avoid lookups in the network core by establishing a label-switched path between an ingress and egress router [Newman et al. 1997; Chandranmenon and Varghese 1996; Rekhter et al. 1997; Rosen et al. 2001]. Forwarding, security, and QoS decisions at core routers are simplified to an indexed or exact match lookup on a table of ATM virtual circuit identifiers, tags, or labels depending on the protocol in use. While these protocols have enjoyed limited success, two major issues prevent them from obviating packet classification. First, routers at autonomous system (AS) boundaries are still required to perform packet classification in order to assign labels. Even if these routers are restricted to network edges, increasing bandwidth demands require high-performance packet classification techniques. The second major issue is coordination between multiple autonomous systems. Due to issues like security, trust, resource allocations, and differing views of the network, end-to-end coordination in the Internet is difficult. Terminating and reestablishing connections at AS boundaries requires packet classification by each AS router at the boundary.

2. OBSERVATIONS OF FILTER SET CHARACTERISTICS

Recent efforts to identify better packet classification techniques have focused on leveraging the characteristics of real filter sets for faster searches. While the lower bounds for the general multifield searching problem have been established, observations made in recent packet classification work offer enticing new possibilities to provide significantly better average performance.

Gupta and McKeown [1999a] published a number of observations regarding the characteristics of real filter sets which have been widely cited. Others have performed analyses on real filter sets and published their observations [Baboescu et al. 2003; Baboescu and Varghese 2001; van Lunteren and Engbersen 2003; Woo 2000]. The following is a distillation of observations relevant to our discussion.

—Current filter set sizes are small, ranging from tens of filters to less than 5000 filters. It is unclear if the size limitation is natural or a result of the limited performance and high expense of existing packet classification solutions.

—The protocol field is restricted to a small set of values. In most filter sets, TCP, UDP, and the wildcard are the most common specifications. Other specifications include ICMP, IGMP, (E)IGRP, GRE and IPIP.

—Transport-layer specifications vary widely. Common range specifications for port numbers such as gt 1023 (greater than 1023) suggest that the use of range to prefix conversion techniques may be inefficient.

—The number of unique address prefixes matching a given address is typically five or less.

—The number of filters matching a given packet is typically five or less.

—Different filters often share a number of the same match conditions.

The final observation has been a source of deeper insight and a springboard for several recent contributions in the area. This characteristic arises due to the administrative policies that drive filter construction. Consider a model of filter construction in which the administrator first specifies the communicating hosts or subnetworks (source and destination address prefix pair), then specifies the application (transport-layer specifications). Administrators often must apply a policy regarding an application to a number of distinct subnetwork pairs; hence, multiple filters will share the same transport-layer specification. Likewise, administrators often apply multiple policies to a subnetwork pair; hence, multiple filters will share the same source and destination prefix pair. In general, the observation suggests that the number of intermediate results generated by independent searches on fields or collections of fields may be inherently limited. This observation led to a recently proposed framework for packet classification in network processors [Kounavis et al. 2003]. For example, in the filter table of 16 filters shown in Table I, there are 12 unique address prefix pairs. For any given packet, a maximum of four unique address pairs will match. Likewise, there are 14 unique application specifications (source port, destination port, and protocol) and a maximum of five match any given packet.

Taylor and Turner [2005b] also performed a battery of analyses on real filter sets, focusing on the maximum number of unique match conditions in a filter set and unique combinations of match conditions that match any packet. They also observed two key properties.

—*Match Condition Redundancy*. For each filter field, the number of unique match conditions specified by filters in the filter set is significantly less than the number of filters in the filter set.

—*Matching Set Confinement*. For each filter field, the number of unique match conditions that can be matched by a single packet header field is small and remains small as filter sets scale in size.

The first property logically follows from the last observation in the previous list. The matching set confinement property was also found to extend to combinations of match conditions. Taylor and Turner analyzed every possible combination of fields (every possible combination of two fields, three fields, etc.) and observed that the maximum number of unique combinations of match conditions that match any packet is typically bounded by twice the maximum number of single match conditions. They also observed that the size of the *matching set* remains relatively constant for various filter set sizes. Taylor and Turner also argue that additional fields beyond the standard 5-tuple are relevant and new services and administrative policies will demand that packet classification techniques scale to support additional fields (i.e., more dimensions) beyond the standard 5-tuple. They assert that packet classification techniques must scale to support additional fields while maintaining flexibility in the types of additional matches that may arise with new applications.

It is not difficult to identify applications that could benefit from packet classification on fields in higher-level protocol headers. Consider the following example: an ISP wants to deploy Voice over IP (VoIP) service running over an IPv6/UDP/RTP stack for new IP-enabled handsets and mobile appliances. The ISP also wants to make efficient use of expensive wireless links connecting Base Station Controllers (BSCs) to multiple Base Station Transceivers (BSTs); hence, the ISP would like to use a header compression protocol like Robust Header Compression (ROHC). ROHC is a robust protocol that compresses packet headers for efficient use of wireless links with high loss rates [Bormann et al. 2001]. In order to support this, the BSC must maintain a dynamic filter set which binds packets to ROHC contexts based on fields in the IPv6, UDP, and RTP headers. A total of seven header fields (352 bits) must be examined in order to classify such packets. Matches on ICMP type number, RTP Synchronization Source Identifier (SSRC), and other higher-level header fields are likely to be exact matches. Therefore, the number of unique match conditions matching any packet are at most two, an exact value and the wildcard if present. There may be other types of matches that more naturally suit the application, such as arbitrary bit masks on TCP flags, however, we do not foresee any reasons why the structure of filters with these additional fields will significantly deviate from the observed structure in current filter tables.

3. TAXONOMY

Given the subtle differences in formalizing the problem and the enormous need for good solutions, numerous algorithms and architectures for packet classification have been proposed. Rather than categorize techniques based on their performance, memory requirements, or scaling properties, we present a taxonomy that breaks the design space into four regions based on the high-level approach to the problem. We feel that such a taxonomy is useful as a number of the salient features and properties of a packet classification technique are consequences of the high-level approach. We frame each technique as employing one or a blend of the following high-level approaches to finding the best matching filter or filters for a given packet.

—*Exhaustive Search*. Examine all entries in the filter set.

—*Decision Tree*. Construct a decision tree from the filters in the filter set and use the packet fields to traverse the decision tree.

—*Decomposition*. Decompose the multiple field search into instances of single field searches, perform independent searches on each packet field, then combine the results.

—*Tuple Space*. Partition the filter set according to the number of specified bits in the filters, probe the partitions or a subset of the partitions using simple exact match searches.

Figure 1 presents a visualization of our taxonomy. Several techniques, including a few of the most promising ones, employ more than one approach. This is reflected

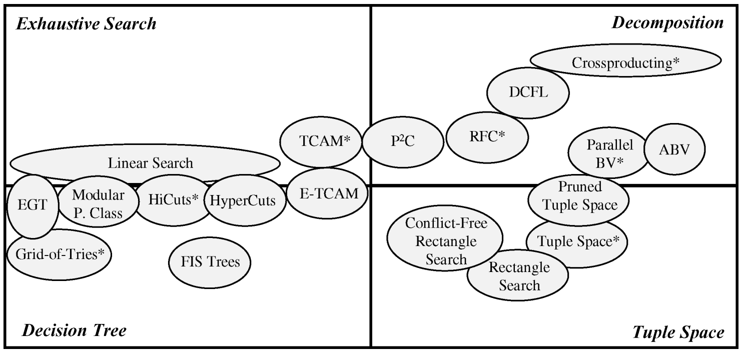


Fig. 1. Taxonomy of multiple field search techniques for packet classification; adjacent techniques are related; hybrid techniques overlap quadrant boundaries; $*$ denotes a seminal technique.

in Figure 1 by overlapping quadrant boundaries. Relationships among techniques are reflected by proximity.

In the following sections, we discuss each high-level approach in more detail along with the performance consequences of each. We also present a survey of the specific techniques using each approach. We note that the choice of high-level approach largely dictates the optimal architecture for high-performance implementation and a number of the scaling properties. Commonly, papers introducing new search techniques focus on clearly describing the algorithm, extracting scaling properties, and presenting some form of simulation results to reinforce baseline performance claims. Seldom is paper real estate devoted to flushing out the details of a high-performance implementation; thus, our taxonomy provides valuable insight into the potential of these techniques. In general, the choice of high-level approach does not preclude a technique from taking advantage of the statistical structure of the filter set thus, we address this aspect of each technique individually.

4. EXHAUSTIVE SEARCH

The most rudimentary solution to any searching problem is simply to search through all entries in the set. For the purpose of our discussion, assume that the set may be divided into a number of subsets to be searched independently. The two most common embodiments of the exhaustive search approach for packet classification are a linear search through a list of filters or a massively parallel search over the set of filters. Interestingly, these two solutions represent the extremes of the performance spectrum where the lowest performance option, linear search, does not divide the set into subsets and the highest performance option, Ternary Content Addressable Memory (TCAM), completely divides the set such that each subset contains only one entry. We will discuss both of these solutions in more detail. The intermediate option of exhaustively searching subsets containing more than one entry is not a common solution, thus we do not discuss it directly. It is important to note that a number of recent solutions using the decision tree approach use a linear search over a bounded subset of filters as the final step. These solutions are discussed in Section 5.

Computational resource requirements for exhaustive search generally scale linearly with the degree of parallelism. Likewise, the realized throughput of the solution is proportional to the degree of parallelism. Linear search requires the minimum amount of computation resources while TCAMs require the maximum, thus linear search and TCAM provide the lowest- and highest-performance exhaustive search techniques, respectively.

Given that each filter is explicitly stored once, exhaustive search techniques enjoy a favorable linear memory requirement, $O(N)$, where N is the number of filters

SA	*DA*	*Prot*
11*	001*	TCP
11*	001*	UDP
11*	101*	TCP
11*	101*	UDP
111*	001*	TCP
111*	001*	UDP
111*	101*	TCP
111*	101*	UDP

	SA
a	11*
b	111*

	DA
a	001*
b	101*

	Prot
a	TCP
b	UDP

filters
(*a*,*a*,*a*)
(*a*,*a*,*b*)
(*a*,*b*,*a*)
(*a*,*b*,*b*)
(*b*,*a*,*a*)
(*b*,*a*,*b*)
(*b*,*b*,*a*)
(*b*,*b*,*b*)

Fig. 2. Example of encoding filters by unique match conditions to reduce storage requirements.

in the filter set. Here we seek to challenge a commonly held view that the $O(N)$ storage requirement enjoyed by these techniques is optimal. We address this issue by considering the match condition redundancy property and the number of fields in a filter. These are vital parameters when considering a third dimension of scaling, filter size. By filter size we mean the number of bits required to specify a filter. A filter using the standard IPv4 5-tuple requires about 168 bits to specify explicitly. With that number of bits, we can specify 2^{168} distinct filters. Typical filter sets contain fewer than 2^{20} filters, suggesting that there is potential for a factor of eight savings in memory.

Here we illustrate a simple encoding scheme that represents filters in a filter set more efficiently than explicitly storing them. Let a filter be defined by fields $f_1 \dots f_d$ where each field f_i requires b_i bits to specify its match condition. For example, a filter may be defined by a source address prefix requiring 64 bits[1], a destination address prefix requiring 64 bits, a protocol number requiring 8 bits, etc. By this definition, the memory requirement for the exhaustive search approach is:

$$N \sum_{i=1}^{d} b_i \qquad (1)$$

Now, let $u_1 \dots u_d$ be the number of unique match conditions in the filter set for each filter field i. If each filter in the filter set contained a unique match condition in each field, then exhaustive search would have an optimal storage requirement. Note that in order for a filter to be unique, it must only differ from each filter in the filter set by one bit. As we discuss in Section 2, there is significant redundancy among filter match conditions. Through efficient encoding, the storage requirement can be reduced from linear in the number of filters, to logarithmic in the number of unique match conditions. Consider the example shown in Figure 2. Note that all 8 filters are unique, however, there are only two unique match conditions for each field for all filters in the filter set. In order to represent the filter set, we only need to store the unique match conditions for each field once. As shown in Figure 2, we assign a locally unique label to each unique match condition. The number of bits required for each label is $\lg(u_i)$, only one bit in our example. Note that each filter in the filter set can now be represented using the labels for its constituent match conditions. Using this encoding technique, the memory requirement becomes:

$$\sum_{i=1}^{d} (u_i \times b_i) + N \sum_{i=1}^{d} \lg u_i \qquad (2)$$

The first term accounts for the storage of unique match conditions and the

[1] We are assuming a 32-bit address where an additional 32 bits are used to specify a mask. There are more efficient ways to represent a prefix, but this is tangential to our argument.

second term accounts for the storage of the encoded filters. The savings factor for a given filter set is simply the ratio of Equation (1) and Equation (2). For simplicity, let $b_i = b \forall i$, and let $u_i = u \forall i$; the savings factor is:

$$\frac{Nb}{ub + N \lg u} \tag{3}$$

In order for the savings factor to be greater than one, the following relationship must hold:

$$\frac{u}{N} + \frac{\lg u}{b} < 1 \tag{4}$$

Note that $u \leq 2^b$, and $u \leq N$. Thus, the savings factor increases as the number of filters in the filter set and the size (number of bits) of match conditions increases relative to the number of unique match conditions. For our simple example in Figure 2, this encoding technique reduces the storage requirement from 1088 bits to 296 bits, or a factor of 3.7. As discussed in Section 2, we anticipate that future filter sets will include filters with more fields. It is also likely that the additional fields will contain a handful of unique match conditions. As this occurs, the linear memory requirement of techniques explicitly storing the filter set will become increasingly suboptimal.

4.1. Linear Search

Performing a linear search through a list of filters has $O(N)$ storage requirements, but it also requires $O(N)$ memory accesses per lookup. For even modest-sized filter sets, linear search becomes prohibitively slow. It is possible to reduce the number of memory accesses per lookup by a small constant factor by partitioning the list into sublists and pipelining the search where each stage searches a sublist. If p is the number of pipeline stages, then the number of memory accesses per lookup is reduced to $O(\frac{N}{p})$, but the computational resource requirement increases by a factor of p. While one could argue that a hardware device with many small embedded memory blocks could provide reasonable performance and capacity, latency increasingly becomes an issue with deeper pipelines and higher link rates. Linear search is a popular solution for the final stage of a lookup when the set of possible matching filters has been reduced to a bounded constant [Gupta and McKeown 1999b; Woo 2000; Singh et al. 2003].

4.2. Ternary Content Addressable Memory (TCAM)

Taking a cue from fully-associative cache memories, Ternary Content Addressable Memory (TCAM) devices perform a parallel search over all filters in the filter set [Montoye 1994]. TCAMs were developed with the ability to store a “Don’t Care” state in addition to a binary digit. Input keys are compared against every TCAM entry, thereby enabling them to retain single clock cycle lookups for arbitrary bit mask matches. TCAMs do suffer from four primary deficiencies: (1) high cost per bit relative to other memory technologies, (2) storage inefficiency, (3) high power consumption, and (4) limited scalability to long input keys. With respect to cost, a current (2005) price check revealed that TCAM costs about 30 times more per bit of storage than DDR SRAM. While it is likely that TCAM prices will fall in the future, it is unlikely that they will be able to leverage the economy of scale enjoyed by SRAM and DRAM technology.

The storage inefficiency comes from two sources. First, arbitrary ranges must be converted into prefixes. In the worst case, a range covering w-bit port numbers may require $2(w-1)$ prefixes. Note that a single filter including two port ranges could require $2(w-1)^2$ entries, or 900 entries for 16-bit port numbers. As shown in Table II, we performed an analysis of 12 real filter sets and found that the *expansion factor*, or ratio of the number of required TCAM entries to the number of filters, ranged from 1.0 to 6.2 with an average of 2.32. This suggests that designers should budget at least seven TCAM entries per filter, compounding the hardware and power inefficiencies described in the following. The second source of storage inefficiency

Table II. Number of Entries Required to Store Filter Set in a Standard TCAM

		TCAM	Expansion
Set	Size	Entries	Factor
acl1	733	997	1.3602
acl2	623	1259	2.0209
acl3	2400	4421	1.8421
acl4	3061	5368	1.7537
acl5	4557	5726	1.2565
fw1	283	998	3.5265
fw2	68	128	1.8824
fw3	184	554	3.0109
fw4	264	1638	6.2045
fw5	160	420	2.6250
ipc1	1702	2332	1.3702
ipc2	192	192	1.0000
Average			2.3211

stems from the additional hardware required to implement the third "Don't Care" state. In addition to the six transistors required for binary digit storage, a typical TCAM cell requires an additional six transistors to store the mask bit and four transistors for the match logic, resulting in a total of 16 transistors and a cell 2.7 times larger than a standard SRAM cell [Montoye 1994]. A circuit diagram of a standard TCAM cell is shown in Figure 3. Some proprietary architectures allow TCAM cells to require as few as 14 transistors [Kempke and McAuley 1998; Gibson et al. 2000].

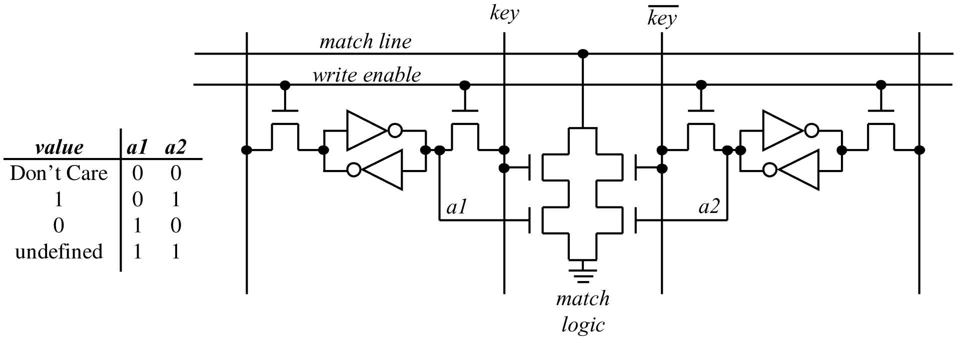
The massive parallelism inherent in TCAM architecture is the source of high power consumption. Each bit of TCAM match logic must drive a match word line which signals a match for the given key. The extra logic and capacitive loading result in access times approximately three times longer than SRAM [SiberCore Technologies Inc. 2002]. Additionally, power consumption per bit of storage is on the order of 3 micro-Watts per bit [Micron Technology Inc. 2003] compared to 20 to 30 nano-Watts per bit for SRAM [Micron Technology Inc. 2002b]. In summary, TCAMs consume 150 times more power per bit than SRAM.

Spitznagel et al. [2003] recently introduced *Extended TCAM* (E-TCAM) which implements range matching directly in hardware and reduces power consumption by over 90% relative to standard TCAM. We discuss E-TCAM in more detail in Section 5.6. While this represents promising new work in the architectural thread of research, it does not address the high cost per bit or scalability issues inherent in TCAMs for longer search keys. TCAM suffers from limited scalability to longer search keys due to its use of the exhaustive search approach. As previously discussed, the explicit storage of each filter becomes more inefficient as filter sizes increase and the number of unique match conditions remains limited. If the additional filter fields require range matches, this effect is compounded due to the previously described inefficiency of mapping arbitrary ranges to prefixes.

5. DECISION TREE

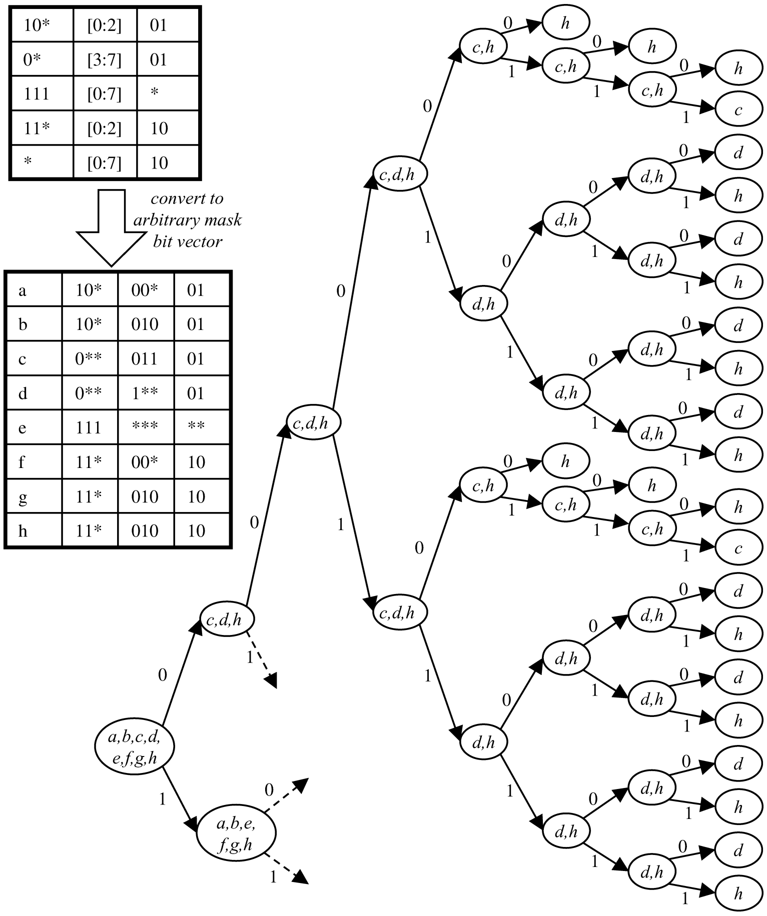
Another popular approach to packet classification on multiple fields is to construct a decision tree where the leaves of the tree contain filters or subsets of filters. In order to perform a search using a decision tree, we construct a search key from the packet header fields. We traverse the decision tree by using individual bits or subsets of bits from the search key to make branching decisions at each node of the tree. The search continues until we reach a leaf node storing the best matching filter or subset of filters. Decision tree construction is complicated by the fact that a filter may specify several different types of searches. The mix of longest prefix match, arbitrary range match, and exact match conditions significantly complicates the branching decisions at each node of the decision tree. A common solution to this problem is to convert match conditions to a single type. Several techniques convert all match conditions to bit vectors with arbitrary bit masks, that is, bit vectors where each bit may be a 1, 0, or $*$ ("Don't Care"). Recall that filters containing arbitrary ranges do not readily map to arbitrary bit masks; therefore, this conversion process results in filter replication. Likewise, the use of wildcards may cause a filter to be stored at many leaves of the decision tree.

To better illustrate these issues, we provide an example of a naïve construction of a decision tree in Figure 4. The five filters



value	*a1*	*a2*
Don't Care	0	0
1	0	1
0	1	0
undefined	1	1

Fig. 3. Circuit diagram of a standard TCAM cell; the stored value (0, 1, Don't Care) is encoded using two registers *a1* and *a2*.



10*	[0:2]	01
0*	[3:7]	01
111	[0:7]	*
11*	[0:2]	10
*	[0:7]	10

a	10*	00*	01
b	10*	010	01
c	0**	011	01
d	0**	1**	01
e	111	***	**
f	11*	00*	10
g	11*	010	10
h	11*	010	10

Fig. 4. Example of a naïve construction of a decision tree for packet classification on three fields; all match conditions are converted to bit vectors with arbitrary bit masks.

in the example set contain three fields: 3-bit address prefix, an arbitrary range covering 3-bit port numbers, and an exact 2-bit value or wildcard. We first convert the five filters into bit vectors with arbitrary bit masks which increases the number of filters to eight. Viewing the construction process as progressing in a depth-first manner, a decision tree path is expanded until the node covers only one filter or the bit vector is exhausted. Nodes at the last level may cover more than one filter if filters overlap. We assume that leaf nodes contain the action to be applied to packets matching the filter or subset of filters covered by the node. Due to the size of the full decision tree, we show a portion of the data structure in Figure 4. If we evaluate this data structure by its ability to distinguish between potentially matching filters for a given packet, we see that this naïve construction is not highly effective. As the reader has most likely observed already, there are numerous optimizations that could allow a decision tree to more effectively distinguish between potentially matching filters. The algorithms and architectures discussed in the following subsections explore these optimizations.

Several of the algorithms that we classify as using a decision tree approach are more commonly referred to as cutting algorithms. These algorithms view filters with d fields as defining d-dimensional rectangles in d-dimensional space, thus a cut in multidimensional space is isomorphic to a branch in a decision tree. The branching decision in a cutting algorithm is typically more complex than examining a single bit in a bit vector. Note that the E-TCAM approach discussed in Section 5.6 employs a variant on the cutting algorithms that may be viewed as a parallel search of several decision trees containing different parts of the filter set. Thus, we view some cutting algorithms as relaxing the constraints on classical decision trees.

Due to the many degrees of freedom in decision tree approaches, the performance characteristics and resource requirements vary significantly among algorithms. In general, lookup time is $O(W)$, where W is the number of bits used to specify the filter. Given that filters classifying on the standard 5-tuple require approximately 168 bits, viable approaches must employ some optimizations in order to meet throughput constraints. The memory requirement for our naïve construction is $O(2^{W+1})$. In general, memory requirements vary widely depending on the complexity of the branching decisions employed by the data structure. One common feature of algorithms employing the decision tree approach is memory access dependency. Stated another way, the decision tree searches are inherently serial; a matching filter is found by traversing the tree from root to leaf. The serial nature of the decision tree approach precludes fully parallel implementations. If an algorithm places a bound on the depth of the decision tree, then implementing the algorithm in a pipelined architecture can yield high throughput. This does require an independent memory interfaces for each pipeline stage.

5.1. Grid-of-Tries

Srinivasan et al. [1998] introduced the seminal *Grid-of-Tries* and *Crossproducting* algorithms for packet classification . In this section, we focus on Grid-of-Tries which applies a decision tree approach to the problem of packet classification on source and destination address prefixes. Crossproducting was one of the first techniques to employ decomposition and we discuss it in Section 6.3. For filters defined by source and destination prefixes, Grid-of-Tries improves upon the directed acyclic graph (DAG) technique introduced by Decasper et al. [1998]. This technique is also called set pruning trees because redundant subtrees can be pruned from the tree by allowing multiple incoming edges at a node. While this optimization does eliminate redundant subtrees, it does not completely eliminate replication as filters may be stored at multiple nodes in the tree. Grid-of-Tries eliminates this replication by storing filters at a single node and using *switch pointers* to direct searches to potentially matching filters.

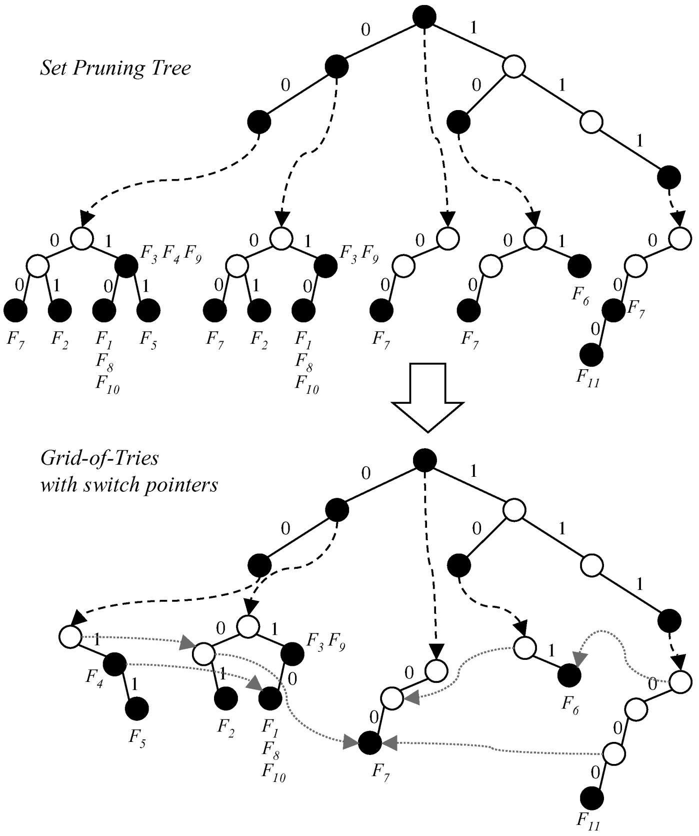


Fig. 5. Example of set pruning trees and Grid-of-Tries classifying on the destination and source address prefixes for the example filter set in Table III.

Figure 5 highlights the differences between set pruning trees and Grid-of-Tries using the example filter set shown in Table III. Note that we have restricted the classification to two fields, the destination address prefix followed by source address prefix. Assume we are searching for the best matching filter for a packet with destination and source addresses equal to 0011. In the Grid-of-Tries structure, we find the longest matching destination address prefix 00∗ and follow the pointer to the source address tree. Since there is no 0 branch at the root node, we follow the switch pointer to the 0∗ node in the source address tree for destination address prefix 0∗. Since there is no branch for 00∗ in this tree, we follow the switch pointer to the 00∗ node in the source address tree for destination address prefix ∗. Here we find a stored filter F_7 which is the best matching filter for the packet.

Table III. Example Filter Set (Port numbers are restricted to be an exact value or wildcard.)

Filter	*DA*	*SA*	*DP*	*SP*	*PR*
F_1	0∗	10∗	∗	80	TCP
F_2	0∗	01∗	∗	80	TCP
F_3	0∗	1∗	17	17	UDP
F_4	00∗	1∗	∗	∗	∗
F_5	00∗	11∗	∗	∗	TCP
F_6	10∗	1∗	17	17	UDP
F_7	∗	00∗	∗	∗	∗
F_8	0∗	10∗	∗	100	TCP
F_9	0∗	1∗	17	44	UDP
F_{10}	0∗	10∗	80	∗	TCP
F_{11}	111∗	000∗	∗	44	UDP

Grid-of-Tries bounds memory usage to $O(NW)$ while achieving a search time of $O(W)$, N is the number of filters and W is the maximum number of bits specified

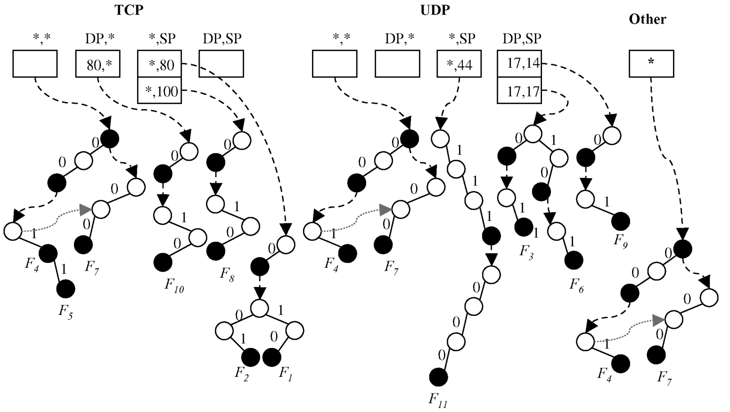


Fig. 6. Example of 5-tuple packet classification using Grid-of-Tries, prefiltering on protocol and port number classes, for the example filter set in Table III.

in the source or destination fields. For the case of searching on IPv4 source and destination address prefixes, the measured implementation used multibit tries sampling 8 bits at a time for the destination trie; each of the source tries started with a 12-bit node, followed by 5-bit trie nodes. This yields a worst case of 9 memory accesses; the authors claim that this could be reduced to 8 with an increase in storage. Memory requirements for 20k filters was around 2MB.

While Grid-of-Tries is an efficient technique for classifying on address prefix pairs, it does not directly extend to searches with additional filter fields. Consider searching the filter set in Table III using the following header fields: destination address 0000, source address 1101, destination port 17, source port 17, protocol UDP. Using the Grid-of-Tries structure in Figure 5, we find the longest matching prefix for the destination address, 00∗, followed by the longest matching prefix for the source address, 11∗. Filter F_5 is stored at this node and there are no switch pointers to continue the search. Since the remaining three fields of F_5 match the packet header, we declare F_5 is the best matching filter. Note that F_3, F_4, and F_9 match. F_3 and F_9 have more specific matches on the port number fields. Clearly, Grid-of-Tries does not directly extend to multiple field searches beyond address prefix matching.

The authors do propose a technique using multiple instances of the Grid-of-Tries structure for packet classification on the standard 5-tuple. The general approach is to partition the filter set into classes based on the tuple defined by the port number fields and protocol fields. An example is shown in Figure 6. Operating under the restriction that port numbers must either be an exact port number or wildcard[2], we first partition the filter set into three classes according to protocol: TCP, UDP, and "other". Filters with a wildcard are replicated and placed into each class. We then partition the filters in the "other" class into subclasses by protocol specification. For each "other" subclass, we construct a Grid-of-Tries. The construction for the TCP and UDP classes are slightly different due to the use of port numbers. For both the UDP and TCP classes, we partition the constituent filters into four subclasses according to the port number tuple: both ports specified; destination port specified, source port wildcard; destination port wildcard, source port specified; both ports wildcard. For each subclass, we construct a hash table storing the unique combinations of port number specifications.

[2] Note that this restriction can be prohibitive for filter sets specifying arbitrary ranges. While filters could be replicated, typical ranges cover thousands of port numbers which induces an unmanageable expansion in the size of the filter set.

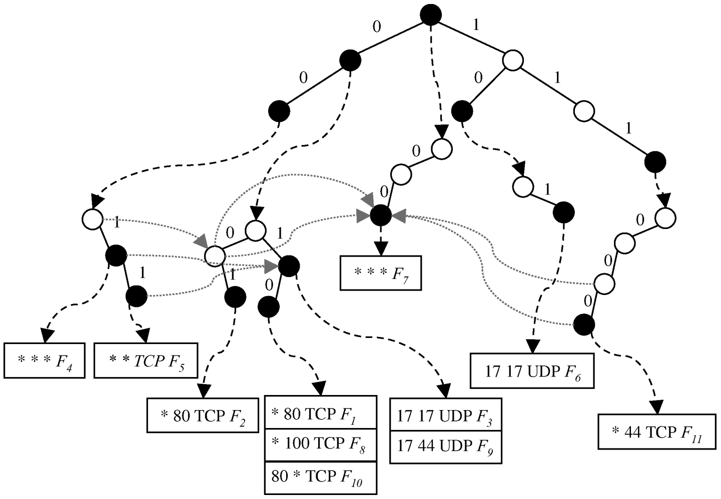


Fig. 7. Example of 5-tuple packet classification using Extended Grid-of-Tries (EGT) for the example filter set in Table III.

Each entry contains a pointer to a Grid-of-Tries constructed from the constituent filters. Ignoring the draconian restriction on port number specifications, this approach may require $O(N)$ separate data structures and filters with a wildcard protocol specification replicated across many of them. It is generally agreed that the great value of the Grid-of-Tries technique lies in its ability to efficiently handle filters classifying on address prefixes.

5.2. Extended Grid-of-Tries (EGT)

Baboescu et al. [2003] proposed *Extended Grid-of-Tries* (EGT) that supports multiple field searches without the need for many instances of the data structure. EGT essentially alters the switch pointers to be jump pointers that direct the search to all possible matching filters, rather than the filters with the longest matching destination and source address prefixes. As shown in Figure 7, EGT begins by constructing a standard Grid-of-Tries using the destination and source address prefixes of all the filters in the filters set. Rather than storing matching filters at source address prefix nodes, EGT stores a pointer to a list of filters that specify the destination and source address prefixes, along with the remaining three fields of the filters. The authors observe that the size of these lists is small for typical *core router* filter sets[3], thus a linear search through the list of filters is a viable option. Note that the jump pointers between source tries direct the search to all possible matching filters. In the worst case, EGT requires $O(W^2)$ memory accesses where W is the address length. Simulated results with core router filter sets show that EGT requires 84 to 137 memory accesses per lookup for filter sets ranging in size from 85 to 2799 filters. Simulated results with synthetically generated filter sets resulted in 121 to 213 memory accesses for filter sets ranging in size from 5k to 10k filters. Memory requirements ranged from 33 bytes per filter to 57 bytes per filter.

5.3. Hierarchical Intelligent Cuttings (HiCuts)

Gupta and McKeown [1999b] introduced a seminal technique called *Hierarchical Intelligent Cuttings* (*HiCuts*). The concept of cutting comes from viewing the packet classification problem geometrically. Each filter in the filter set defines a d-dimensional rectangle in d-dimensional

[3] This property does not necessarily hold for filter sets in other application environments such as firewalls and edge routers.

Table IV. Example Filter Set
(Address field is 4-bits and port ranges cover 4-bit port numbers.)

Filter	Address	Port
a	1010	2 : 2
b	1100	5 : 5
c	0101	8 : 8
d	∗	6 : 6
e	111∗	0 : 15
f	001∗	9 : 15
g	00∗	0 : 4
h	0∗	0 : 3
i	0110	0 : 15
j	1∗	7 : 15
k	0∗	11 : 11

space, where d is the number of fields in the filter. Selecting a decision criteria is analogous to choosing a partitioning, or cutting, of the space. Consider the example filter set in Table IV consisting of filters with two fields: a 4-bit address prefix and a port range covering 4-bit port numbers. This filter set is shown geometrically in Figure 8.

HiCuts preprocesses the filter set in order to build a decision tree with leaves containing a small number of filters bounded by a threshold. Packet header fields are used to traverse the decision tree until a leaf is reached. The filters stored in that leaf are then linearly searched for a match. HiCuts converts all filter fields to arbitrary ranges, avoiding filter replication. The algorithm uses various heuristics to select decision criteria at each node that minimizes the depth of the tree while controlling the amount of memory used.

A HiCuts data structure for the example filter set in Table IV is shown in Figure 9. Each tree node covers a portion of the d-dimensional space and the root node covers the entire space. In order to keep the decisions at each node simple, each node is cut into equal sized partitions along a single dimension. For example, the root node in Figure 9 is cut into four partitions along the *Address* dimension. In this example, we have set the thresholds such that a leaf contains at most two filters, and a node may contain at most four children. A geometric representation of the partitions created by the search tree are shown in Figure 10. The authors describe a number of more sophisticated heuristics and optimizations for minimizing the depth of the tree and the memory resource requirement.

Experimental results in the two-dimensional case show that a filter set of 20k filters requires 1.3MB with a tree depth of 4 in the worst case and 2.3 on average. Experiments with four-dimensional classifiers used filter sets ranging in size from approximately 100 to 2000 filters. Memory consumption ranged from less than 10KB to 1MB, with associated worst-case tree depths of 12 (20 memory accesses). Due to the considerable preprocessing required, this scheme does not readily support incremental updates. Measured update times ranged from 1ms to 70ms.

5.4. Modular Packet Classification

Woo [2000] independently applied the same approach as HiCuts and introduced a flexible framework for packet classification based on a multistage search over ternary strings representing the filters. The framework contains three stages: an index jump table, search trees, and filter buckets. An example data structure for the filter set in Table IV is shown in Figure 11. A search begins by using selected bits of the input packet fields to address the index jump table. If the entry contains a valid pointer to a search tree, the search continues starting at the root of the search tree. Entries without a search tree pointer store the action to apply to matching packets. Each search tree node specifies the bits of the input packet fields to use in order to make a branching decision. When a filter bucket is reached, the matching filter is selected from the set in the bucket via linear search, binary search, or CAM. A key assumption is that every filter can be expressed as a ternary string of 1's, 0's, and ∗'s which represent don't care bits. A filter containing prefix matches on each field is easily expressed as a ternary string by concatenating the fields of the filter; however, a filter containing arbitrary ranges may require replication. Recall that standard

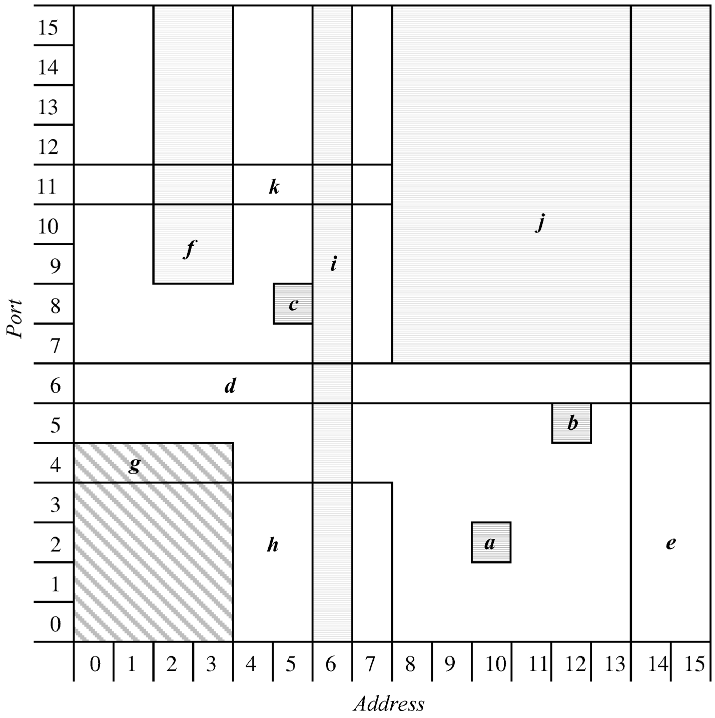


Fig. 8. Geometric representation of the example filter set shown in Table IV.

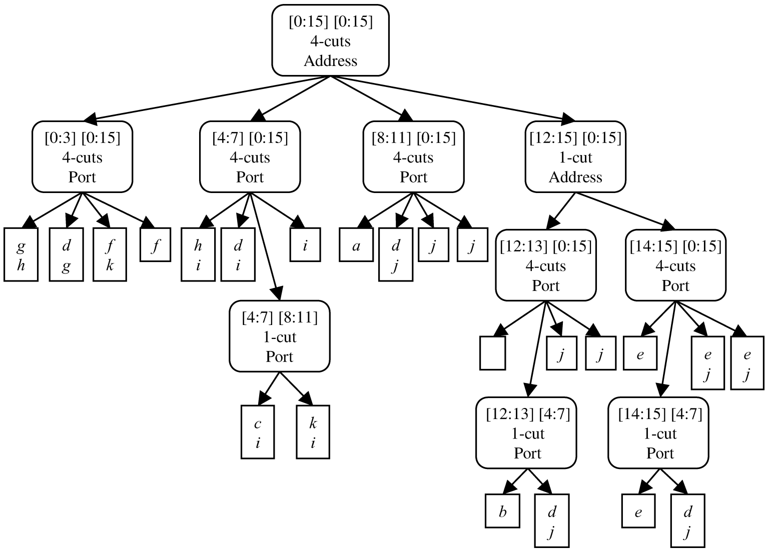


Fig. 9. Example HiCuts data structure for example filter set in Table IV.

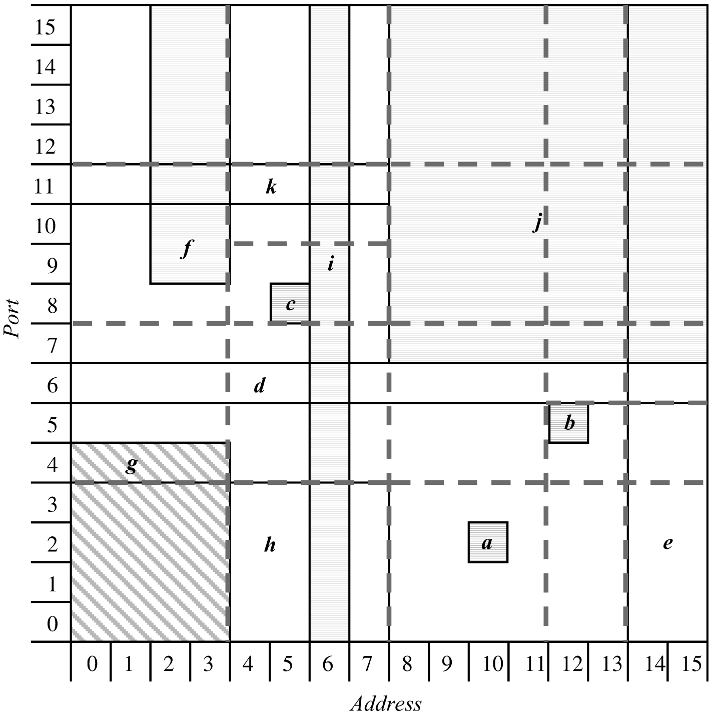


Fig. 10. Geometric representation of partitioning created by HiCuts data structure shown in Figure 9.

Filter	B(7:0)
a	1010 0010
b	1100 0101
c	0101 1000
d	**** 0110
e	111* ****
*f*1	001* 1001
*f*2	001* 101*
*f*3	001* 11**
*g*1	00** 00**
*g*2	00** 0100
h	0*** 00**
i	0110 ****
*j*1	1*** 0111
*j*2	1*** 1***
k	0*** 1011

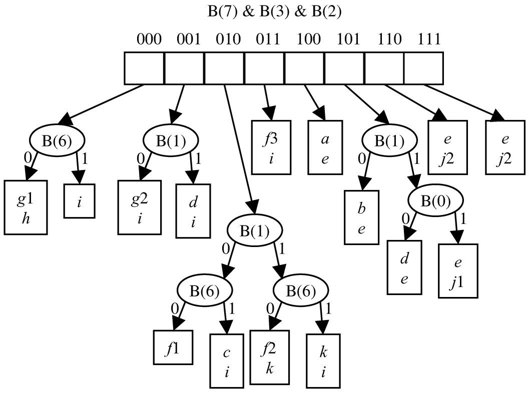


Fig. 11. Modular packet classification using ternary strings and a three-stage search architecture.

5-tuple filters may contain arbitrary ranges for each of the two 16-bit transport port numbers. Hence, a single filter may yield 900 filter strings in the worst case.

The first step in constructing the data structures is to convert the filters in the filter into ternary strings, and organize them in an $n \times m$ array where the number of rows n is equal to the number of ternary strings, and the number of columns m is equal to the number of bits in each string. Each string has an associated weight W_i which is proportional to its frequency of use relative to the other strings; more frequently, matching filter strings will have a larger weight. Next, the bits used to address the index jump table are selected. For our example in Figure 11, we create a 3-bit index concatenate from bits 7, 3, and 2 of the ternary search strings. Typically, the bits used for the jump table address are selected such that every filter specifies those bits. When filters contain don't cares in jump table address bits, it must be stored in all search trees associated with the addresses covered by the jump index. For each entry in the index jump table that is addressed by at least one filter, a search tree is constructed. In the general framework, the search trees may examine any number of bits at each node in order to make a branching decision. Selection of bits is made based on a weighted average of the search path length where weights are derived from the filter weights W_i. This attempts to balance the search tree while placing more frequently accessed filter buckets nearer to the root of the search tree. Note that our example in Figure 11 does not reflect this weighting scheme. Search tree construction is performed recursively until the number of filters at each node falls below a threshold for filter bucket size, usually 128 filters or less. We set the threshold to two filters in our example. The construction algorithm is greedy in that it performs local optimizations.

Simulation results with synthetically generated filter sets show that memory scales linearly with the number of filters. For 512k filters and a filter bucket size of 16, the depth of the search tree ranged from 11 levels to 35 levels, and the number of filter buckets ranged from 76k to 350k depending on the size of the index jump table. Note that larger index jump tables decrease tree depth at the cost of increasing the number of filter buckets due to filter replication.

5.5. HyperCuts

Introduced by Singh et al. [2003], the *HyperCuts* algorithm improves upon the HiCuts algorithm developed by Gupta and McKeown [1999b] and also shares similarities with the Modular Packet Classification algorithms introduced by Woo [2000]. In essence, HyperCuts is a decision tree algorithm that attempts to minimize the depth of the tree by selecting *multiple* cuts in multidimensional space that partition the filter set into lists of bounded size. By forcing cuts to create uniform regions, HyperCuts efficiently encodes pointers using indexing, which allows the data structure to make multiple cuts in multiple dimensions without a significant memory penalty.

According to reported simulation results, traversing the HyperCuts decision tree required between 8 and 32 memory accesses for real filter sets ranging in size from 85 to 4740 filters, respectively. Memory requirements for the decision tree ranged from 5.4 bytes per filter to 145.9 bytes per filter. For synthetic filter sets ranging in size from 5000 to 20000 filters, traversing the HyperCuts decision tree required between 8 and 35 memory accesses, while memory requirements for the decision tree ranged from 11.8 to 30.1 bytes per filter. The number of filters and encoding of filters in the final lists are not provided and so it is difficult to assess the additional time and space requirements for searching the lists at the leaves of the decision tree. HyperCuts's support for incremental updates are not specifically addressed. While it is conceivable that the data structure can easily support a moderate rate of randomized updates, it appears that an adversarial worst-case stream of updates can either create an arbitrarily deep decision tree or force a significant restructuring of the tree.

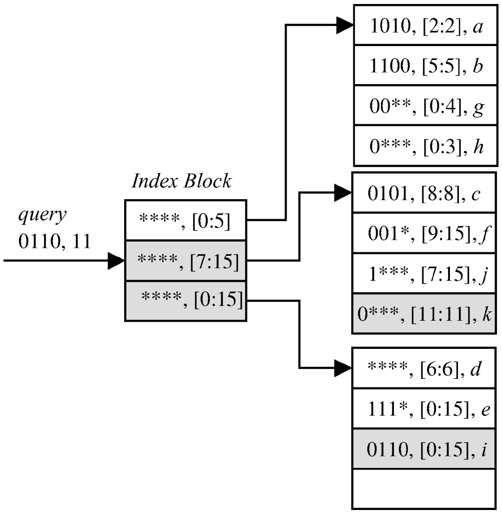


Fig. 12. Example of searching the filter set in Table IV utilizing an Extended TCAM (E-TCAM) using a two-stage search and a filter block size of four.

5.6. Extended TCAM (E-TCAM)

Spitznagel et al. [2003] recently introduced *Extended TCAM* (E-TCAM) to address two of the primary inefficiencies of Ternary Content-Addressable Memory (TCAM): power consumption and storage inefficiency. Recall that in standard TCAM, a single filter, including two port ranges, requires up to $2(w-1)^2$ entries where w is the number of bits required to specify a point in the range. Thus, a single filter with two fields specifying ranges on 16-bit port numbers requires 900 entries in the worst case. The authors found that storage efficiency of TCAMs for real filter sets ranges from 16% to 53% and thus, the average filter occupies between 1.8 and 6.2 TCAM entries. By implementing range matching directly in hardware, E-TCAM avoids this storage inefficiency at the cost of a small increase in hardware resources. When implemented in standard CMOS technology, a range matching circuit requires $44w$ transistors. This is considerably more than the $16w$ transistors required for prefix matching; however, the total hardware resources saved by eliminating the expansion factor for typical packet filter sets far outweighs the additional cost per bit for hardware range matching. Storing a filter for the standard IPv4 5-tuple requires approximately 18% more transistors per entry. This is a small increase relative to the 180% to 620% incurred by filter replication.

Given a query word, TCAMs compare the query word against every entry word in the device. This massively parallel operation results in high power consumption. E-TCAM reduces power consumption by limiting the number of active regions of the device during a search. The second architectural extension of E-TCAM is to partition the device into blocks that may be independently activated during a query. Realistic implementations would partition the device into blocks capable of storing hundreds of filters. In order to group filters into blocks, E-TCAM uses a multiphase partitioning algorithm similar to the previously discussed cutting algorithms. The key differences in E-TCAM are that the depth of the decision tree used for the search is strictly limited by the hardware architecture, and a query may search several branches of the decision tree in parallel. Figure 12 shows an example of an E-TCAM architecture and search using the example filter set in Table IV.

In this simple example, filter blocks may store up to four filters and the decision tree depth is limited to two levels. The first stage of the search queries the *index block* which contains one entry for each group created by the partitioning algorithm. For each phase of the partitioning algorithm except the last phase, a group is defined which completely contains at most b filters where b is the block size. Filters overlapping the group boundaries are not included in the group. The final phase of the algorithm includes such overlapping filters in the group. The number of phases determines the number of *index* entries that may match a query, and hence the number of filter blocks that need to be searched. A geometric representation of the groupings created for our

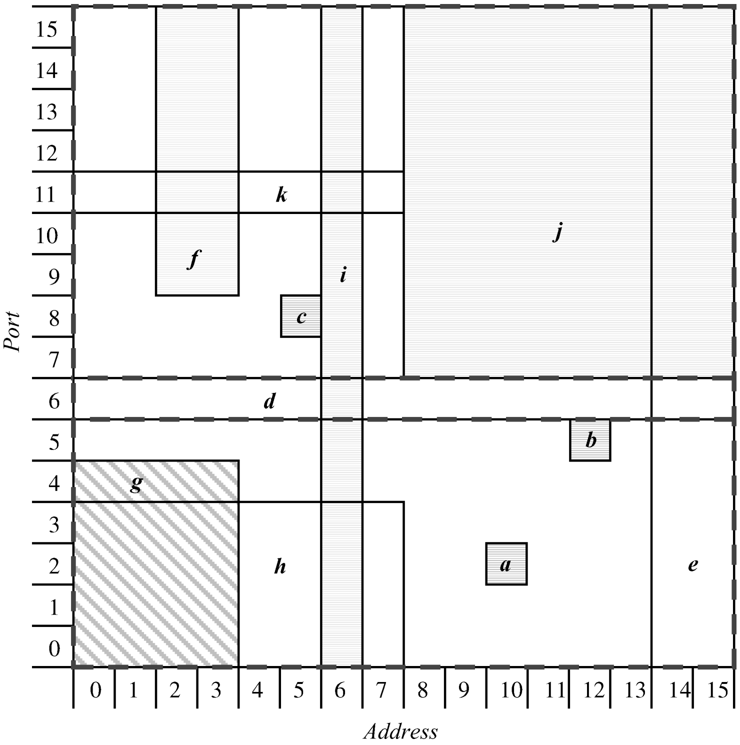


Fig. 13. Example of partitioning the filter set in Table IV for an Extended TCAM (E-TCAM) with a two-stage search and a filter block size of four.

example is shown in Figure 13. Returning to our example in Figure 12, the matching entries in the index block activate the associated filter blocks for the next stage of the search. In this case, two filter blocks are active. Note that all active filter blocks are searched in parallel. Thus, with a pipelined implementation, E-TCAM can retain single-cycle lookups. Simulations show that E-TCAM requires less than five percent of the power required by regular TCAM. Also note that multistage index blocks can be used to further reduce power consumption and provide finer partitioning of the filter set.

5.7. Fat Inverted Segment (FIS) Trees

Feldmann and Muthukrishnan [2000] introduced another framework for packet classification using independent field searches on *Fat Inverted Segment* (*FIS*) *Trees*. Like the previously discussed cutting algorithms, FIS Trees utilize a geometric view of the filter set and map filters into d-dimensional space. As shown in Figure 14, projections from the edges of the d-dimensional rectangles specified by the filters define elementary intervals on the axes. In this case, we form elementary intervals on the *Address* axis. Note that we are using the example filter set shown in Table IV where filters contain two fields: a 4-bit address prefix and a range covering 4-bit port numbers. N filters will define a maximum of $I = (2N+1)$ elementary intervals on each axis. An FIS Tree is a balanced t-ary tree with l levels that stores a set of segments or ranges. Note that $t = (2I + 1)^{1/l}$ is the maximum number of children a node may have. The leaf nodes of the tree correspond to the elementary intervals on the axis. Each node in the tree stores a canonical set of ranges such that the union of the canonical sets at the nodes visited on the path from the

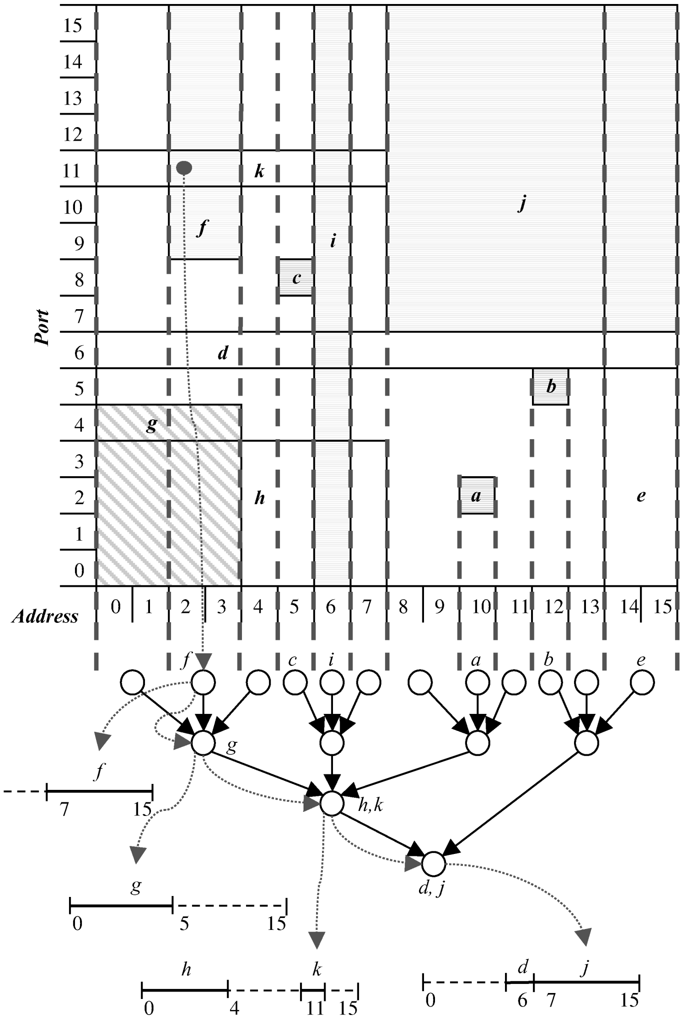


Fig. 14. Example of Fat Inverted Segment (*FIS*) *Trees* for the filter set in Table IV.

leaf node associated with the elementary interval covering a point p to the root node is the set of ranges containing p.

As shown in Figure 14, the framework starts by building a FIS Tree on one axis. For each node with a nonempty canonical set of filters, we construct a FIS Tree for the elementary intervals formed by the projections of the filters in the canonical set on the next axis (filter field) in the search. Note that an FIS Tree is not necessary for the last packet field. In this case, we only need to store the left-endpoints of the elementary intervals and the highest priority filter covering the elementary interval. The authors propose a method to use a longest prefix matching technique to locate the elementary interval covering a given point. This method requires at most $2I$ prefixes.

Figure 14 also provides an example search for a packet with address 2, and port number 11. A search begins by locating the elementary interval covering the first packet field; interval [2 : 3] on the *Address* axis in our example. The search proceeds by following the parent pointers in the FIS Tree from leaf to root node. Along the path, we follow pointers to the sets of elementary intervals formed by the port projections and search for the covering interval. Throughout the search, we remember the highest priority matching filter. Note that the basic framework requires a significant amount of precomputation due to its use of elementary intervals. This property does not readily support dynamic updates at high rates. The authors propose several data structure augmentations to allow dynamic updates. We do not discuss these sophisticated augmentations but do point out that they incur a performance penalty. The authors performed simulations with real and synthetic filter sets containing filters classifying on source and destination address prefixes. For filter sets ranging in size from 1k to 1M filters, memory requirements ranged from 100 to 60 bytes per filter. Lookups required between 10 and 21 cache-line accesses which amounts to 80 to 168 word accesses, assuming 8 words per cache line.

6. DECOMPOSITION

Given the wealth of efficient single-field search-techniques, decomposing a multiple-field search problem into several instances of a single-field search problem is a viable approach. Employing this high-level approach has several advantages. First, each single-field search engine operates independently, thus we have the opportunity to leverage the parallelism offered by modern hardware. Performing each search independently also offers more degrees of freedom in optimizing each type of search on the packet fields. While these are compelling advantages, decomposing a multifield-search problem raises subtle issues.

The primary challenge in taking this high-level approach lies in efficiently aggregating the results of the single-field searches. Many of the techniques discussed in this section use an encoding of the filters to facilitate result aggregation. Due to the freedom in choosing single-field search techniques and filter encodings, the resource requirements and achievable performance vary drastically among the constituent techniques—even more so than with decision tree techniques. Limiting and managing the number of intermediate results returned by single-field search engines is also a crucial design issue for decomposition techniques. Single field search engines often must return more than one result because packets may match more than one filter. As was highlighted by the previous discussion of using Grid-of-Tries for filters with additional port and protocol fields, it is not sufficient for single-field search engines to simply return the longest matching prefix for a given filter field. The best matching filter may contain a field which is not necessarily the longest matching prefix relative to other filters; it may be more specific or higher priority in other fields. As a result, techniques employing decomposition tend to leverage filter set characteristics that allow them to limit the number of intermediate results. In general, solutions using decomposition tend to provide the high throughput due to their amenability to parallel hardware implementations. The high level of lookup performance often comes at the cost of memory inefficiency and, hence, capacity constraints.

6.1. Parallel Bit-Vectors (BV)

Lakshman and Stiliadis [1998] introduced one of the first multiple-field packet classification algorithms targeted to a hardware implementation. Their seminal technique is commonly referred to as the Lucent bit-vector scheme or *Parallel Bit-Vectors* (*BV*). The authors make the initial assumption that the filters may be sorted according to priority. Like the previously discussed cutting algorithms, Parallel BV utilizes a geometric view of the filter set and maps filters into d-dimensional space. As shown in Figure 15, projections from the edges of the d-dimensional rectangles specified by the filters define elementary intervals on the axes. Note that we are using the example filter set shown in Table IV where filters contain two fields, a 4-bit address prefix and a range covering 4-bit port numbers. N filters will define a maximum of $(2N + 1)$ elementary intervals on each axis.

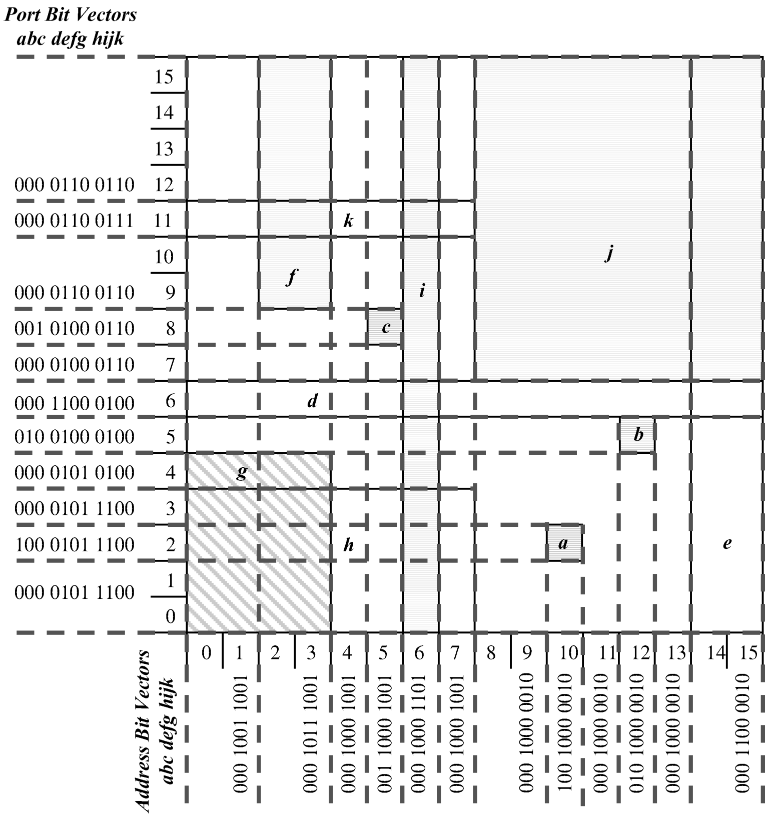


Fig. 15. Example of bit-vector construction for the Parallel Bit-Vectors technique using the filter set shown in Table IV.

For each elementary interval on each axis, we define an N-bit bit-vector. Each bit position corresponds to a filter in the filter set, sorted by priority. All bit-vectors are initialized to all "0"s. For each bit-vector, we set the bits corresponding to the filters that overlap the associated elementary interval. Consider the interval [12 : 15] on the *Port* axis in Figure 15. Assume that sorting the filters according to priority places them in alphabetical order. Filters e, f, i, and j overlap this elementary interval; therefore, the bit-vector for that elementary interval is 00001100110 where the bits correspond to filters a through k in alphabetical order. For each dimension d, we construct an independent data structure that locates the elementary interval covering a given point, then returns the bit-vector associated with that interval. The authors utilize binary search, but any range location algorithm is suitable.

Once we compute all the bit-vectors and construct the d data structures, searches are relatively simple. We search the d data structures with the corresponding packet fields independently. Once we have all d bit vectors from the field searches, we simply perform the bit-wise *AND* of all the vectors. The most significant "1" bit in the result denotes the highest priority matching filter. Multiple matches are easily supported by examining the most significant set of bits in the resulting bit vector.

The authors implemented a five-field version in an FPGA operating at 33MHz with five 128Kbyte SRAMs. This configuration supports 512 filters and performs one million lookups per second. Assuming a binary search technique over the elementary intervals, the general

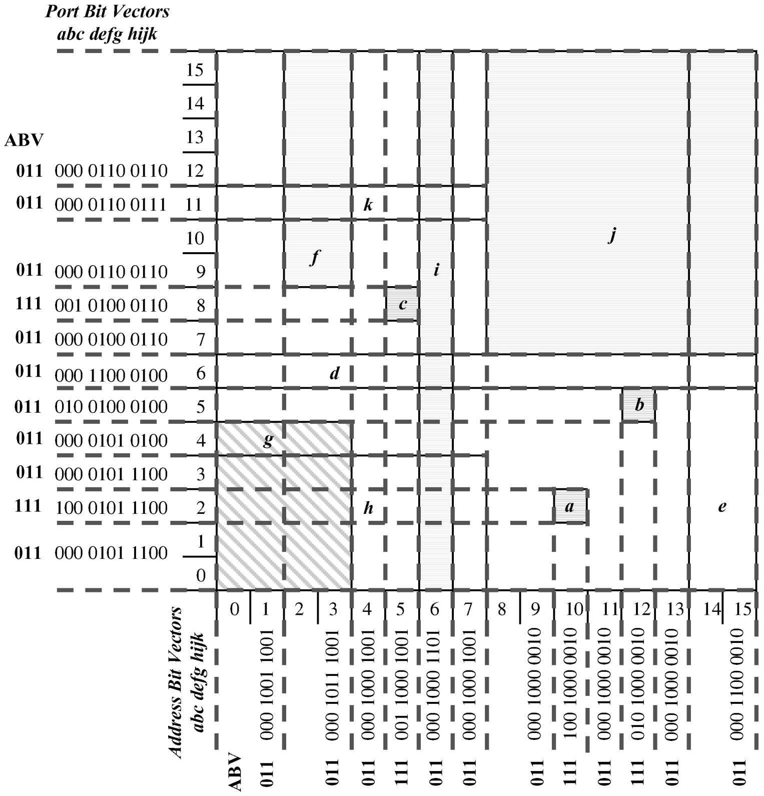


Fig. 16. Example of bit vector and aggregate bit vector construction for the Aggregated Bit-Vectors technique using the filter set shown in Table IV.

Parallel BV approach has $O(\lg N)$ search time and a rather unfavorable $O(N^2)$ memory requirement. The authors propose an algorithm to reduce the memory requirement to $O(N \log N)$ using incremental reads. The main idea behind this approach is to store a single bit vector for each dimension and a set of N pointers of size $\log N$ that record the bits that change between elementary intervals. This technique increases the number of memory accesses by $O(N \log N)$. The authors also propose a technique optimized for classification on source and destination address prefixes only, which we do not discuss here.

6.2. Aggregated Bit-Vector (ABV)

Baboescu and Varghese [2001] introduced the *Aggregated Bit-Vector* (*ABV*) algorithm which seeks to improve the performance of the Parallel BV technique by leveraging statistical observations of real filter sets. ABV converts all filter fields to prefixes, hence it incurs the same replication penalty as TCAMs which we described in Section 4.2. Conceptually, ABV starts with d sets of N-bit vectors constructed in the same manner as in Parallel BV. The authors leverage the widely-known property that the maximum number of filters matching a packet is inherently limited in real filter sets. This property causes the N-bit vectors to be sparse. In order to reduce the number of memory accesses, ABV essentially partitions the N-bit vectors into A chunks and only retrieves chunks containing "1" bits. Each chunk is $\lceil \frac{N}{A} \rceil$ bits in size. Each chunk has an associated bit in an A-bit aggregate bit vector. If any of the bits in the chunk are set to "1", then the corresponding bit in the aggregate bit vector is set to "1". Figure 16 provides an example using the filter set in Table IV.

Each independent search on the d packet fields returns an A-bit aggregate bit vector. We perform the bit-wise *AND* on the aggregate bit vectors. For each "1" bit in the resulting bit vector, we retrieve the d chunks of the original N-bit bit vectors from memory and perform a bit-wise *AND*. Each "1" bit in the resulting bit vector denotes a matching filter for the packet. ABV also removes the strict priority ordering of filters by storing each filter's priority in an array. This allows us to reorder the filters in order to cluster "1" bits in the bit vectors. This in turn reduces the number of memory accesses. Simulations with real filter sets show that ABV reduced the number of memory accesses relative to Parallel BV by a factor of a four. Simulations with synthetic filter sets show more dramatic reductions of a factor of 20 or more when the filter sets do not contain any wildcards. As wildcards increase, the reductions become much more modest.

6.3. Crossproducting

In addition to the previously described Grid-of-Tries algorithm, Srinivasan et al. [1998] also introduced the seminal Crossproducting technique. Motivated by the observation that the number of unique field specifications is significantly less than the number of filters in the filter set, Crossproducting utilizes independent field searches, then combines the results in a single step. For example, a filter set containing 100 filters may contain only 22 unique source address prefixes, 17 unique destination address prefixes, 11 unique source port ranges, etc. Crossproducting begins by constructing d sets of unique match conditions. For example, all of the destination address prefixes from all the filters in the filter set comprise a set, all the source address prefixes comprise a set, etc. Next, we construct independent data structures for each set that return a single best matching entry for a given packet field. In order to resolve the best matching filter for the given packet from the set of best matching entries for each field, we construct a table of crossproducts. In essence, we precompute the best matching filter for every possible combination of results from the d field searches. We locate the best matching filter for a given packet by using the concatenation of results from the independent lookups as a hash probe into the crossproduct table; thus, 5-tuple classification only requires five independent field searches and a single probe to a table of crossproducts. We provide a simple example for a filter set with three fields in Figure 17. Note that the full crossproduct table is not shown due to space constraints.

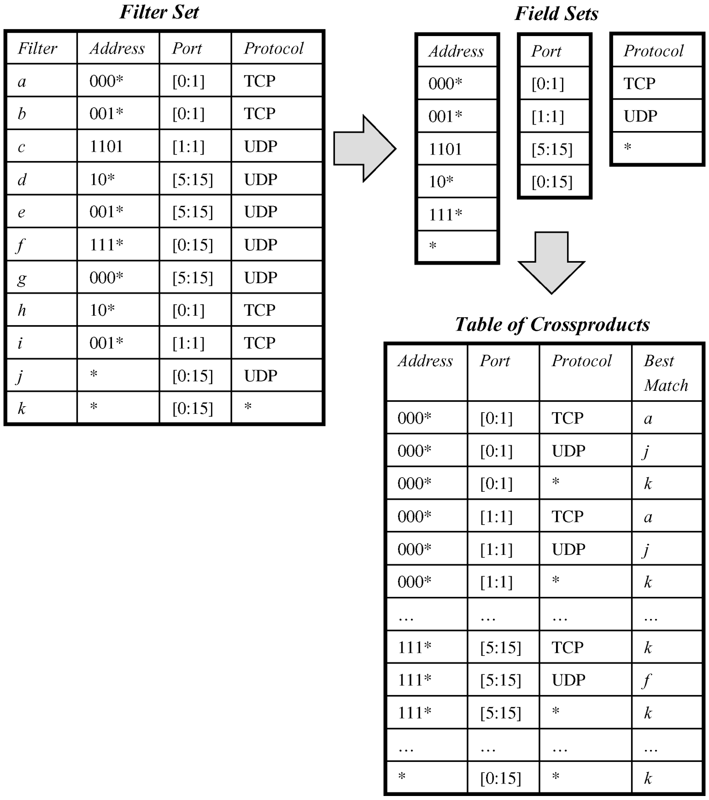
Given a parallel implementation, Crossproducting can provide high throughput, however it suffers from exponential memory requirements. For a set of N filters containing d fields each, the size of the crossproduct table can grow to $O(N^d)$. To keep a bound on the table size, the authors propose *On-demand Crossproducting* which places a limit on the size of the crossproduct table and treats it like a cache. If the field lookups produce a result without an entry in the crossproduct table of limited size, then we compute the crossproduct from the filter set and store it in the table[4]. The performance of this scheme largely depends upon locality of reference.

Finally, the authors propose a combined scheme that seeks to leverage the strengths of both Grid-of-Tries and Crossproducting. The scheme utilizes Grid-of-Tries to perform the destination, then source prefix matches and Crossproducting for ports and flags. The search terminates as soon as a matching filter is found. This assumes that the most specific filters are the highest priority and that a nonoverlapping filter set can be constructed. Using optimistic assumptions regarding caching, the authors claim that a full filter match requires a worst case of 12 memory accesses.

6.4. Recursive Flow Classification (RFC)

Leveraging many of the same observations, Gupta and McKeown [1999a]

[4] Cache entry replacement algorithms can be used to decide which entry to overwrite in the on-demand crossproduct table.



Filter Set

Filter	Address	Port	Protocol
a	000*	[0:1]	TCP
b	001*	[0:1]	TCP
c	1101	[1:1]	UDP
d	10*	[5:15]	UDP
e	001*	[5:15]	UDP
f	111*	[0:15]	UDP
g	000*	[5:15]	UDP
h	10*	[0:1]	TCP
i	001*	[1:1]	TCP
j	*	[0:15]	UDP
k	*	[0:15]	*

Field Sets

Address
000*
001*
1101
10*
111*
*

Port
[0:1]
[1:1]
[5:15]
[0:15]

Protocol
TCP
UDP
*

Table of Crossproducts

Address	Port	Protocol	Best Match
000*	[0:1]	TCP	*a*
000*	[0:1]	UDP	*j*
000*	[0:1]	*	*k*
000*	[1:1]	TCP	*a*
000*	[1:1]	UDP	*j*
000*	[1:1]	*	*k*
…	…	…	…
111*	[5:15]	TCP	*k*
111*	[5:15]	UDP	*f*
111*	[5:15]	*	*k*
…	…	…	…
*	[0:15]	*	*k*

Fig. 17. Example of Crossproducting technique for filter set with three fields; full crossproduct table is not shown due to space constraints.

introduced *Recursive Flow Classification* (*RFC*) which provides high lookup rates at the cost of memory inefficiency. The authors introduced a unique high-level view of the packet classification problem. Essentially, packet classification can be viewed as the reduction of an m-bit string defined by the packet fields to a k-bit string specifying the set of matching filters for the packet or action to apply to the packet. For classification on the IPv4 5-tuple, m is 104 bits and k is typically on the order of 10 bits. The authors also performed a rather comprehensive and widely cited study of real filter sets and extracted several useful properties. Specifically, they noted that filter overlap and the associated number of distinct regions created in multidimensional space is much smaller than the worst case of $O(n^d)$. For a filter set with 1734 filters, the number of distinct overlapping regions in four-dimensional space was found to be 4316, as compared to the worst case which is approximately 10^{13}.

Similar to the Crossproducting technique, RFC performs independent, parallel searches on chunks of the packet header, where chunks may or may not correspond to packet header fields. The results of the chunk searches are combined in multiple phases rather than a single step as in Crossproducting. The result of each chunk lookup and aggregation step in RFC is an equivalence class identifier, *eqID*, that represents the set of potentially matching filters for the packet. The number of eqIDs in RFC depends upon the number of distinct sets of filters that can be matched by a packet. The number of eqIDs in an aggregation step scales with the number of unique overlapping regions formed by filter projections. An example of assigning eqIDs is shown in Figure 18. In

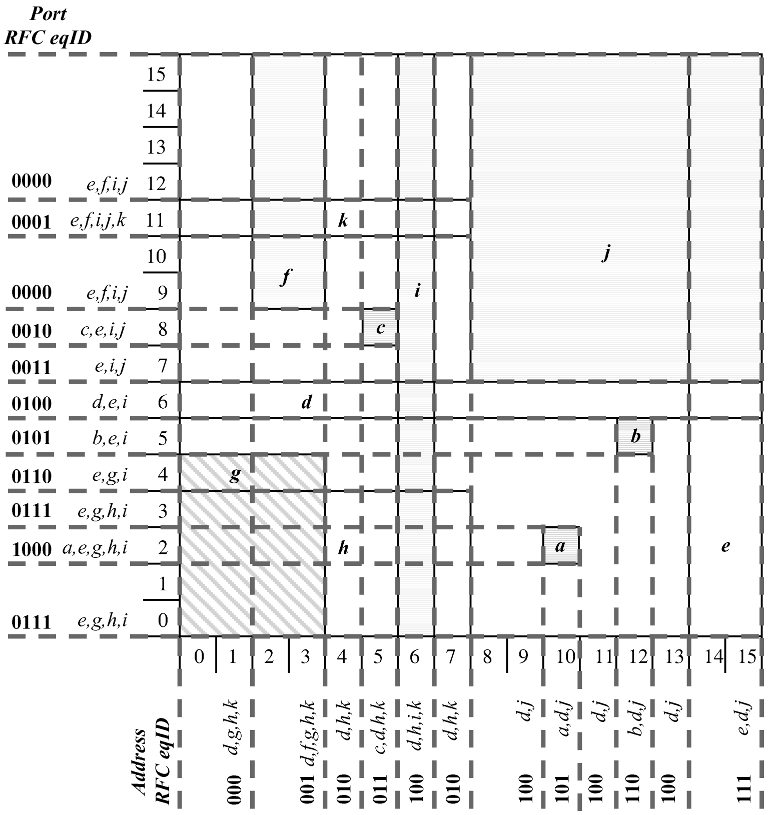


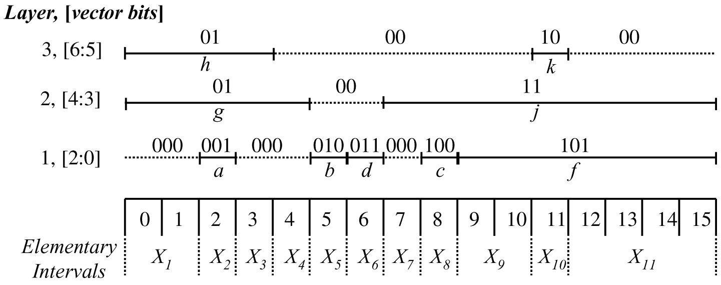
Fig. 18. Example of Recursive Flow Classification (RFC) using the filter set in Table IV.

this example, the rectangles $a, \ldots, k$ are defined by the two fields of the filters in our running example filter set in Table IV. In general, these could be rectangles defined by the projections of two chunks of the filters in the filter set. Note that the fields create nine equivalence classes in the *port* field and eight equivalence classes in the *address* field requiring 4-bit and 3-bit eqIDs, respectively.

RFC lookups in chunk and aggregation tables utilize indexing; the address for the table lookup is formed by concatenating the eqIDs from the previous stages. The resulting eqID is smaller (fewer number of bits) than the address; thus, RFC performs a multistage reduction to a final eqID that specifies the action to apply to the packet. The use of indexing simplifies the lookup process at each stage and allows RFC to provide high throughput. This simplicity and performance comes at the cost of memory inefficiency. Memory usage for less than 1000 filters ranged from a few hundred kilobytes to over one gigabyte of memory depending on the number of stages. The authors discuss a hardware architecture using two 64MB SDRAMs and two 4Mb SRAMs that could perform 30 million lookups per second when operating at 125MHz. The index tables used for aggregation also require significant precomputation in order to assign the proper eqID for the combination of the eqIDs of the previous phases. Such extensive precomputation precludes dynamic updates at high rates.

6.5. Parallel Packet Classification (P^2C)

The *Parallel Packet Classification* (P^2C) scheme introduced by van Lunteren and Engbersen [2003] also falls into the class of techniques using decomposition. The



Elementary Interval	*Intermediate Vector* [6:0]
X_1	0101000
X_2	0101001
X_3	0101000
X_4	0001000
X_5	0000010
X_6	0000011
X_7	0011000
X_8	0011100
X_9	0011101
X_{10}	1011101
X_{11}	0011101

Filter	*Ternary Match Condition*
a	0101001
b	0000010
c	0000100
d	0000011
e	*******
f	0*11101
g	0*0100*
h	010100*
i	*******
j	0*11*0*
k	1011101

Fig. 19. Example of Parallel Packet Classification (P^{2}C) using the most update-efficient encoding style for the port ranges defined in the filter set in Table IV.

key novelties of P^{2}C are its encoding and aggregation of intermediate results. Similar to the Parallel Bit-Vector and RFC techniques, P^{2}C performs parallel searches in order to identify the elementary interval covering each packet field. The authors introduce three techniques for encoding the elementary intervals formed by the projections of filter fields. These techniques explore the design tradeoffs between update speed, space efficiency, and lookup complexity. For each field of each filter, P^{2}C computes the common bits of all the encodings for the elementary intervals covered by the given filter field. This computation produces a ternary search string for each filter field. The ternary strings for each field are concatenated and stored in a TCAM according to the filter priority.

Figure 19 shows an example of the first, and most update efficient, P^{2}C encoding technique for the *port* fields of the filters in Table IV. In this encoding technique, we organize the ranges defined by the filters in the filter set into a multilayer hierarchy such that the ranges at each layer are nonoverlapping and the number of layers is minimized. Note that the number of layers is equal to the maximum number of overlapping ranges for any port number. At each layer, we assign a unique label to the ranges, using the minimum number of bits. Within each layer, regions not covered by a range may share the same label. Next, we compute an intermediate bit vector for each elementary interval $X_1, \ldots, X_{11}$ defined by the filter fields. We form an intermediate bit vector by concatenating the labels for the covering ranges in each layer. Consider elementary interval X_2 which is covered by range $h(01)$ in layer 3, $g(01)$ in layer 2, and $a(001)$ in layer 1; its intermediate bit vector is 0101001. Finally, we compute the ternary match condition for each filter by computing the common bits of the intermediate bit vectors for the set of elementary intervals covered by each filter. For each bit position in the intermediate bit vectors, if all elementary intervals share the same bit value, then we assign that bit value to the corresponding bit position of the ternary match string; otherwise, we assign a don't care, $*$, to the bit position in the ternary match string. Consider filter g which covers elementary intervals X_1, X_2, X_3, and X_4. For all bit vectors, the most significant bit is "0", but they differ in the next bit position. Thus, the ternary match string for g begins with $0*$.

Once we construct the table of ternary match strings for each filter field, we concatenate the field strings associated with each filter and store it in a TCAM. Strings are stored in order of filter priority. We also construct a data structure for each filter field which returns the intermediate bit vector for the elementary interval covering the given packet field. A search locates the best matching filter for a packet by searching these data structures in parallel, concatenating the intermediate bit vectors to form a search key and querying the TCAM with the search key. For the single-field searches, the authors employ the *BARTs* technique which restricts independent field searches to be either prefix or exact match [van Lunteren 2001]. Arbitrary ranges must be converted to prefixes, increasing the number of unique match conditions. The primary deficiency of P^2C is its use of elementary intervals as a single-filter update that may add or remove several elementary intervals for each field. When using the most space efficient encoding techniques, it is possible for one filter update to require updates to every primitive range encoding. Using the most update efficient encoding, the number and size of intermediate results grows super-linearly with the number of filters. For a sample filter set of 1733 filters, P^2C required 2k bytes of SRAM and 5.1k bytes of TCAM. The same filter set requires 24k bytes using a standard TCAM exclusively, thus P^2C reduced TCAM storage requirements by a factor of 4.7 and required only 1.2 bytes of SRAM per filter.

6.6. Distributed Crossproducting of Field Labels (DCFL)

Introduced by Taylor and Turner [2005b], *Distributed Crossproducting of Field Labels* (DCFL) leverages filter set characteristics, decomposition, and a novel labeling technique to construct a packet classification technique targeted to a high-performance hardware implementation. As we discussed in Section 2, Taylor and Turner identified two key properties of real filter sets namely, match condition redundancy and matching set confinement. These properties serve as the antecedent for the decomposition approach and encoding techniques employed by DCFL. Specifically, independent search engines find the matching conditions for each filter field. Field search engines operate in parallel and employ data structures optimized for the type of match condition specified for the particular filter field. DCFL solves the problem of efficiently aggregating the results from independent search engines by transforming it into a distributed set intersection problem. By introducing simple and efficient encoding techniques for intermediate search results, DCFL computes the set of matching filters using a network of aggregation nodes. By performing aggregation in a distributed fashion, DCFL avoids the exponential increase in time or space incurred when performing this operation in a single step as in the original Crossproducting technique discussed in Section 6.3.

DCFL takes advantage of the match condition redundancy property by labeling unique match conditions with locally unique labels. As in Crossproducting, DCFL begins by creating sets of unique match conditions. Each match condition is assigned a locally unique label, and the number of filters specifying the match condition is recorded. The count values are

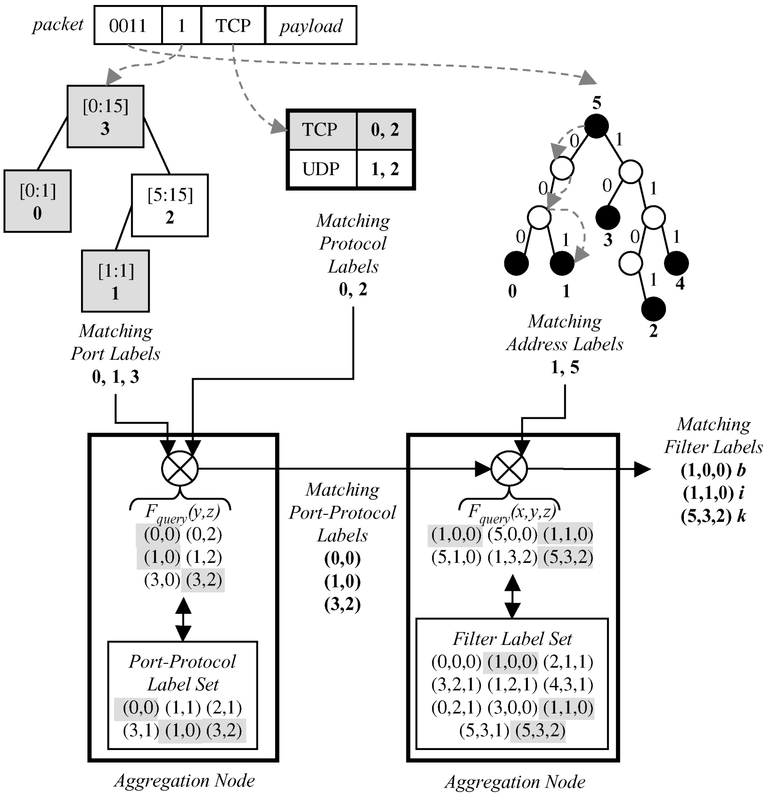


Fig. 20. Example of search using Distributed Crossproducting of Field Labels (DCFL).

incremented and decremented as filters specifying the corresponding conditions are added and removed from the filter set. A data structure in a field search engine or aggregation node only needs to be updated when the count value changes from 0 to 1 or 1 to 0. Given the sets of labels for each field, we can construct a unique label for each filter by simply concatenating the labels for each match condition in the filter. For example, filter *j* in Figure 17 may be uniquely labeled by (5, 3, 1), where the order of field labels is (*Address, Port, Protocol*). The use of labels allows DCFL to aggregate the results from independent field searches using set membership data structures that only store labels corresponding to match conditions and combinations of match conditions present in the filter table. As shown in the *Port-Protocol Label Set* in the first aggregation node in Figure 20, we represent the unique combinations of port and protocol values specified by filters in the filter set by concatenating the labels for the individual match conditions[5].

We provide an example of a DCFL search in Figure 20. We begin by performing parallel searches on the individual packet fields and returning all matching results. In Figure 20, we employ a range tree for the port ranges, a direct lookup table for the protocol fields, and a binary trie for the address prefixes. Note that various options exist for each type of search and DCFL allows each search engine to apply local optimizations. DCFL allows intermediate result aggregation to occur in any order. In our example, we first

[5] Count values are maintained for the sets of unique match condition combinations like the sets of unique match conditions for each field. We do not show the count values in the example in Figure 20.

aggregate the results from the port and protocol field searches. We form the set of all possible matching port-protocol pairs, $F_{query}(y, z)$, by computing the crossproduct of the results from the field searches. Since the field searches returned three port range labels and two protocol field labels, $F_{query}(y, z)$ contains six port-protocol labels. We then compute the intersection of sets $F_{query}(y, z)$ and *Port-Protocol Label Set*. The resulting set of three label pairs is passed on to the next aggregation node. We perform the same steps to form the set of possible matching filter labels, $F_{query}(x, y, z)$, and compute the intersection with the *Filter Label Set*. In this example, three filters match the packet. The labels for the matching filters are passed on to a final priority resolution stage that selects the highest priority filter or set of filters.

In addition to introducing several efficient set intersection data structures, the authors also introduce the concept of *Meta-Labeling* which reduces the memory requirements for aggregation nodes. They also provide techniques for minimizing the number of sequential memory accesses at each aggregation node by computing the optimal ordering of aggregation nodes and limiting the number of labels returned by field search engines. The latter is achieved by a novel technique called *Field Splitting* which we do not discuss in this survey. Using a collection of 12 real filter sets and the *ClassBench* tools suite [Taylor and Turner 2005a], the authors provide analyses of DCFL performance and resource requirements on filter sets of various sizes and compositions. For the 12 real filter sets, they show that the worst-case number of sequential memory accesses is at most ten, and memory requirements are at most 40 bytes per filter. Based on these results, an optimized implementation of DCFL can provide over 100 million searches per second and storage for over 200,000 filters with current generation hardware technology. Using the *ClassBench Filter Set Generator*, the authors show that DCFL may be configured to maintain this performance for filter sets as large as 50,000 filters. They also show that adding an additional aggregation node increases memory requirements by a modest 12.5 bytes per filter. Based on this observation, they assert that DCFL demonstrates scalability to additional filter fields.

7. TUPLE SPACE

We have discussed three high-level approaches to the packet classification problem thus far. The last high-level approach in our taxonomy attempts to quickly narrow the scope of a multiple-field search by partitioning the filter set by tuples. A *tuple* defines the number of specified bits in each field of the filter. Motivated by the observation that the number of distinct tuples is much less than the number of filters in the filter set, Srinivasan et al. [1999] introduced the tuple space approach and a collection of *Tuple Space Search* algorithms in a seminal paper.

In order to illustrate the concept of tuples, we provide an example filter set of filters classifying on five fields in Table V. Address prefixes cover 4-bit addresses and port ranges cover 4-bit port numbers. For address prefix fields, the number of specified bits is simply the number of nonwildcard bits in the prefix. For the protocol field, the value is simply a Boolean: “1” if a protocol is specified, “0” if the wildcard is specified. The number of specified bits in a port range is not as straightforward to define. The authors introduce the concepts of *Nesting Level and Range ID* to define the tuple values for port ranges. Similar to the P^{2}C encoding technique discussed in Section 6.5, all ranges on a given port field are placed into a nonoverlapping hierarchy. The Nesting Level specifies the layer of the hierarchy and the Range ID uniquely labels the range within its layer. In this way, we convert all port ranges to a (Nesting Level, Range ID) pair. The Nesting Level is used as the tuple value for the range, and the Range ID is used to identify the specific range within the tuple. We show an example of assigning Nesting Level and Range ID for the source port ranges of Table V in Figure 21. Given these definitions of tuple values, we list the tuple of each filter in Table V in the last column.

Table V. Example Filter Set
(Address fields are 4-bits and port ranges cover 4-bit portnumbers.)

Filter	SA	DA	SP	DP	Prot	Tuple
a	0*	001*	2 : 2	0 : 15	TCP	[1, 3, 2, 0, 1]
b	01*	0*	0 : 15	0 : 4	UDP	[2, 1, 0, 1, 1]
c	0110	0011	0 : 4	5 : 15	TCP	[4, 4, 1, 1, 1]
d	1100	*	5 : 15	2 : 2	UDP	[4, 0, 1, 2, 1]
e	1*	110*	2 : 2	0 : 15	UDP	[1, 3, 2, 0, 1]
f	10*	1*	0 : 15	0 : 4	TCP	[2, 1, 0, 1, 1]
g	1001	1100	0 : 4	5 : 15	UDP	[4, 4, 1, 1, 1]
h	0011	*	5 : 15	2 : 2	TCP	[4, 0, 1, 2, 1]
i	0*	110*	2 : 2	0 : 15	UDP	[1, 3, 2, 0, 1]
j	10*	0*	2 : 2	2 : 2	TCP	[2, 1, 2, 2, 1]
k	0110	1100	0 : 15	0 : 15	ICMP	[4, 4, 0, 0, 1]
l	1110	*	2 : 2	0 : 15	*	[4, 0, 2, 0, 0]

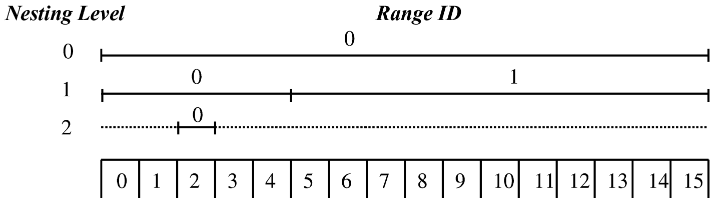


Fig. 21. Example of assigning tuple values for ranges based on Nesting Level and Range ID.

Since the tuple specifies the valid bits of its constituent filters, we can probe tuples for matching filters using a fast exact match technique like hashing. We probe a tuple for a matching filter by using the bits of the packet field specified by the tuple as the search key. For example, we construct a search key for the tuple [1, 3, 2, 0, 1] by concatenating the first bit of the packet source address, the first three bits of the packet destination address, the Range ID of the source port range at Nesting Level 2 covering the packet source port number, the Range ID of the destination port range at Nesting Level 0 covering the packet destination port number, and the protocol field.

All algorithms using the tuple space approach involve a search of the tuple space or a subset of the tuples in the space. Probes to separate tuples may be performed independently, thus tuple space techniques can take advantage of parallelism. The challenge in designing a parallel implementation lies in the unpredictability of the size of the tuple space or subset to be searched. As a result, the realizable lookup performance for tuple space techniques varies widely. Implementations of tuple space techniques can also be made memory efficient due to the effective compression of the filters. The masks or specification of valid bits for filters in the same tuple only needs to be stored once; likewise, only the valid bits of those filters need to be stored in memory. For filter sets with many fields and many wildcards within fields, tuple space techniques can be more space efficient than the $O(N)$ exhaustive techniques discussed in Section 4.

7.1. Tuple Space Search & Tuple Pruning

The basic Tuple Space Search technique introduced by Srinivasan et al. [1999] performs an exhaustive search of the tuple space. For our example filter set in Table V, a search would have to probe seven tuples instead of searching all 12 filters. Using a modest set of real filter sets, the authors found that Tuple Space Search reduced the number of searches by a factor of four to seven relative to an exhaustive search over the set of filters[6].

[6] We make a simplifying assumption that a probe to a tuple is equivalent to examining one filter in an exhaustive search.

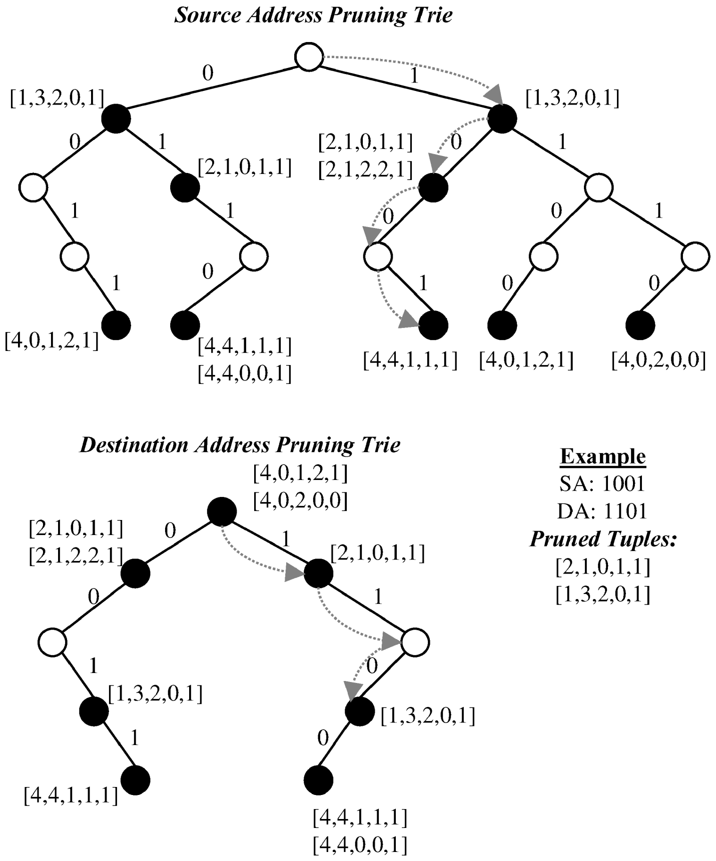


Fig. 22. Example of *Tuple Pruning* to narrow the scope of the *Tuple Space Search*; the set of *pruned tuples* is the intersection of the sets of tuples found along the search paths for each field.

The basic technique can provide adequate performance for large filter sets given favorable filter set properties and a massively parallel implementation.

Motivated by the observation that no address has more than six matching prefixes in backbone route tables, the authors introduced techniques to limit the number of tuples that need to be searched exhaustively. *Pruned Tuple Space Search* reduces the scope of the exhaustive search by performing searches on individual filter fields to find a subset of candidate tuples. While any field or combinations of fields may be used for pruning, the authors found that pruning on the source and destination address strikes a favorable balance between the reduction in candidate tuples and overhead for the pruning steps. We provide an example of pruning on the source and destination addresses in Figure 22. In this case, we begin by constructing tries for the source and destination address prefixes in the filter set in Table V. Nodes representing valid prefixes store a list of tuples containing filters that specify the prefix[7]. We begin a Pruned Tuple Space Search by performing independent searches of the source and destination tries. The result of each search is a list of all possible candidate tuples for each field. In order to construct the list of candidate tuples for the packet, we compute the intersection of the tuple lists returned by each search.

[7] Note that the list of tuples may also be represented as a bitmap as in the Parallel BV technique.

Note that this is very similar to the Parallel Bit-Vector technique discussed in Section 6.1. The key difference is that Pruned Tuple Space Search computes the candidate tuples rather than the overlapping filters. In our example in Figure 22, we demonstrate pruning for a packet with source address 1001 and destination address 1101. In this case, we only have to probe two tuples instead of seven in the basic search. Using a modest set of real filter sets, the authors found that Pruned Tuple Space Search reduced the number of searches by a factor of three to five relative to the basic Tuple Space Search and by a factor of 13 to 26 relative to an exhaustive search over the set of filters.

Srinivasan [2001] expanded this set of algorithms with *Entry Pruned Tuple Search* (*EPTS*). This technique seeks to optimize the Pruned Tuple Search algorithm by eliminating the need to store a search data structure for each dimension by storing pruning information with *matches* in the tuples. The tuple pruning information is stored with each filter in the form of a bitmap of tuples containing nonconflicting filters. These bitmaps may be precomputed for each filter in the filter set. The author presents an algorithm to compute the tuple bitmaps in $O(TN)$, where T is the number of tuples and N is the number of filters.

7.2. Rectangle Search

In their seminal paper, Srinivasan et al. [1999] also present the *Rectangle Search* algorithm that provides theoretically optimal performance for packet classification on two fields without making assumptions about the structure of the filter set. Rectangle Search employs the concepts of *markers* and *precomputation* introduced by the *Binary Search on Prefix Lengths* technique for longest prefix matching [Waldvogel et al. 1997]. As shown in Figure 23, the tuple space for filters with two prefix fields may be viewed as a grid of rectangles where each rectangle is a tuple. For this example, we use the source and destination addresses of the filters in the example filter set shown in Table V[8]. Implementing an exhaustive search over the grid of tuples requires W^2 probes in the worst case.

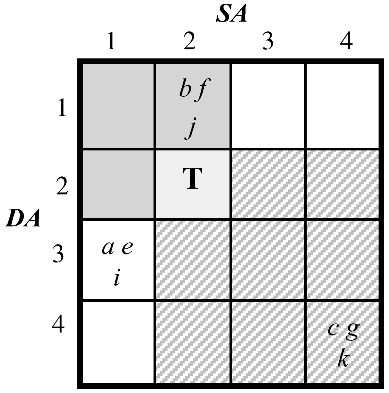


Fig. 23. Example of Rectangle Search on source and destination prefixes of filters in Table V.

The strategy of Rectangle Search is to leverage precomputation and markers to limit the number of probes to at most $(2W - 1)$ where W is the address length. Each filter mapping to a tuple $[i, j]$ leaves a marker in each tuple to its left in its row. For example, a filter $(110*, 0111)$ stored in tuple $[3, 4]$ leaves markers $(11*, 0111)$ in $[2, 4]$ and $(1*, 0111)$ in $[1, 4]$. For all filters and markers in a tuple $[i, j]$, we can precompute the best matching filter from among the filters stored in less specific tuples. Consider tuple $[2, 2]$, labeled T in Figure 23. Graphically, less specific tuples are those in the shaded quadrant above and left of T in Figure 23. For example, if a marker $[01*, 00*]$ were stored in T, then we would precompute the best matching filter b and store it with the marker in T.

Rectangle Search begins at tuple $[1, W]$, the bottom-left tuple. If a matching filter is found, then we need not search any tuples above and left of this tuple due to precomputation. The search moves one tuple to the right to the next tuple in the row. If no matching filter is found in the tuple, then we need not search any tuples below and right of this tuple due to markers. The search moves one tuple up to the

[8] Note that filters containing a wildcard are not included; these filters may be searched by maintaining separate search tries.

next tuple in the column. Note that the worst-case search path follows a staircase pattern from tuple $[1, W]$ to tuple $[W, 1]$, probing a total of $(2W - 1)$ tuples. Rectangle Search requires $O(NW)$ memory as each filter may leave a marker in at most W tuples to its left in its row. The authors proved that $(2W - 1)$ probes is the theoretical lower bound for two fields and extend this bound to show that for d fields the lower bound is:

$$\frac{W^{(d-1)}}{d!} \tag{5}$$

7.3. Conflict-Free Rectangle Search

Warkhede et al. [2001] provide an optimized version of Rectangle Search for the special case of packet classification on a *conflict-free* filter set. A filter set is defined to be conflict-free if there is no pair of overlapping filters in the filter set such that one filter is more specific than the other in one field and less specific in another field. The authors observe that in real filter sets conflicts are rare. Furthermore, techniques exist to resolve filter conflicts by inserting a small set of *resolving filters* that resolve filter conflicts [Hari et al. 2000].

Conflict-Free Rectangle Search begins by mapping the filter set to the $W \times W$ tuple space. Using precomputation and markers, the authors prove that a binary search can be performed on the columns of the grid due to the conflict-free nature of the filter set. This provides an $O(\log^2 w)$ bound on the number of tuple probes and an $O(n \log^2 w)$ bound on memory.

8. CACHING

Finally, we briefly discuss caching, a general technique that can be combined with any search technique to improve average performance. A *cache* is a fast storage buffer for commonly referenced data. If data requests contain sufficient locality, the average time to access data is significantly reduced when the time to access the cache is significantly less than the time to access other storage media [Hennessy and Patterson 1996]. In the context of packet classification, the lookup time is significantly reduced if the time to perform a cache query is significantly less than the time to perform a full lookup. The efficacy of caching schemes largely depends on the data request patterns of the application.

Caching techniques have met with much skepticism from the research community due to the wire-speed requirement discussed in Section 1.1. In short, improving average case performance is irrelevant if we we evaluate packet classification techniques based on worst-case performance. Another argument against caching is the perception that packet flows on high-speed links lack locality of reference. As link speeds have increased, caching schemes have also met with increasing skepticism due to the question of sufficient temporal locality. This question arises due to the fact that the bandwidth requirement of the average packet flow has not increased at the same rate as link capacity. To put it simply, as link bandwidth increases, the number of flows sharing the link also increases. In order for a caching scheme to retain its effectiveness, we must scale the size of the cache with the link speed. Consider the example of a 10Gb/s link supporting individual flows with peak rates of at most 1Mb/s. The packet of a given flow will appear, at most, once in ten thousand packets, thus the cache must have a minimum capacity of ten thousand entries.

Despite the skepticism, a number of cache designs for packet classification have emerged [Chang et al. 2003; Chvets 2002; Li et al. 2003]. One intriguing design utilizes Bloom filters and allows for a small probability of misclassification [Chang et al. 2003]. Holding the misclassification probability to approximately one in a billion, the authors measured an average cache hit-rate of approximately 95 percent using 4KB of memory and real packet traces collected from an OC-3 link; thus, only five percent of the traffic required a full classification lookup. While these results are compelling for low-speed links, the viability of caching for OC-192 (10Gb/s) links remains an open question. It is a difficult one to answer due to the technical challenges of collecting packet traces from such high-speed links. If we simply scale the size of the cache with link speed, this Bloom filter approach would require 256k bytes of cache memory which may be prohibitively large for some applications.

9. DISCUSSION

We have presented a survey of packet classification techniques. Using a taxonomy based on the high-level approach to the problem and a minimal set of running examples, we attempted to provide the reader with a deeper understanding of the seminal and recent algorithms and architectures as well as a useful framework for discerning the relationships and distinctions. While we mentioned the simulation results reported by the authors of the literature introducing each technique, we consciously avoided a direct comparison of the techniques based on throughput, memory requirements, or update performance. Given the various implementation options and variability in simulation parameters, a fair comparison using these metrics is difficult. We believe that future high-performance packet classifiers will invariably be implementations of hybrid techniques that borrow ideas from a number of the previously described techniques. In closing, we would like to briefly highlight the implementation platforms for current and future packet classifiers.

Thanks to the endurance of Moore's Law, integrated circuits continue to provide better performance at lower cost. Application Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) provide millions of logic gates and millions of bits of memory distributed across many multiport embedded memory blocks. For example, a current generation Xilinx FPGA operates at over 400 MHz and contains 556 dual-port embedded memory blocks, 18Kb each with 36-bit wide data paths for a total of over 10Mb of embedded memory [Xilinx 2003]. Current ASIC standard cell libraries offer dual- and quad-port embedded SRAMs operating at 625MHz [IBM Blue Logic 2002]. It is standard practice to utilize several embedded memories in parallel in order to achieve wide data paths. Dual Data Rate (DDR) and Quad Data Rate (QDR) SRAM technologies provide high bandwidth interfaces to several megabytes of off-chip memory [Micron Technology Inc. 2002b; Micron Technology Inc. 2002a]. Network processors also provide a flexible platform for implementing packet classification techniques [Crowley et al. 2002; Kounavis et al. 2003; Shah 2001]. A number of current generation processors provide hardware assists for packet classification, interfaces to TCAM, and/or special instructions for search applications such as hash functions.

ACKNOWLEDGMENTS

I would like to thank Dr. Jonathan S. Turner for his editorial contributions to this work and invaluable mentorship. His consummate emphasis on understanding and clarity inspired this work. I would also like to thank Dr. Ed Spitznagel for contributing his insight to countless discussions on packet classification techniques.

REFERENCES

BABOESCU, F., SINGH, S., AND VARGHESE, G. 2003. Packet classification for core routers: Is there an alternative to CAMs? In *IEEE Infocom*.

BABOESCU, F. AND VARGHESE, G. 2001. Scalable packet classification. In *ACM Sigcomm*.

C. BORMANN, ED. 2001. RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. RFC 3095. *IETF Network Working Group*.

CHANDRANMENON, G. AND VARGHESE, G. 1996. Trading packet headers for packet processing. *IEEE/ACM Trans. Network. 4*, 2 (April) 141–152.

CHANG, F., LI, K., AND CHANG FENG, W. 2003. Approximate packet classification caching. Tech. Rep. CSE-03-002, OGI School of Science and Engineering at OHSU.

CHVETS, I. AND MACGREGOR, N. H. 2002. Multi-zone caches for accelerating IP routing table lookups. In *Proceedings of High-Performance Switching and Routing*.

CROWLEY, P., FRANKLIN, M., HADIMIOGLU, H., AND ONUFRYK, P. 2002. *Network Processor Design: Issues and Practices*. Vol. 1. Morgan Kaufmann Publishers, Inc.

DECASPER, D., PARULKAR, G., DITTIA, Z., AND PLATTNER, B. 1998. Router plugins: A software architecture for next generation routers. In *Proceedings of ACM Sigcomm*.

FELDMANN, A. AND MUTHUKRISHNAN, S. 2000. Tradeoffs for Packet Classification. In *IEEE Infocom*.

GIBSON, G., SHAFAI, F., AND PODAIMA, J. 2000. Content addressable memory storage device. United States Patent 6,044,005. SiberCore Technologies, Inc.

GUPTA, P. AND MCKEOWN, N. 1999a. Packet classification on multiple fields. In *ACM Sigcomm*.

GUPTA, P. AND MCKEOWN, N. 1999b. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*.

HARI, A., SURI, S., AND PARULKAR, G. 2000. Detecting and resolving packet filter conflicts. In *Proceedings of IEEE Infocom*.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture A Quantitative Approach*, 2nd Ed. Morgan Kaufmann Publishers, Inc.

IBM BLUE LOGIC. 2002. *Embedded SRAM Selection Guide*.

KEMPKE, R. A. AND MCAULEY, A. J. 1998. Ternary CAM memory architecture and methodology. United States Patent 5,841,874. Motorola, Inc.

KOUNAVIS, M. E., KUMAR, A., VIN, H., YAVATKAR, R., AND CAMPBELL, A. T. 2003. Directions in packet classification for network processors. In *Second Workshop on Network Processors (NP2)*.

LAKSHMAN, T. V. AND STILIADIS, D. 1998. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *ACM SIGCOMM*.

LI, K., CHANG, F., BERGER, D., AND CHANG FANG, W. 2003. Architectures for packet classification caching. In *Proceedings of IEEE ICON*.

MCAULAY, A. J. AND FRANCIS, P. 1993. Fast routing table lookup using CAMs. In *IEEE Infocom*.

MICRON TECHNOLOGY INC. 2002a. 256Mb Double Data Rate (DDR) SDRAM. Datasheet.

MICRON TECHNOLOGY INC. 2002b. 36Mb DDR SIO SRAM 2-Word Burst. Datasheet.

MICRON TECHNOLOGY INC. 2003. Harmony TCAM 1Mb and 2Mb. Datasheet.

MONTOYE, R. K. 1994. Apparatus for storing "Don't Care" in a content addressable memory cell. United States Patent 5,319,590. HaL Computer Systems, Inc.

NEWMAN, P., MINSHALL, G., AND HUSTON, L. 1997. IP switching and gigabit routers. *IEEE Communications Magazine*.

REKHTER, Y., DAVIE, B., KATZ, D., ROSEN, E., AND SWALLOW, G. 1997. Cisco systems' tag switching architecture overview. RFC 2105.

ROSEN, E., VISWANATHAN, A., AND CALLON, R. 2001. Multiprotocol label switching architecture. RFC 3031.

SHAH, N. 2001. Understanding network processors. Tech. Rep. Version 1.0, EECS, University of California, Berkeley (Sept.).

SIBERCORE TECHNOLOGIES INC. 2002. SiberCAM Ultra-18M SCT1842. Product Brief.

SINGH, S., BABOESCU, F., VARGHESE, G., AND WANG, J. 2003. Packet classification using multidimensional cutting. In *Proceedings of ACM SIGCOMM*. Karlsruhe, Germany.

SPITZNAGEL, E., TAYLOR, D., AND TURNER, J. 2003. Packet classification using extended TCAMs. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*.

SRINIVASAN, V. 2001. A packet classification and filter management system. Microsoft Research.

SRINIVASAN, V., SURI, S., AND VARGHESE, G. 1999. Packet classification using tuple space search. In *ACM SIGCOMM*. 135–146.

SRINIVASAN, V., SURI, S., VARGHESE, G., AND WALDVOGEL, M. 1998. Fast and scalable layer four switching. In *ACM Sigcomm*.

TAYLOR, D. E. AND TURNER, J. S. 2005a. ClassBench: A packet classification benchmark. In *Proceedings of IEEE Infocom*.

TAYLOR, D. E. AND TURNER, J. S. 2005b. Scalable packet classification using distributed crossproducting of field labels. In *Proceedings of IEEE Infocom*.

VAN LUNTEREN, J. 2001. Searching very large routing tables in wide embedded memory. In *Proceedings of IEEE Globecom*. 3, 1615–1619.

VAN LUNTEREN, J. AND ENGBERSEN, T. 2003. Fast and scalable packet classification. *IEEE J. Select. Areas Comm. 21*, 4 (May) 560–571.

WALDVOGEL, M., VARGHESE, G., TURNER, J., AND PLATTNER, B. 1997. Scalable high speed IP routing table lookups. In *Proceedings of ACM SIGCOMM*. 25–36.

WARKHEDE, P., SURI, S., AND VARGHESE, G. 2001. Fast packet classification for two-dimensional conflict-free filters. In *IEEE Infocom*.

WOO, T. Y. C. 2000. A modular approach to packet classification: Algorithms and results. In *IEEE Infocom*.

XILINX. 2003. Virtex-II Pro Platform FPGAs: Introduction and Overview. DS083-1 (v3.0).

Received June 2004; revised November 2005; accepted November 2005