Scalable Packet Classification using Distributed Crossproducting of Field Labels

David E. Taylor, Jonathan S. Turner

WUCSE-2004-38

June 23, 2004

Applied Research Laboratory
Department of Computer Science and Engineering
Washington University in Saint Louis
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130
davidtaylor@wustl.edu

Abstract

A wide variety of packet classification algorithms and devices exist in the research literature and commercial market. The existing solutions exploit various design tradeoffs to provide high search rates, power and space efficiency, fast incremental updates, and the ability to scale to large numbers of filters. There remains a need for techniques that achieve a favorable balance among these tradeoffs and scale to support classification on additional fields beyond the standard 5-tuple. We introduce *Distributed Crossproducting of Field Labels* (*DCFL*), a novel combination of new and existing packet classification techniques that leverages key observations of the structure of real filter sets and takes advantage of the capabilities of modern hardware technology. Using a collection of real and synthetic filter sets, we provide analyses of *DCFL* performance and resource requirements on filter sets of various sizes and compositions. An optimized implementation of *DCFL* can provide over 100 million searches per second and storage for over 200 thousand filters with current generation hardware technology.

Table 1: Example filter set.

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†
10011100	01101010	UDP	[5:5]	3	2
*	*	ICMP	[0:15]	4	9†
100111*	011010*	*	[3:15]	5	6†
10010011	*	TCP	[3:15]	6	3
*	*	UDP	[3:15]	7	9†
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†
10011100	01101010	TCP	[0:1]	13	3
01110010	*	*	[3:3]	14	3
100111*	011010*	UDP	[1:1]	15	4

1 Introduction

Packet classification is an enabling function for a variety of applications including Quality of Service, security, and monitoring. Such applications typically operate on packet flows; therefore, network nodes must classify individual packets traversing the node in order to assign a flow identifier, *FlowID*. Packet classification entails searching a set of filters for the highest priority filter or set of filters which match the packet [1]. At minimum, filters contain multiple field values that specify an exact packet header or set of headers and the associated *FlowID* for packets matching all the field values. The type of field values 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 1. In this simple example, filters contain field values for four packet headers fields: 8-bit source and destination addresses, transport protocol, and a 4-bit destination port number.

Note that the filters in Table 1 also contain an explicit priority tag *PT* and a non-exclusive flag denoted by †. These additional values allow for ease of maintenance and provide a supportive platform for a wider variety of applications. Priority tags allow filter priority to be independent of filter ordering, providing for simple and efficient dynamic updates. Non-exclusive flags allow filters to be designated as either exclusive or non-exclusive. 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 non-exclusive filters, providing support for transparent monitoring and usage-based accounting applications. Note that a parameter may control the number of non-exclusive filters, *r*, returned by the packet classifier. Like exclusive filters, the priority tag is used to select the *r* highest priority non-exclusive filters. We argue that packet classifiers should support these additional filter values and point out that many existing algorithms preclude their use.

Distributed Crossproducting of Field Labels (*DCFL*) is a novel combination of new and existing packet

[1] Note that filters are also referred to as rules in some of the packet classification literature.

classification techniques that leverages key observations of filter set structure and takes advantage of the capabilities of modern hardware technology. We discuss the observed structure of real filter sets in detail and provide motivation for packet classification on larger numbers of fields in Section 2. Two key observations motivate our approach: the number of unique field values for a given field in the filter set is small relative to the number of filters in the filter set, and the number of unique field values matched by any packet is very small relative to the number of filters in the filter set. We also draw from the encoding ideas highlighted in [1] in order to efficiently store the filter set and intermediate search results. Using a high degree of parallelism, *DCFL* employs optimized search engines for each filter field and an efficient technique for aggregating the results of each field search. By performing this aggregation in a distributed fashion, we avoid the exponential increase in the time or space incurred when performing this operation in a single step. Given that search techniques for single packet fields are well-studied, the primary focus of this paper is the development and analysis of an aggregation technique that can make use of the embedded multi-port memory blocks in the current generation of ASICs and FPGAs. We introduce several new concepts including field labeling, *Meta-Labeling* unique field combinations, *Field Splitting*, and optimized data structures such as *Bloom Filter Arrays* that minimize the number of memory accesses to perform set membership queries. As a result, our technique provides fast lookup performance, efficient use of memory, support for dynamic updates at high rates, and scalability to filters with additional fields.

Using a collection of 12 real filter sets and and synthetic filter sets generated with the *ClassBench* tools, we provide an evaluation of *DCFL* performance and resource requirements for filter sets of various sizes and compositions in Section 9. We show that an optimized implementation of *DCFL* can provide over 100 million searches per second and storage for over 200 thousand filters in a current generation FPGA or ASIC without the need for external memory devices. 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. We provide a brief overview of related work in Section 10, focusing on algorithms most closely related to our approach.

2 Key Observations

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 multi-field searching problem have been established, observations made in recent packet classification work offer enticing new possibilities to provide significantly better performance. We begin by reviewing the results of previous efforts to extract statistical characteristics of filter sets, followed by our own observations which led us to develop the *DCFL* technique.

2.1 Previous Observations

Gupta and McKeown published a number of observations regarding the characteristics of real filter sets which have been widely cited [2]. Others have performed analyses on real filter sets and published their observations [3, 4, 5, 6]. The following is a distillation of previous observations relevant to our work:

- 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 IPINIP.

- 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 field values.

The final observation is pivotal. 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 the general framework for packet classification in network processors proposed by Kounavis, et. al. [7].

2.2 Our Observations

We performed a battery of analyses on 12 real filter sets provided by Internet Service Providers (ISPs), a network equipment vendor, and other researchers working in the field. The filter sets range in size from 68 to 4557 entries. In general, our analyses agree with previously published observations. We also performed an exhaustive analysis of the maximum number of unique field values and unique combinations of field values which match any packet. A summary of the single field statistics are given in Table 2. Note that the number of unique field values is significantly less than the number of filters and the maximum number of unique field values matching any packet remains relatively constant for various filter set sizes. We also performed the same analysis for every possible combination of fields (every possible combination of two fields, three fields, etc.). There are

$$\sum_{i=1}^{d} \left(\begin{array}{c} d \\ i \end{array} \right) \tag{1}$$

unique combinations of d fields. We observed that the maximum number of unique combinations of field values which match any packet is typically bounded by twice the maximum number of matching single field values, and also remains relatively constant for various filter set sizes.

Finally, an examination of real filter sets reveals that additional fields beyond the standard 5-tuple are relevant. In nearly all filter sets that we studied, filters contain matches on TCP flags or ICMP type numbers. In most filter sets, a small percentage of the filters specify a non-wildcard value for the flags, typically less then two percent. There are notable exceptions, as approximately half the filters in database *ipc1* contain non-wildcard flags. We argue that 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. 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)

Table 2: Maximum number of unique field values matching any packet; data from 12 real filter sets; number of unique field values in each filter set is given in parentheses.

Filter Set		*Fields*					
Name	*Size*	*Src Addr*	*Dest Addr*	*Src Port*	*Dest Port*	*Prot*	*Flag*
fw2	68	3 (31)	3 (21)	2 (9)	1 (1)	2 (5)	
fw5	160	5 (38)	4 (35)	3 (11)	3 (33)	2 (4)	2 (11)
fw3	184	4 (31)	3 (28)	3 (9)	3 (39)	2 (4)	2 (11)
ipc2	192	3 (29)	2 (32)	2 (3)	2 (3)	2 (4)	2 (8)
fw4	264	3 (30)	4 (43)	4 (28)	3 (49)	2 (9)	
fw1	283	4 (57)	4 (66)	3 (13)	3 (43)	2 (5)	2 (11)
acl2	623	5 (182)	5 (207)	1 (1)	4 (27)	2 (5)	2 (6)
acl1	733	4 (97)	4 (205)	1 (1)	5 (108)	2 (4)	2 (3)
ipc1	1702	4 (152)	5 (128)	4 (34)	5 (54)	2 (7)	2 (11)
acl3	2400	6 (431)	4 (516)	2 (3)	6 (190)	2 (5)	2 (3)
acl4	3061	7 (574)	5 (557)	2 (3)	7 (235)	2 (7)	2 (3)
acl5	4557	3 (169)	2 (80)	1 (1)	4 (40)	1 (4)	2 (2)

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 [8]. 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 field values 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. We believe 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.

3 Description of DCFL

Distributed Crossproducting of Field Labels (*DCFL*) may be described at a high-level using the following notation:

- Partition the filters in the filter set into fields
- Partition each packet header into corresponding fields
- Let F_i be the set of unique field values for filter field i that appear in one or more filters in the filter set

- Let $F_i(x) \subseteq F_i$ be the subset of filter field values in F_i matched by a packet with the value x in header field i
- Let $F_{i,j}$ be the set of unique filter field value pairs for fields i and j in the filter set; i.e. if $(u, v) \in F_{i,j}$ there is some filter or filters in the set with u in field i and v in field j
- Let $F_{i,j}(x, y) \subseteq F_{i,j}$ be the subset of filter field value pairs in $F_{i,j}$ matched by a packet with the value x in header field i and y in header field j
- This can be extended to higher-order combinations, such as set $F_{i,j,k}$ and subset $F_{i,j,k}(x, y, z)$, etc.

The *DCFL* method can be structured in many different ways. In order to illustrate the lookup process, assume that we are performing packet classification on four fields and a header arrives with field values $\{w, x, y, z\}$. One possible configuration of a *DCFL* search is shown in Figure 1 and proceeds as follows:

- In parallel, find subsets $F_1(w)$, $F_2(x)$, $F_3(y)$, and $F_4(z)$
- In parallel, find subsets $F_{1,2}(w, x)$ and $F_{3,4}(y, z)$ as follows:
  - Let $F_{query}(w, x)$ be the set of possible field value pairs formed from the crossproduct of $F_1(w)$ and $F_2(x)$
  - For each field value pair in $F_{query}(w, x)$, query for set membership in $F_{1,2}$, if the field value pair is in set $F_{1,2}$ add it to set $F_{1,2}(w, x)$
  - Perform the symmetric operations to find subset $F_{3,4}(y, z)$
- Find subset $F_{1,2,3,4}(w, x, y, z)$ by querying set $F_{1,2,3,4}$ with the field value combinations formed from the crossproduct of $F_{1,2}(w, x)$ and $F_{3,4}(y, z)$
- Select the highest priority exclusive filter and r highest priority non-exclusive filters in $F_{1,2,3,4}(w, x, y, z)$

Note that there are several variants which are not covered by this example. For instance, we could alter the aggregation process to find the subset $F_{1,2,3}(w, x, y)$ by querying $F_{1,2,3}$ using the crossproduct of $F_{1,2}(w, x)$ and $F_3(y)$. We can then find the subset $F_{1,2,3,4}(w, x, y, z)$ by querying $F_{1,2,3,4}$ using the crossproduct of $F_{1,2,3}(w, x, y)$ and $F_4(z)$. A primary focus of this paper is determining subsets ($F_{1,2}(w, x)$, $F_{3,4}(y, z)$, etc.) via optimized set membership data structures.

As shown in Figure 1, *DCFL* employs three major components: a set of parallel search engines, an aggregation network, and a priority resolution stage. Each search engine F_i independently searches for all filter fields matching the given header field using an algorithm or architecture optimized for the type of search. For example, the search engines for the IP address fields may employ compressed multi-bit tries while the search engine for the protocol and flag fields use simple hash tables. We provide a brief overview of options for performing the independent searches on packet fields in Section 7. As previously discussed in Section 2 and shown in Table 2, each set of matching labels for each header field is typically less than five for real filter tables. The sets of matching labels generated by each search engine are fed to the aggregation network which computes the set of all matching filters for the given packet in a multi-stage, distributed fashion. Finally, the priority resolution stage selects the highest priority exclusive filter and the r highest priority non-exclusive filters. The priority resolution stage may be realized by a number of efficient algorithms and logic circuits; hence, we do not discuss it further.

The first key concept in *DCFL* is labeling unique field values with locally unique labels; thus, sets of matching field values can be represented as sets of labels. Table 3 shows the sets of unique source

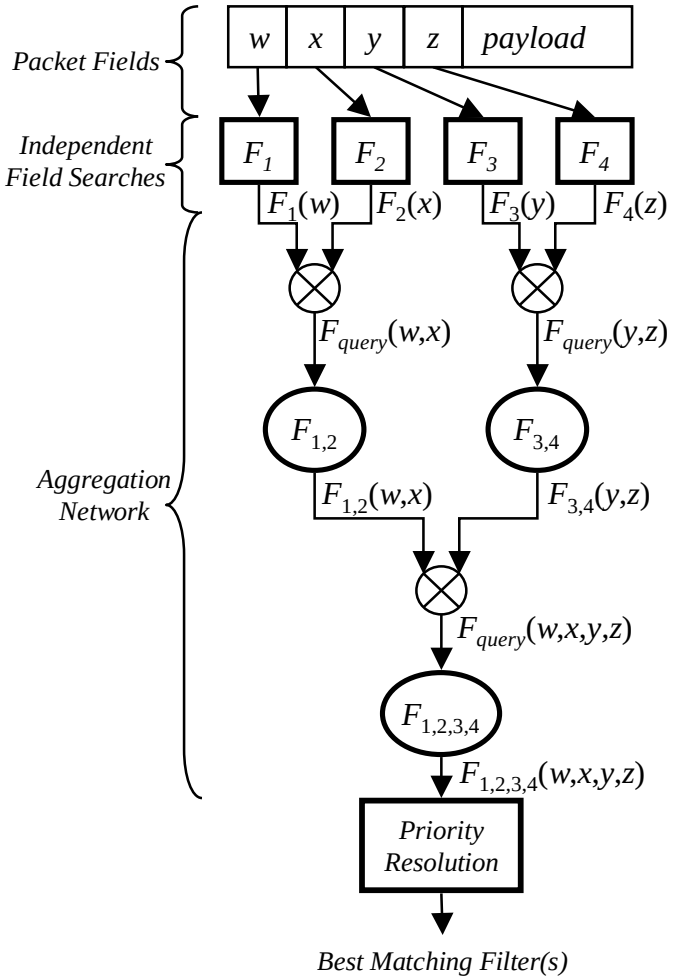

Figure 1: Example configuration of *Distributed Crossproducting of Field Labels* (*DCFL*); field search engines operate in parallel and may be locally optimized; aggregation nodes also operate in parallel; aggregation network may be constructed in a variety of ways.

and destination addresses specified by the filters in Table 1. Note that each unique field value also has an associated "count" value which records the number of filters which specify the field value. The "count" value is used to support dynamic updates; 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. We identify unique combinations of field values by assigning either (1) a composite label formed by concatenating the labels for each field value in the combination, or (2) a new *meta-label* which uniquely identifies the combination in the set of unique combinations[2]. *Meta-Labeling* essentially compresses the size of the label used to uniquely identify the field combination. In addition to reducing the memory requirements for explicitly storing composite labels, this optimization has another subtle benefit. *Meta-Labeling* compresses the space addressed by the label, thus the *meta-label* may be used as an index into a set membership data structure. The use of labels allows us to use set membership data structures that only store labels corresponding to field values and combinations of field values present in the filter table. While storage requirements depend on the structure of the filter set, they scale linearly with the number of filters in the database. Furthermore, at each aggregation node we need not perform set membership queries in any particular order. This property allows us to take advantage

[2]Meta-labeling can be thought of as simply numbering the set of unique field combinations

Table 3: Sets of unique specifications for each field in the sample filter set.

SA	*Label*	*Count*
11010010	0	1
10011100	1	1
101101*	2	1
10011100	3	2
*	4	2
100111*	5	2
10010011	6	1
11101100	7	1
111010*	8	1
100110*	9	1
010110*	10	1
01110010	11	2

DA	*Label*	*Count*
*	0	7
001110*	1	1
01101010	2	2
011010*	3	2
01111010	4	1
01011000	5	1
11011000	6	2

PR	*Label*	*Count*
TCP	0	4
*	1	5
UDP	2	6
ICMP	3	1

DP	*Label*	*Count*
[3:15]	0	5
[1:1]	1	2
[0:15]	2	5
[5:5]	3	1
[6:6]	4	1
[0:1]	5	1
[3:3]	6	1

of hardware parallelism and multi-port embedded memory technology.

The second key concept in *DCFL* is employing a network of aggregation nodes to compute the set of matching filters for a given packet. The aggregation network consists of a set of interconnected aggregation nodes which perform set membership queries to the sets of unique field value combinations, $F_{1,2}$, $F_{3,4,5}$, etc. By performing the aggregation in a multi-stage, distributed fashion, the number of intermediate results operated on by each aggregation node remains small. Consider the case of finding all matching address prefix pairs in the example filter set in Table 1 for a packet with address pair $(x, y) = (10011100, 01101010)$. As shown in Figure 2, an aggregation node takes as input the sets of matching field labels generated by the source and destination address search engines, $F_{SA}(x)$ and $F_{DA}(y)$, respectively. Searching the tables of unique field values shown in Table 3, $F_{SA}(x)$ contains labels {1,4,5} and $F_{DA}(y)$ contains labels {0,2,3}. The first step is to form a query set F_{query} of aggregate labels corresponding to potential address prefix pairs. The query set is formed from the crossproduct of the source and destination address label sets. Next, each label in F_{query} is checked for membership in the set of labels stored at the aggregation node, $F_{SA,DA}$. Note that the set of composite labels corresponds to unique address prefix pairs specified by filters in the example filter set shown in Table 1. Composite labels contained in the set are added to the matching label set $F_{SA,DA}(x, y)$ and passed to the next aggregation node. Since the number of unique field values and field value combinations is limited in real filter sets, the size of the crossproduct at each aggregation node remains manageable. By performing crossproducting in a distributed fashion across a network of aggregation nodes, we avoid an exponential increase in search time that occurs when aggregating the results from all field search

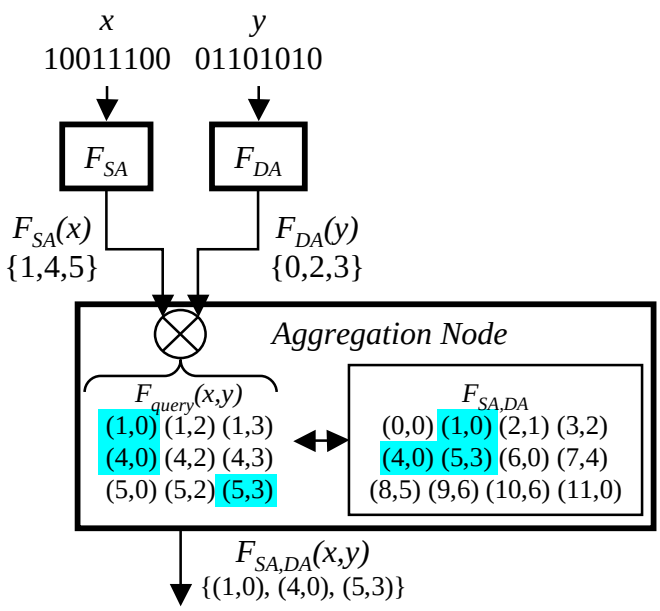

Figure 2: Example aggregation node for source and destination address fields.

engines in a single step. Note that the aggregation nodes only store unique combinations of fields present in the filter table; therefore, we also avoid the exponential blowup in memory requirements suffered by the original *Crossproducting* technique [9] and *Recursive Flow Classification* [2]. In Section 5, we introduce *Field Splitting* which limits the size of F_{query} at aggregation nodes, even when the number matching labels generated by field search engines increases.

DCFL is amenable to various implementation platforms, and where possible, we will highlight the various configurations of the technique that are most suitable for the most popular platforms. In order to illustrate the value of our approach, we focus on the highest performance option for the remainder of this paper. It is important to briefly describe this intended implementation platform here, as it will guide the selection of data structures for aggregation nodes and optimizations in the following sections. 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 multi-port 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 [10]. Current ASIC standard cell libraries offer dual- and quad-port embedded SRAMs operating at 625 MHz [11]. We also point out that it is standard practice to utilize several embedded memories in parallel in order to achieve the desired data path width. It is our goal to make full use of the parallelism and high-performance embedded memory provided by the current generation of ASICs and FPGAs. This requires that we maximize parallel computations and storage efficiency. If necessary, high-performance off-chip memory technologies such as Dual Data Rate (DDR) and Quad Data Rate (QDR) SRAM technologies could be employed; however, the number of available off-chip memories is limited by the number of I/O pins on the implementation device.

4 Aggregation Network

Since all aggregation nodes operate in parallel, the performance bottleneck in the system is the aggregation node with the largest worst-case query set size, $|F_{query}|$. Query set size determines the number of sequential memory accesses performed at the node. The size of query sets vary for different constructions of the aggregation network. We refer to the worst-case query set size, $|F_{query}|$, among all aggregation nodes, $F_1, \ldots, F_{1,\ldots,d}$, as the cost for network construction, G_i. Selecting the most efficient arrangement of aggregation nodes into an aggregation network is a key issue. We want to select the minimum cost aggregation network G_{min} as follows:

$$G_{min} = G : cost(G) = min\left\{cost\left(G_i\right) \forall i\right\} \tag{2}$$

where

$$cost\left(G\right) = max\left\{|F_{query}| \forall F_1, \ldots, F_{1,\ldots,d} \in G_i\right\} \tag{3}$$

Consider an example for packet classification on three fields. Shown in Figure 3 are the maximum sizes for the sets of matching field labels for the three fields and the maximum size for the sets of matching labels for all possible field combinations. For example, label set $F_{1,2}(x, y)$ will contain at most four labels for any values of x and y. Also shown in Figure 3 are three possible aggregation networks for a *DCFL* search; the cost varies between 3 and 6 depending on the construction.

In general, an aggregation node may operate on two or more input label sets. Given that we seek to minimize $|F_{query}|$, we limit the number of input label sets to two. The query set size for aggregation nodes fed by field search engines is partly determined by the size of the matching field label sets, which we have found to be small for real filter sets. Also, the *Field Splitting* optimization provides a control point for the size of the query set at the aggregation nodes fed by the field search engines; thus, we restrict the network structure by requiring that at least one of the inputs to each aggregation node be a matching field label set from a field search engine. Figure 4 shows a generic aggregation network for packet classification on d fields. Aggregation node $F_{1,\ldots,i}$ operates on matching field label set $F_i(x)$ and matching composite label set $F_{1,\ldots,i-1}(a, \ldots, w)$ generated by upstream aggregation node $F_{1,\ldots,i-1}$. Note that the first aggregation node operates on label sets from two field search engines, $F_1(a)$ and $F_2(b)$. We point out that this seemingly "serial" arrangement of aggregation nodes does not prevent *DCFL* from starting a new search on every pipeline cycle. As shown in Figure 4, delay buffers allow field search engines to perform a new lookup on every pipeline cycle. The matching field label sets are delayed by the appropriate number of pipeline cycles such that they arrive at the aggregation node synchronous to the matching label set from the upstream aggregation node. Search engine results experience a maximum delay of $(d - 2)$ pipeline cycles which is tolerable given that the pipeline cycle time is on the order of 10ns. With such an implementation, *DCFL* throughput is inversely proportional to the pipeline cycle time.

In this case, the problem is to choose the ordering of aggregation nodes which results in the minimum network cost. For example, do we first aggregate the source and destination field labels, then aggregate the address pair labels with the protocol field labels? We can empirically determine the optimal arrangement of aggregation nodes for a given filter set by computing the maximum query set size for each combination of field values in the filter set. While this computation is manageable for real filter sets of moderate size, the computational complexity increases exponentially with filter set size. For our set of 12 real filter sets, the optimal network aggregated field labels in the order of decreasing maximum matching filter label set size with few exceptions. This observation can be used as a heuristic for constructing efficient aggregation networks for large filter sets and filter sets with large numbers of filter fields. As previously discussed, we do not expect the filter set properties leveraged by *DCFL* to change. We do point out that a static arrangement of aggregation nodes might be subject to degraded performance if the filter set characteristics were dramatically altered by a sequence of updates. Through the use of reconfigurable interconnect in the aggregation network

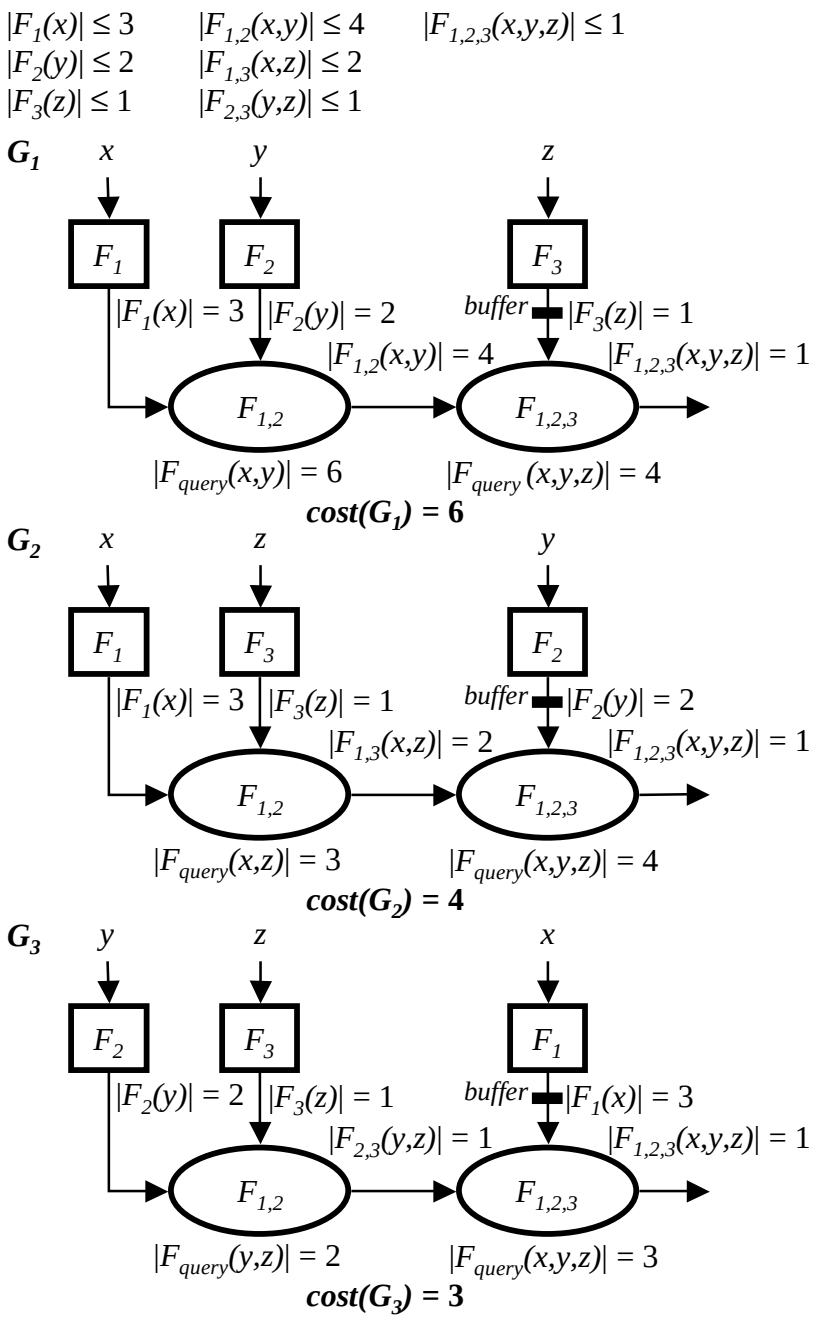

Figure 3: Example of variable aggregation network cost for different aggregation network constructions for packet classification on three fields.

and extra memory for storing off-line aggregation tables, a *DCFL* implementation can minimize the time for restructuring the network for optimal performance. We defer this discussion to future study.

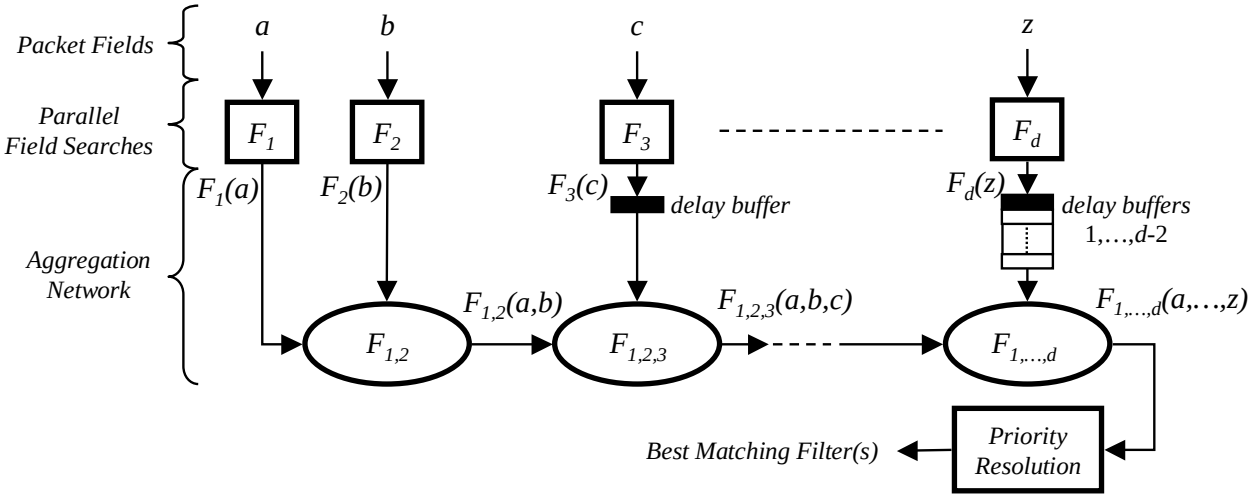

Figure 4: Generalized DCFL aggregation network for a search on d fields.

5 Field Splitting

As discussed in Section 3, the size of the matching field label set, $|F_i(x)|$, affects the size of the crossproduct, $|F_{query}|$, at the following aggregation node. While we observe that $|F_i(x)|$ remains small for real filter sets, we would like to exert control over this value to both increase search speed for existing filter sets and maintain search speed for filter sets with increased address prefix nesting and port range overlaps. Recall that $|F_i(x)| \leq 2$ for all exact match fields such as the transport protocol and protocol flags.

The number of address prefixes matching a given address can be reduced by *splitting* the address prefixes into a set of $(c+1)$ shorter address prefixes, where c is the number of splits. An example of splitting a 6-bit address field is shown in Figure 5. For the original 6-bit address field, *A(5:0)*, the maximum number of field labels matching any address is five. In order to reduce this number, we split the 6-bit address field into a 2-bit address field, *A(5:4)*, and a 4-bit address field, *A(3:0)*. Each original 6-bit prefix creates one entry in each of the new prefix fields as shown. If an original prefix is less than three bits in length, then the entry in field *A(3:0)* is the wildcard. We assign a label to each of the unique prefixes in the new fields and create data structures to search the new fields in parallel in separate search engines. In this example we use binary trees; regardless of the data structure, the search engine must return all matching prefixes. The prefixes originally in *A(5:0)* are now identified by the unique combination of labels corresponding to their entries in *A(5:4)* and *A(3:0)*. For example, the prefix $000*$ in *A(5:0)* is now identified by the label combination $(3, 1)$. A search proceeds by searching *A(5:4)* and *A(3:0)* with the first two bits and remaining 4 bits of the packet address, respectively. Note that the maximum number of field labels returned by the new search engines is three. We point out that the sets of matching labels from *A(5:4)* and *A(3:0)* may be aggregated in any order, with label sets from any other filter field; we need not aggregate the labels from *A(5:4)* and *A(3:0)* in the same aggregation node to ensure correctness. For address prefixes, *Field Splitting* is similar to constructing a variable-stride multi-bit trie; however, with *Field Splitting* we only store one multi-bit node per stride. A matching prefix is denoted by the combination of matching prefixes from the multi-bit nodes in each stride.

Given that the size of the matching field label sets is the property that most directly affects *DCFL* performance, we would like to specify a maximum set size and split those fields that exceed the threshold. Given a field overlap threshold, there is a simple algorithm for determining the number of splits required for an address prefix field. For a given address prefix field, we begin by forming a list of all unique address

A(5:0)	Label	A(5:4)	Label	A(3:0)	Label
*	0	*	0	*	0
0*	1	0*	1	*	0
01*	2	01	2	*	0
000*	3	00	3	0*	1
0110*	4	01	2	10*	2
1010*	5	10	4	10*	2
10100*	6	10	4	100*	3
011010	7	01	2	1010	4

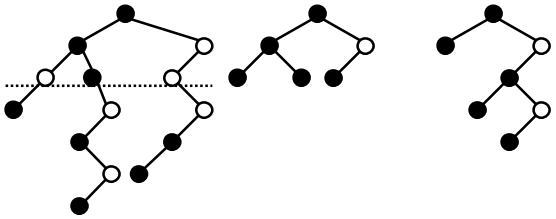

Figure 5: An example of splitting a 6-bit address field; maximum number of matching labels per field is reduced from five to three.

prefixes in the filter set, sorted in non-decreasing order of prefix length. We simply add each prefix in the list to a binary trie, keeping track of the number of prefixes encountered along the path using a nesting counter. If there is a split at the current prefix length, we reset the nesting counter. The splits for the trie may be stored in a list or an array indexed by the prefix length. If the number of prefixes along the path reaches the threshold, we create a split at that prefix length and reset the nesting counter. It is important to note that the number of splits depends upon the structure of the address trie. In the worst case, a threshold of two overlaps could create a split at every prefix length. We argue that given the structure of real filter sets and reasonable threshold values (four or five), that *Field Splitting* provides a highly useful control point for the size of query sets in aggregation nodes.

Field Splitting for port ranges is much simpler. We simply compute the maximum field overlap, m, for the given port field by adding the set of unique port ranges to a segment tree. Given an overlap threshold, t, the number splits is simply $c = \frac{m-2}{t-1}$. We then create $(c+1)$ bins in which to sort the set of unique port ranges. For each port range $[i:j]$, we identify the bin, b_i, containing the minimum number of overlapping ranges using a segment tree constructed from the ranges in the bin. We insert $[i:j]$ into bin b_i and insert wildcards into the remaining bins. Once the sorting is complete, we assign locally unique labels to the port ranges in each bin. Like address field splitting, a range in the original filter field is now identified by a combination of labels corresponding to its matching entry in each bin. Again, label aggregation may occur in any order with labels from any other field.

Finally, we point out that *Field Splitting* is a precomputed optimization. It is possible that the addition of new filters to the filter set could cause one the overlap threshold to be exceeded in a particular field, and thus degrade the performance of *DCFL*. While this is possible, our analysis of real filter sets suggests that it is not probable. Currently most filter sets are manually configured, thus updates are exceedingly rare relative to searches. Furthermore, the common structure of filters in a filter set suggests that new filters will most likely be a new combination of fields already in the filter set. For example, a network administrator may add a filter matching all packets for application A flowing between subnets B and C, where specifications A, B, C already exist in the filter set.

6 Aggregation Nodes

Well-studied data structures such as hash tables and B-Trees are capable of efficiently representing a set [12]. We focus on three options that minimize the number of sequential memory accesses, *SMA*, required to identify the composite labels in F_{query} which are members of the set $F_{1,\ldots,i}$. The first is a variant on the popular Bloom filter which has received renewed attention in the research literature [13]. The second and third options leverage the compression provided by field labels and meta-labels to index into an array of lists containing the composite labels for the field value combinations in $F_{1,\ldots,i}$. These indexing schemes perform parallel comparisons in order to minimize the required *SMA*; thus, the performance of these schemes depends on the word size m of the memory storing the data-structures. For all three options, we derive equations for *SMA* and number of memory words W required to store the data-structure.

6.1 Bloom Filter Arrays

A Bloom filter is an efficient data structure for set membership queries with tunable false positive errors. In our context, a Bloom filter computes k hash functions on a label L to produce k bit positions in a bit vector of m bits. If all k bit positions are set to 1, then the label is declared to be a member of the set. Broder and Mitzenmacher provide a nice introduction to Bloom filters and their use in recent work [13]. We provide a brief introduction to Bloom filters and a derivation of the equations governing false positive probability in Appendix A. False positive answers to membership queries causes the matching label set, $F_{1,\ldots,i}(a,\ldots,x)$, to contain labels that do not correspond to field combinations in the filter set. These false positive errors can be "caught" at downstream aggregation nodes using explicit representations of label sets. We discuss two options for such data-structures in the next section. This property does preclude use of Bloom filters in the last aggregation node in the network. As we discuss in Section 9, this does not incur a performance penalty in real filter sets.

Given that we want to minimize the number of sequential memory accesses at each aggregation node, we want to avoid performing multiply memory accesses per set membership query. It would be highly inefficient to perform k separate memory accesses to check if a single bit is set in the vector. In order to limit the number of memory accesses per membership query to one, we propose the use of an array of Bloom filters as shown in Figure 6. A *Bloom Filter Array* is a set of Bloom filters indexed by the result of a pre-filter hash function $H(L)$. In order to perform a set membership query for a label L, we read the Bloom filter addressed by $H(L)$ from memory and store it in a register. We then check the bit positions specified by the results of hash functions $h_1(L),\ldots,h_k(L)$. The *Match Logic* checks if all bit positions are set to 1. If so, it adds label L to the set of matching labels $F_{1,\ldots,i}(a,\ldots,x)$.

Set membership queries for the labels in F_{query} need not be performed in any order and may be performed in parallel. Using an embedded memory block with P ports requires P copies of the logic for the hash functions and *Match Logic*. Given the ease of implementing these functions in hardware and the fact that P is rarely more than four, the additional hardware cost is tolerable. The number of sequential memory accesses, *SMA*, required to perform set membership queries for all labels in F_{query} is simply

$$SMA = \frac{|F_{query}|}{P} \tag{4}$$

The false positive probability is

$$f = \left(\frac{1}{2}\right)^k \tag{5}$$

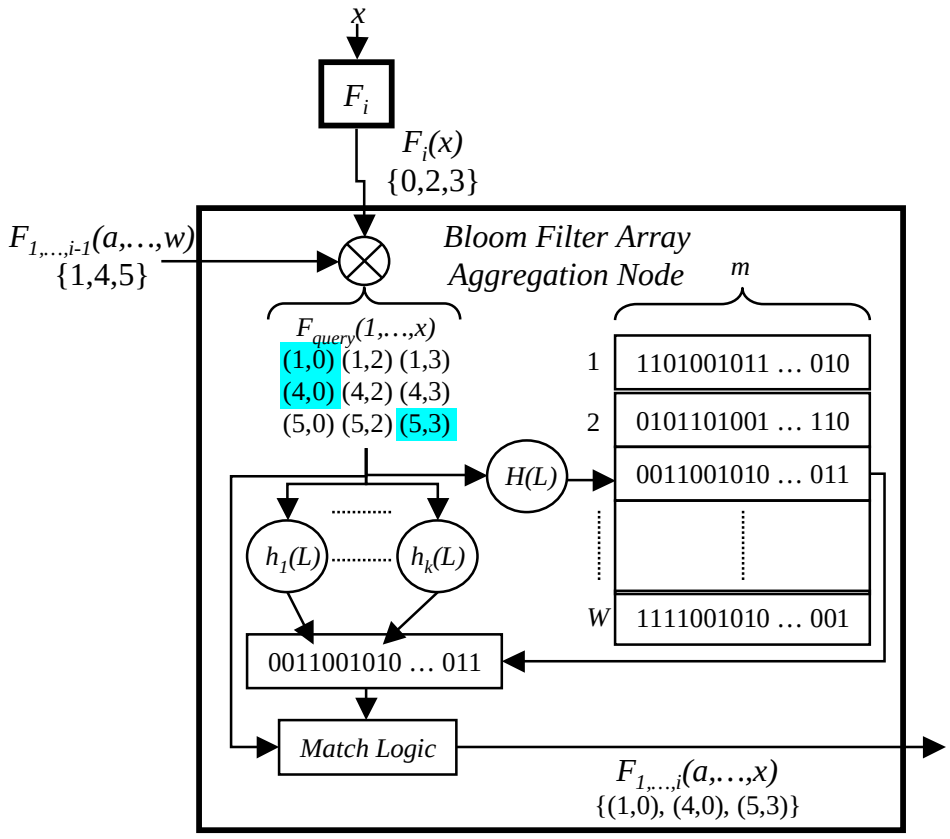

Figure 6: Example of an aggregation node using a *Bloom Filter Array* to aggregate field label set $F_i(x)$ with label set $F_{1,\ldots,i-1}(a,\ldots,w)$.

when the following relationship holds

$$k = \frac{m}{n} \ln 2 \tag{6}$$

where n is the number of labels $|F_{1,\ldots,i}|$ stored in the Bloom filter. Setting k to four produces a tolerable false positive probability of 0.06. Assuming that we store one Bloom filter per memory word, we can calculate the required memory resources given the memory word size m. Let W be the number of memory words. The hash function $H(L)$ uniformly distributes the labels in $F_{1,\ldots,i}$ across the W Bloom filters in the *Bloom Filter Array*. Thus, the number of labels stored in each Bloom filter is

$$n = \frac{|F_{1,\ldots,i}|}{W} \tag{7}$$

Using Equation 6 we can compute the number of memory words, W, required to maintain the false positive probability given by Equation 5:

$$W = \left\lceil \frac{k \times |F_{1,\ldots,i}|}{m \times \ln 2} \right\rceil \tag{8}$$

The total memory requirement is $m \times W$ bits. Recent work has provided efficient mechanisms for dynamically updating Bloom filters [14, 15].

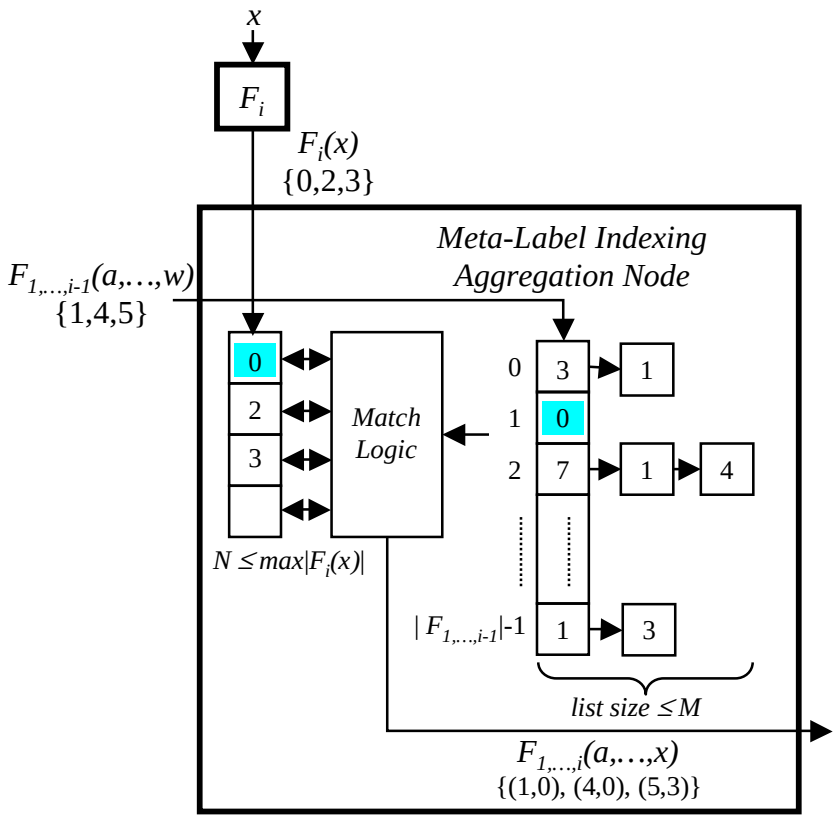

Figure 7: Example of an aggregation node using *Meta-Label Indexing* to aggregate field label set $F_i(x)$ with meta-label set $F_{1,\ldots,i-1}(a,\ldots,w)$.

6.2 Meta-Label Indexing

We can leverage the compression provided by meta-labels to construct aggregation nodes that explicitly represent the set of field value combinations, $F_{1,\ldots,i}$. The field value combinations in $F_{1,\ldots,i}$ can be identified by a composite label which is the concatenation of the meta-label for the combination of the first $(i-1)$ fields, $L_{1,\ldots,i-1}$, and the label for field i, L_i. We sort these composite labels into bins based on meta-label $L_{1,\ldots,i-1}$. For each bin, we construct a list of the labels L_i, where each entry stores L_i and the new meta-label for the combination of i fields, $L_{1,\ldots,i}$. We store these lists in an array A_i indexed by meta-label $L_{1,\ldots,i-1}$ as shown in Figure 7.

Using $L_{1,\ldots,i-1}$ as an index allows the total number of set membership queries to be limited by the number of meta-labels received from the upstream aggregation node, $|F_{1,\ldots,i-1}(a,\ldots,w)|$. Note that the size of a list entry, s, is

$$s = \lg|F_i| + \lg|F_{1,\ldots,i}| \tag{9}$$

and s is typically much smaller than the memory word size, m. In order to limit the number of memory accesses per set membership query, we store N list entries in each memory word, where $N = \lfloor \frac{m}{s} \rfloor$. This requires $N \times |F_i(x)|$ way match logic to compare all of the field labels in the memory word with the set of matching field labels from the field search engine, $F_i(x)$. Since set membership queries may be performed independently, the total number of sequential memory accesses, *SMA*, depends on the size of the index meta-label set, $|F_{1,\ldots,i-1}(a,\ldots,w)|$, the size of the lists indexed by the labels in $F_{1,\ldots,i-1}(a,\ldots,w)$, and the number of memory ports P. In the worst case, the labels index the $|F_{1,\ldots,i-1}(a,\ldots,w)|$ longest lists in A_i.

Let $Length$ be an array storing the lengths of the lists in A_i in decreasing order. The worst-case sequential memory accesses is

$$SMA = \frac{\sum_{j=1}^{|F_{1,\ldots,i-1}(a,\ldots,w)|} \left\lceil \frac{Length(j)}{N} \right\rceil}{P} \tag{10}$$

As with the *Bloom Filter Array*, the use of multi-port memory blocks does require replication of the multi-way match logic. Due to the limited number of memory ports, we argue that this represents a negligible increase in the resources required to implement *DCFL*. The number of memory words, W, needed to store the data structure is

$$W = \sum_{j=1}^{|F_{1,\ldots,i-1}|} \left\lceil \frac{Length(j)}{N} \right\rceil \tag{11}$$

The total memory requirement is $m \times W$ bits. Adding or removing a label from $F_{1,\ldots,i}$ requires an update to a single list entry. Packing multiple list entries on to a single memory word slightly complicates the memory management; however, given that we seek to minimize the number of memory words occupied by a list, the number of individual memory reads and writes per update is small.

Finally, we point out that the data structure may be re-organized to use L_i as the index. This variant, *Field Label Indexing*, is effective when $|F_x|$ approaches $|F_{1,\ldots,x}|$. When this is the case, the number of composite labels $L_{1,\ldots,i}$ containing label L_i is small and the length of the lists indexed by $F_i(x)$ are short.

7 Field Search Engines

A primary advantage of *DCFL* is that it allows each filter field to be searched by a search engine optimized for the particular type of search. While the focus of this paper is the novel aggregation technique, we briefly discuss single field search techniques suitable for use with *DCFL* in order to to highlight the potential performance.

7.1 Prefix Matching

Due to its use of decomposition, *DCFL* requires that the search engines for the IP source and destination addresses return *all* matching prefixes for the given addresses. Any longest prefix matching technique can support All Prefix Matching (APM), but some more efficiently than others. The most computationally efficient technique for longest prefix matching is *Binary Search on Prefix Lengths* [16]. When precomputation and marker optimizations are used, the technique requires at most five hash probes per lookup for 32-bit IPv4 addresses. Real filter sets contain a relatively small number of unique prefix lengths, thus the realized performance should be better for real filter sets. Recall that markers direct the search to longer prefixes that potentially match, thus skipping shorter prefixes that may match. In order to support APM, *Binary Search on Prefix Lengths* must precompute all matching prefixes for each "leaf" in the trie defined by the set of address prefixes. While computationally efficient for searches, this technique does present several challenges for hardware implementation. Likewise, the significant use of precomputation and markers degrades the dynamic update performance, as an update may require many memory transactions.

As demonstrated in [17], compressed multi-bit trie algorithms readily map to hardware and provide excellent lookup and update performance with efficient memory and hardware utilization. Specifically, our implementation of the Tree Bitmap technique requires at most 11 memory accesses per lookup and approximately six bytes of memory per prefix. Each search engine consumes less than 1% of the logic resources on a commodity FPGA. As discussed in [17], there are a number of optimizations to improve the

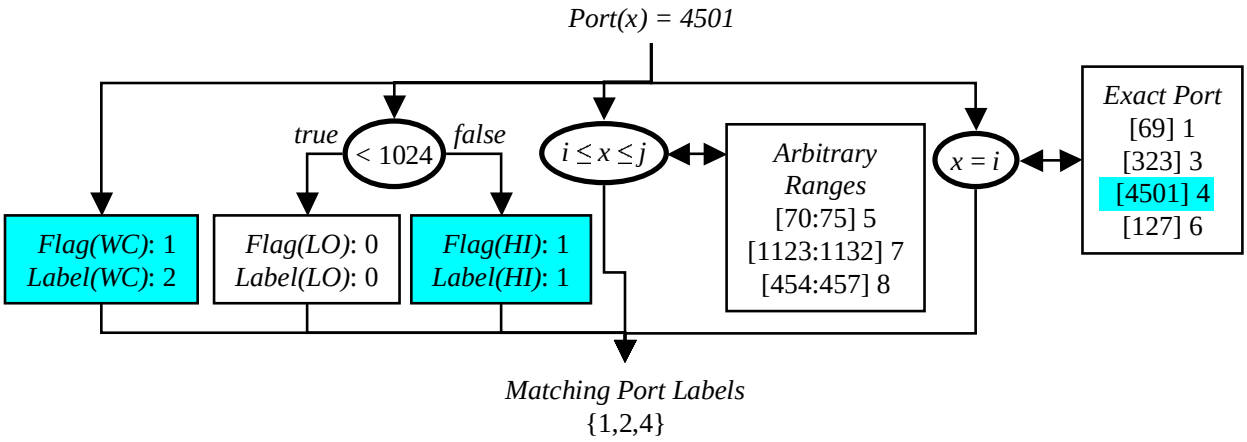

Figure 8: Block diagram of range matching using parallel search engines for each port class.

performance of this particular implementation. Use of an initial lookup array for the first 16 bits reduces the number of memory accesses to at most seven. Coupled with a simple two-stage pipeline, the number of sequential memory accesses per lookup can be reduced to at most four. Trie-based LPM techniques such as Tree Bitmap easily support all prefix matching with trivial modifications to the search algorithm. For the purpose of our discussion, we will assume an optimized Tree Bitmap implementation requiring at most four memory accesses per lookup and six bytes per prefix of memory.

7.2 Range Matching

Searching for all arbitrary ranges that overlap a given point presents a greater challenge than prefix matching. Based on the observations reported in Appendix B, range matching can be made sufficiently fast for real filter sets using a set of parallel search engines, one for each port class, as shown in Figure 8. Recall that three port classes, WC, HI, and LO, consist of a single range specification. The first port class, wildcard (WC), the search engine simply consists of flag specifying whether or not the wildcard is specified by any filters in the filter set and a register for the label assigned to this range specification. Similarly, the search engines for the HI and LO port classes also consist of flags specifying whether or not the ranges are specified by any filters in the filter set and registers for the labels assigned to those range specifications. We also add logic to check if the port is less than 1024; this checks for a match on the HI and LO port ranges, $[1024 : 65535]$ and $[0 : 1023]$, respectively.

For the 12 real filter sets we studied, the number of exact port numbers specified by filters was at most 183. The port ranges in the EM port class may be efficiently searched using any sufficiently fast exact match data-structure. Entries in this data-structure are simply the port number and the assigned label. A simple hash table could bound searches to at most two memory accesses. Finally, the set of arbitrary ranges in the AR port class may be searched with any range matching technique. Fortunately, the set of arbitrary ranges tends to be small; the 12 real filter sets specified at most 27 arbitrary ranges. A simple balanced interval tree data-structure requires at most $O(k \lg n)$ accesses, where k is the number of matching ranges and n is the number of ranges in the tree. Other options for the AR search engine include the *Fat Inverted Segment Tree* [18] and converting the arbitrary ranges to prefixes and employing an all prefix matching search engine. Given the limited number of arbitrary ranges, adding multiple prefixes per range to the data-structure does not cause significant memory inefficiency. With sufficient optimization, we assume that range matching can be performed with at most four sequential memory accesses and the data-structures for the AR and EM port

classes easily fit within a standard embedded memory block of 18kb.

7.3 Exact Matching

The protocol and flag fields may be easily searched with a simple exact match data-structure such as a hash table. Given the small number of unique protocol and flag specifications in the real filter sets (less than 9 unique protocols and 11 unique flags), the time per search and memory space required is trivial. As we discuss in Section 2, we expect that additional filter fields will also require exact match search engines. Given the ease of implementing hash functions in custom and reconfigurable logic, we do not foresee any performance bottlenecks for the search engines for these fields.

8 Dynamic Updates

Another of strength of *DCFL* is its support of incremental updates. Adding or deleting a filter from the filter set requires approximately the same amount of time as a search operation and does not require that we flush the pipeline and update all data-structures in an atomic operation. An update operation is treated as a search operation in that it propagates through the *DCFL* architecture in the same manner. The query preceding the update in the pipeline operates on data-structures prior to the update; the query following the update in the pipeline operates on data-structures following the update.

Consider inserting a filter to the filter set. We partition the filter into fields (performing field splits, if necessary) and insert each field into the appropriate input buffer of the field search engines. In parallel, each field search engine performs the update operation just as it would perform searches in parallel. As shown in Figure 9, an add operation entails a search of the data-structure for the given filter field. If the data-structure does not contain the field, then we add the field to the data-structure and assign the next free label[3]. Finally, we increment the count value for the field entry. Each field search engine returns the label for the filter field. At the next pipeline cycle, the field search engines feed the update operation and field labels to the aggregation network. Logically, the same `Insert` operation is used by both field search engines and aggregation nodes, only the type of *item* and *label* is different for the two. Each aggregation node receives the "insert" command and the labels from the upstream nodes. The *item* is the composite label formed from the labels from the upstream nodes. Note that for an update operation, field search engines and aggregation nodes only pass on one label, thus each aggregation node only operates on one composite label or *item*. If the composite label is not in the set, then the aggregation node adds it to the set. Note that the *label* returned by the `Search` or `Add` operations may be a composite label or meta-label, depending on the type of aggregation nodes in use. Finally, the aggregation increments the count for the *label* and passes it on to the next aggregation node. The final aggregation node passes the *label* on to the priority resolution stage which adds the field label to its data-structure according to its priority tag.

Removing a filter from the filter set proceeds in the same way. Both field search engines and aggregation nodes perform the same logical `Remove` operation shown in Figure 10. We first find the *label* for the *item*, then decrement the count value for the *item*. A `Delete` operation is performed if the count value for the *item* is zero. The *label* is passed on to the next node in the *DCFL* structure. The final aggregation node passes the filter label to the priority resolution stage which removes the field label from its data-structure.

Note that `Add` and `Delete` operations on field search engine and aggregation node data-structures are only performed when count values change from zero to one and one to zero, respectively. The limited

[3]We assume that each data-structure keeps a simple list of free labels that is initialized with all available labels. When labels are "freed" due to a delete operation, they are added to the end of the list.

```
Insert(item)

1  label←Search(item)
2  If (label = NULL)
3     label←Add(item)
4  Count[label]++
5  return label
```

Figure 9: Pseudocode for *DCFL* update (add).

```
Remove(item)

1  label←Search(item)
2  Count[label]−−
3  If (Count[label] = 0)
4     Delete(item)
5  return label
```

Figure 10: Pseudocode for *DCFL* update (delete).

number of unique field values in real filter sets suggests significant sharing of unique field values among filters. We expect typical updates to only change a couple field search engine data-structures and aggregation node data-structures. In the worst case, inserting or removing a filter produces an update to d field search engine data-structures and $(d-1)$ updates to aggregation node data-structures, where d is the number of filter fields.

9 Performance Evaluation

In order to evaluate the performance and scalability of *DCFL*, we used a combination of real and synthetic filter sets of various sizes and compositions. The 12 real filter sets were graciously provided from ISPs, a network equipment vendor, and other researchers in the field. *ClassBench* is a publicly available tools suite for benchmarking packet classification algorithms and devices [19]. It includes a *Filter Set Analyzer* that extracts the relevant statistics and probability distributions from a seed filter set and generates a *parameter file*. The *ClassBench Filter Set Generator* takes as input a *parameter file* and a few high-level parameters that provide high-level control over the composition of the filters in the resulting filter set. We constructed a *ClassBench parameter file* for each of the 12 real filter sets and used these files to generate large synthetic filter sets that retain the structural properties of the real filter sets. The *ClassBench Trace Generator* was used to generate input traffic for both the real filter sets and the synthetic filter sets used in the performance evaluation. For all simulations, header trace size is at least an order of magnitude larger than filter set size. The metrics of interest for *DCFL* are the maximum number of sequential memory accesses per lookup at any aggregation node, *SMA*, and the memory requirements. We choose to report the memory requirements in bytes per filter, *BpF*, in order to better assess the scalability of our technique.

The type of embedded memory technology directly influences the achievable performance and efficiency of *DCFL*; thus, for each simulation run we compute the *SMA* and total memory words required for various

memory word sizes. Standard embedded memory blocks provide 36-bit memory word widths [20, 11]; therefore, we computed results for memory word sizes of 36, 72, 144, 288, and 576 bits corresponding to using 1, 2, 4, 8, and 16 memory blocks per aggregation node. All results are reported relative to memory word size. The choice of memory word size allows us to explore the tradeoff between memory efficiency and lookup speed. We assert that the use of 16 embedded memory blocks to achieve a memory word size of 576 bits is reasonable given current technology, but certainly near the practical limit. For simplicity, we assume all memory blocks are single-port, $(P = 1)$. Given that all set membership queries are independent, the *SMA* for a given implementation of *DCFL* may be reduced by a factor of P.

In order to demonstrate the achievable performance of *DCFL*, each simulation performs lookups on all possible aggregation network constructions. At the end of the simulation, we compute the optimal aggregation network by choosing the optimal network structure and optimal node type for each aggregation node in the graph. The three node types are discussed in Section 6 along with the derivation of the equations for *SMA* and memory requirements for each type: *Bloom Filter Array*, *Meta-Label Indexing*, and *Field Label Indexing*. In the case that two node types produce the same *SMA* value, we choose the node type with the smaller memory requirements. Our simulation also allows us to select the aggregation network structure and node types in order to optimize worst-case or average-case performance. Worst-case optimal aggregation networks select the structure and node types such that the value of the maximum *SMA* for any aggregation node in the network is minimized. Likewise, average-case optimal selects the structure and node types such that the maximum value of the average *SMA* for any aggregation node in the network is minimized. Computing the optimal aggregation network at the end of the simulation allows us to observe trends in the optimal network structure and node type for filter sets of various type, structure, and size. We observe that optimal network structure and node type largely depends on filter set structure. With few exceptions, variables such as filter set size and memory word size do not affect the composition of the optimal aggregation network. We observe that the *Bloom Filter Array* technique is commonly selected as the optimal choice for the first one or two nodes in the aggregation network. With rare exceptions, *Meta-Label Indexing* is chosen for aggregation nodes at the end of the aggregation network. This is a convenient result, as the final aggregation node in the network cannot use the *Bloom Filter Array* technique in order to ensure correctness. We find this result to be somewhat intuitive since the size of a meta-label increases with the number of unique combinations in the set which typically increases with the number of fields in the combination. When using meta-labels to index into an array of lists, a larger meta-label addresses a larger space which in turn "spreads" the labels across a larger array and limits the length of the lists at each array index.

In the first set of tests we used the 12 real filter sets and generated header traces using the *ClassBench Trace Generator*. The number of headers in the trace was 50 times the number of filters in the filter set. As shown in Figure 11(a), the worst-case *SMA* for all 12 real filter sets is ten or less for a worst-case optimal aggregation network using memory blocks with a word size of 288 bits. Also note that the largest filter set, *acl5*, of 4557 filters achieves the best performance with a worst-case *SMA* of two for worst-case optimal aggregation network using memory blocks with a word size of 144 bits. In order to translate these results into achievable lookup rates, assume a current generation logic device with dual-port memory blocks, $(P = 2)$, operating at 500 MHz. The worst-case *SMA* for all 12 filter sets is then five or less using a word size of 288 bits. Under these assumptions, the pipeline cycle time can be 10ns allowing the *DCFL* implementation to achieve 100 million searches per second which is comparable to current TCAMs. Search performance can be doubled by doubling the clock frequency or using quad-port memory blocks, both of which are possible in current generation ASICs.

As shown in Figure 11(c), the average *SMA* for all filter sets falls to four or less using a memory word size of 288 bits. Filter set *acl5* also achieves the best average performance with an average *SMA* of 1.2 for

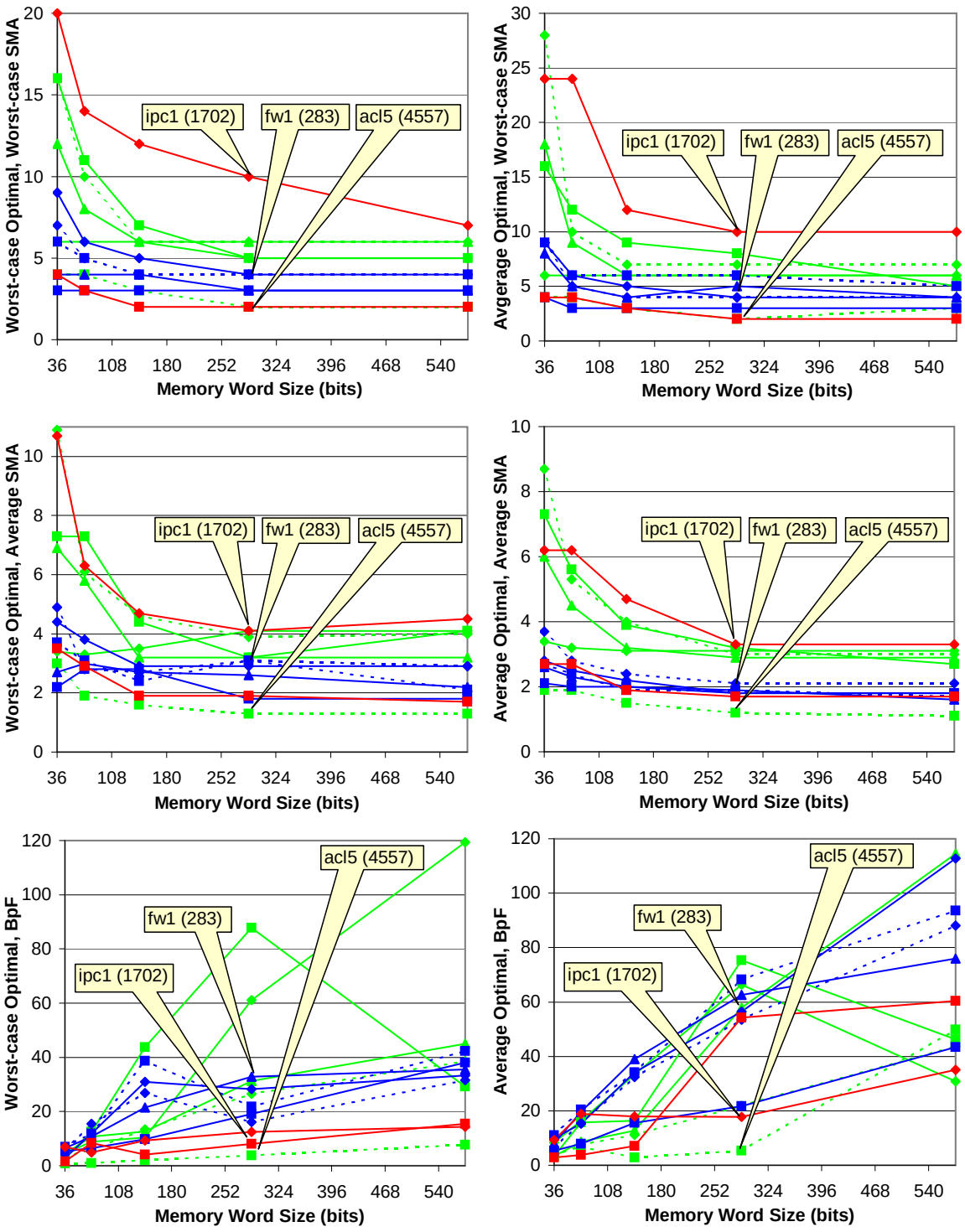

Figure 11: Performance results for 12 real filter sets; left-column shows worst-case sequential memory accesses (SMA), average SMA, and memory requirements in bytes per filter (BpF) for aggregation network optimized for worst-case SMA; right-column shows same results for aggregation network optimized for average-case SMA; call-outs highlight three specific filter sets of various sizes and types (filter set size given in parentheses).

a word size of 288. As in many other packet classification techniques, average performance is significantly better than worst-case performance.

Worst-case optimal memory consumption is shown in Figure 11(e). Most filter sets required at most 40 bytes per filter (*BpF*) for all word sizes; thus, 1MB of embedded memory would be sufficient to store 200k filters. There are two notable exceptions. The results for filter set *acl1* show a significant increase in memory requirements for larger word sizes. For memory word sizes of 36, 72, and 144 bits, *acl1* requires less than 11 bytes per filter; however, memory requirements increase to 61 and 119 bytes per filter for word sizes 288 and 576, respectively. We also note that increasing the memory word size for *acl1* yields no appreciable reduction in *SMA*; all memory word sizes yielded an *SMA* of five or six. These two pieces of data suggest that in the aggregation node data-structures, the size of the lists at each index entry are short; thus, increasing the memory word-size linearly increases the memory inefficiency without yielding any fewer memory accesses. We believe that this is also the case with the optimal aggregation network for *acl2* with memory word size 288.

Figure 11(b) shows the worst-case *SMA* for all 12 real filter sets for an average-case optimal aggregation network. Figure 11(d) shows the average *SMA* for all 12 real filter sets for an average-case optimal aggregation network. When optimizing for average *SMA*, average performance is improved by approximately 25%, but worst-case performance suffers by approximately 50%. With the exception of rare application environments, sacrificing worst-case performance for average performance is unfavorable. For the remaining simulations, we only report worst-case optimal results.

The second set of simulations investigates the scalability of *DCFL* to larger filter sets. Results are shown in Figure 12. This set of simulations utilized the *ClassBench* tools suite to generate synthetic filter sets containing 10k, 20k, and 50k filters using *parameter files* extracted from filter sets *acl5*, *fw1*, and *fw5*. As shown in Figure 12(a), the worst-case *SMA* is eight or less for all filter sets using memory word sizes of 72 bits or more. The most striking feature of these results is the indistinguishable difference between filter set sizes of 20k and 50k. The *ClassBench Synthetic Filter Set Generator* maintains the field overlap properties specified in the *parameter file*. Coupled with the results in Figure 12, this confirms that the property of filter set structure most influential on *DCFL* performance is the maximum number of unique field values matching any packet header field. As discussed in Section 2, we expect this property to hold as filter sets scale in size. If field overlap does increase, the *Field Splitting* optimization provides a way to reduce this to a desired threshold. As shown in Figure 12(c), the memory requirements increase with memory word size. Given the favorable *SMA* performance there is no need to increase the word size beyond 72, as it only results in a linear increase in memory inefficiency. Clearly, finding the optimum balance of lookup performance and memory efficiency requires careful selection of memory word size.

The third set of simulations investigates the effect of filter scope on the performance of *DCFL*. Recall that scope is measure of the specificity of the filters in the filter set. *ClassBench* provides high-level control over the average scope of the filters in the filter set via an input parameter s. We generated synthetic filter sets containing 16000 filters using *parameter files* from a variety of filter sets.For each *parameter file*, we generated filter sets using scope parameters -1, 0, and 1. The scope parameter had the most pronounced effects on worst-case *SMA* for the filter sets generated with the *parameter file* from *ipc1*. As shown in Figure 13(a), decreasing the average scope of the filters in the filter set ($s = -1$) results in significantly better performance; thus, as filters become more specific the performance of *DCFL* improves. This is a favorable result given the generally accepted conjecture the primary source of future filter set growth will be flow specific filters for applying network services. If we increase the scope of the filters in the filter set, *DCFL* performance suffers. This trend also holds for the average *SMA*. As shown in Figure 13(c), filter set specificity has little effect on memory requirements for memory word sizes of 144 bits or less. When using larger memory word sizes, the results diverge but no clear trend is evident.

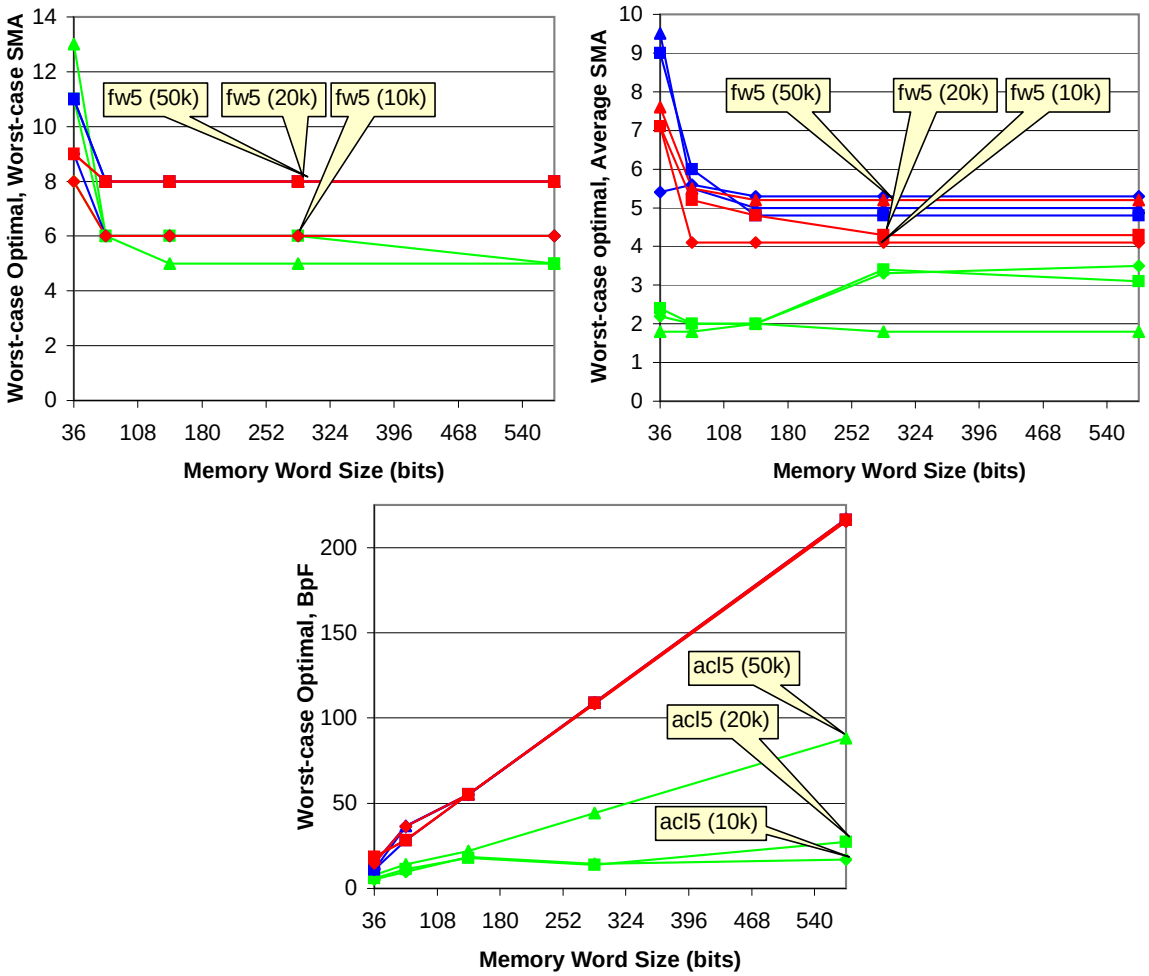

Figure 12: Performance results for synthetic filter sets containing 10k, 20k, and 50k filters, generated with parameter files from filter sets *acl5*, *fw1*, and *fw5*; call-outs highlight most pronounced effects (number of filters given in parentheses).

The fourth set of simulations investigate the efficacy and consequences of the *Field Splitting* optimization. We selected two of the worst-performing real filter sets and performed simulations with various field overlap thresholds. The performance results are summarized in Figure 14. For *acl2*, *Field Splitting* reduces the worst-case *SMA* from 16 to 12 for 36-bit memory words and 11 to 8 for 74-bit memory words. This amounts to a 33% performance increase; however, the impact of *Field Splitting* is reduced as we increase memory word size. Clearly, the primary benefit of *Field Splitting* is that it allows us to achieve better performance using smaller memory word sizes which improves the memory efficiency. As shown in Figure 14(c), the memory utilization for all filter sets using memory word sizes of 74-bits or less remains well-below 40 bytes per filter. Consider the specific case of *acl2*. In order to achieve a worst-case *SMA* of eight or less without *Field Splitting*, we must use a memory word-size of 144 bits resulting in memory requirements of 44 bytes per filter. Using *Field Splitting* with a field overlap threshold of four, we achieve the desired worst-case *SMA* performance using a memory word-size of 72 bits resulting in memory requirements of 32 bytes per filter. Recall that *Field Splitting* does increase the number of aggregation nodes in the aggregation network, thus increasing the number of memory blocks and logic required for implementation. However, these results show that the total memory requirements are actually reduced for a particular performance

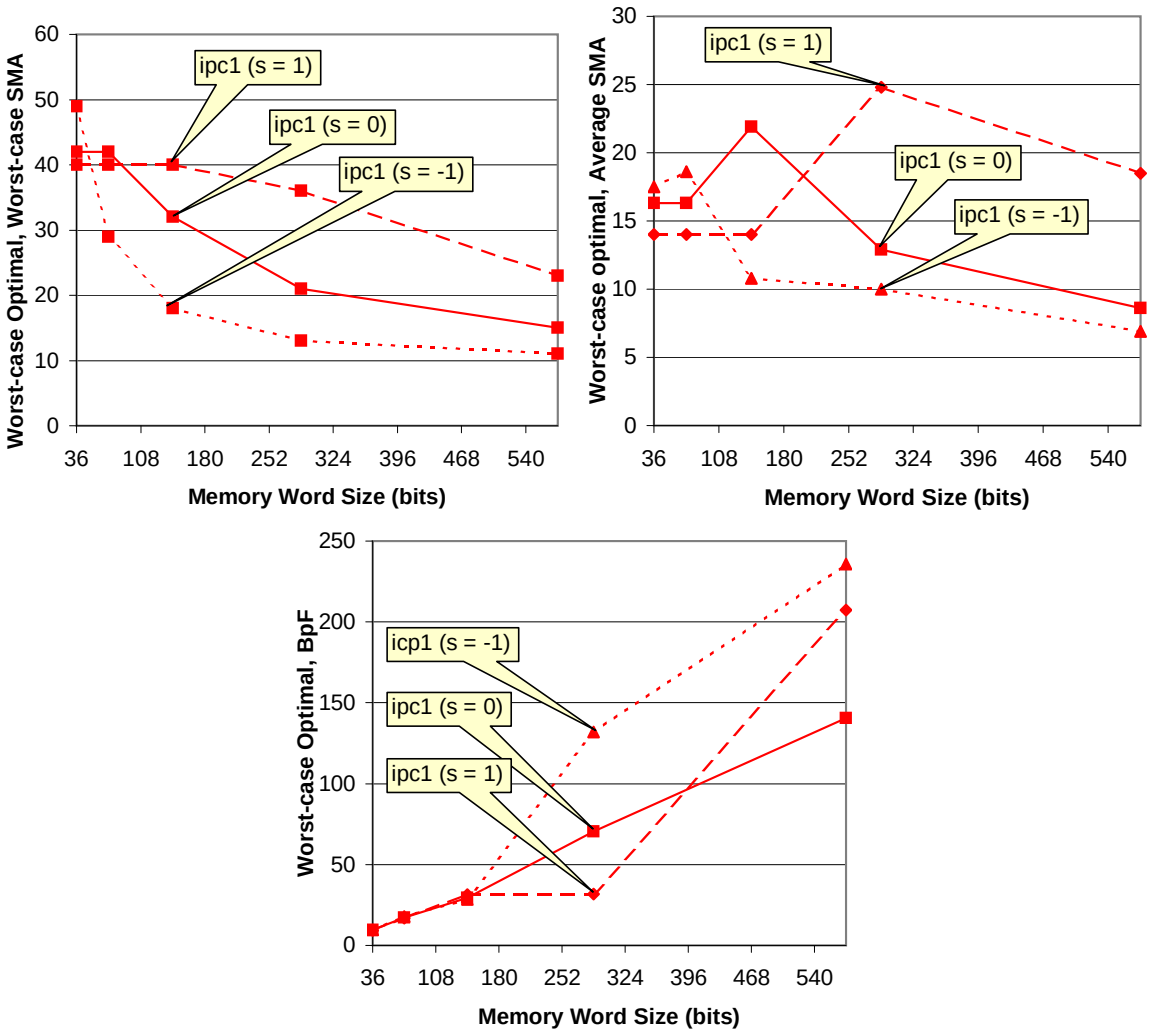

Figure 13: Performance results for synthetic filter sets containing 16k filters, generated with the *ipc1 parameter file* with scope parameters s {-1,0,1}; call-outs highlight most pronounced effects (scope parameter given in parentheses).

target. It is important to note that we do reach a point of diminishing returns with *Field Splitting*. The aggregation network can grow too large if too many splits are required to achieve a particularly low field overlap threshold. In this case, the impact on worst-case *SMA* is minimal while the memory resource requirements increase drastically due to the additional overhead. This situation is reflected in Figure 14(c) for filter set *fw5* with a field overlap threshold of three and memory word size of 288 bits.

The fifth and final set of simulations investigate the scalability of *DCFL* to additional filter fields. Using the *ClassBench* tools suite, we generated three filter sets containing 16000 filters using the *acl5 parameter file*. No *smoothing* or *scope* adjustments were applied. The first filter set was generated such that half of the filters specifying the TCP or UDP protocols specified one non-wildcard field in addition to the standard six filter fields (the 5-tuple plus protocol flags). The non-wildcard field value was selected from a set of 100 random values using a uniform random variable. The second and third filter sets were generated in the same manner with two and four extra field values, respectively. Results from simulation runs are shown in Figure 15. The slight improvement in worst-case *SMA* is attributable to two impetuses: (1) the additional

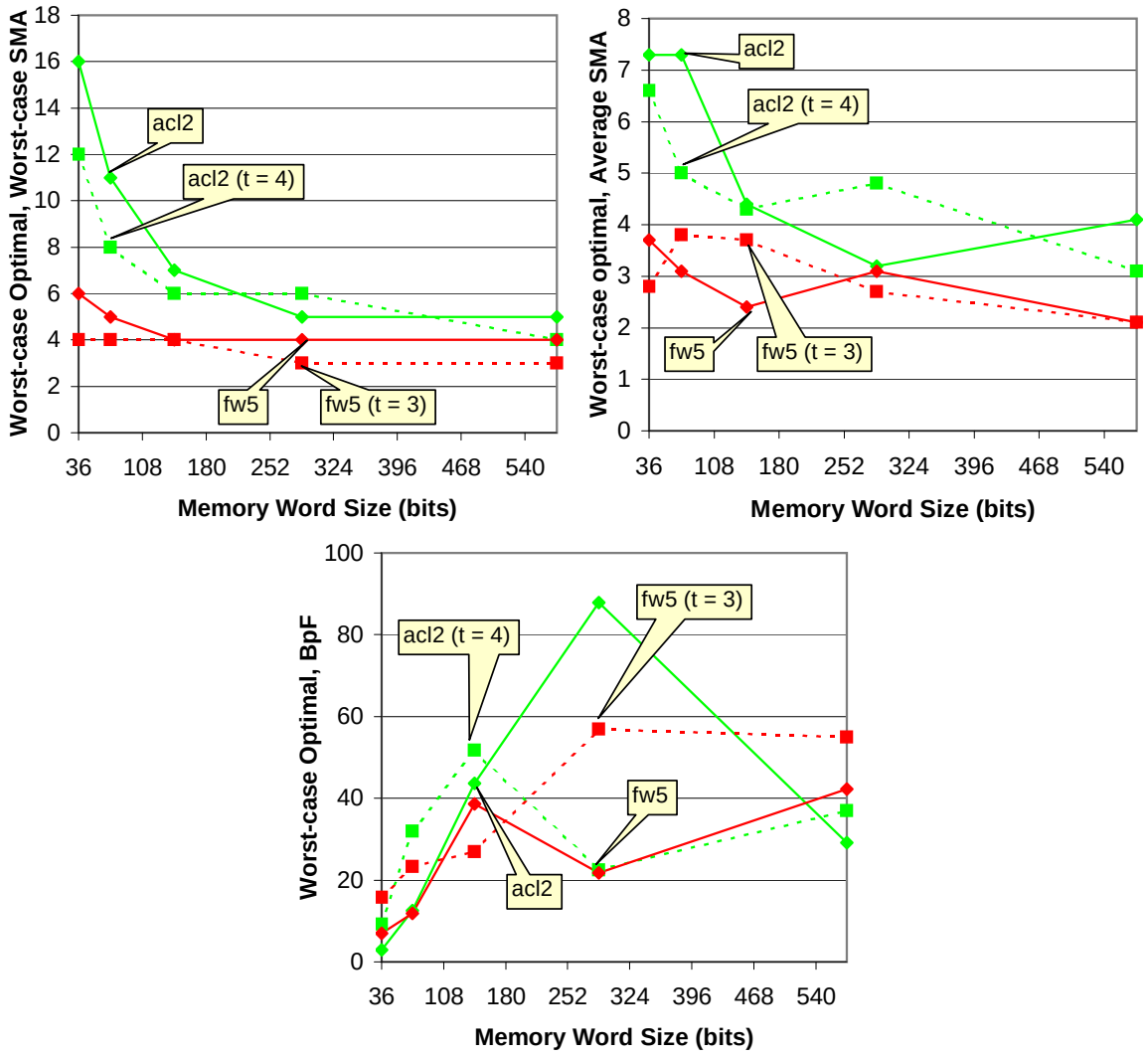

Figure 14: Performance results for four real filter sets (*acl2*, *fw1*, *fw4*, and *fw5*) using the *Field-Splitting* optimization; call-outs highlight most pronounced effects (field overlap threshold given in parentheses).

filter fields allow filters to be more specific, and (2) the additional filter fields are exact match fields and the maximum fields overlap is at most two. As reflected in Figure 15(c), the increase in memory requirements for an additional filter field is small for memory word sizes of 144 bits or less. Specifically, when using 144-bit memory words the memory requirements increase by 18 bytes per filter when adding a seventh field, 17 bytes per filter when adding an eighth filter field, and 3 bytes per filter when adding the ninth filter field. This is constitutes an average of 12.5 bytes per filter for each additional field. Given our reasonable assumptions regarding the nature of additional filter fields in future filter sets, we assert that the performance and scalability of *DCFL* will make it an even more compelling solution for packet classification as filter sets scale in size and the number of filter fields.

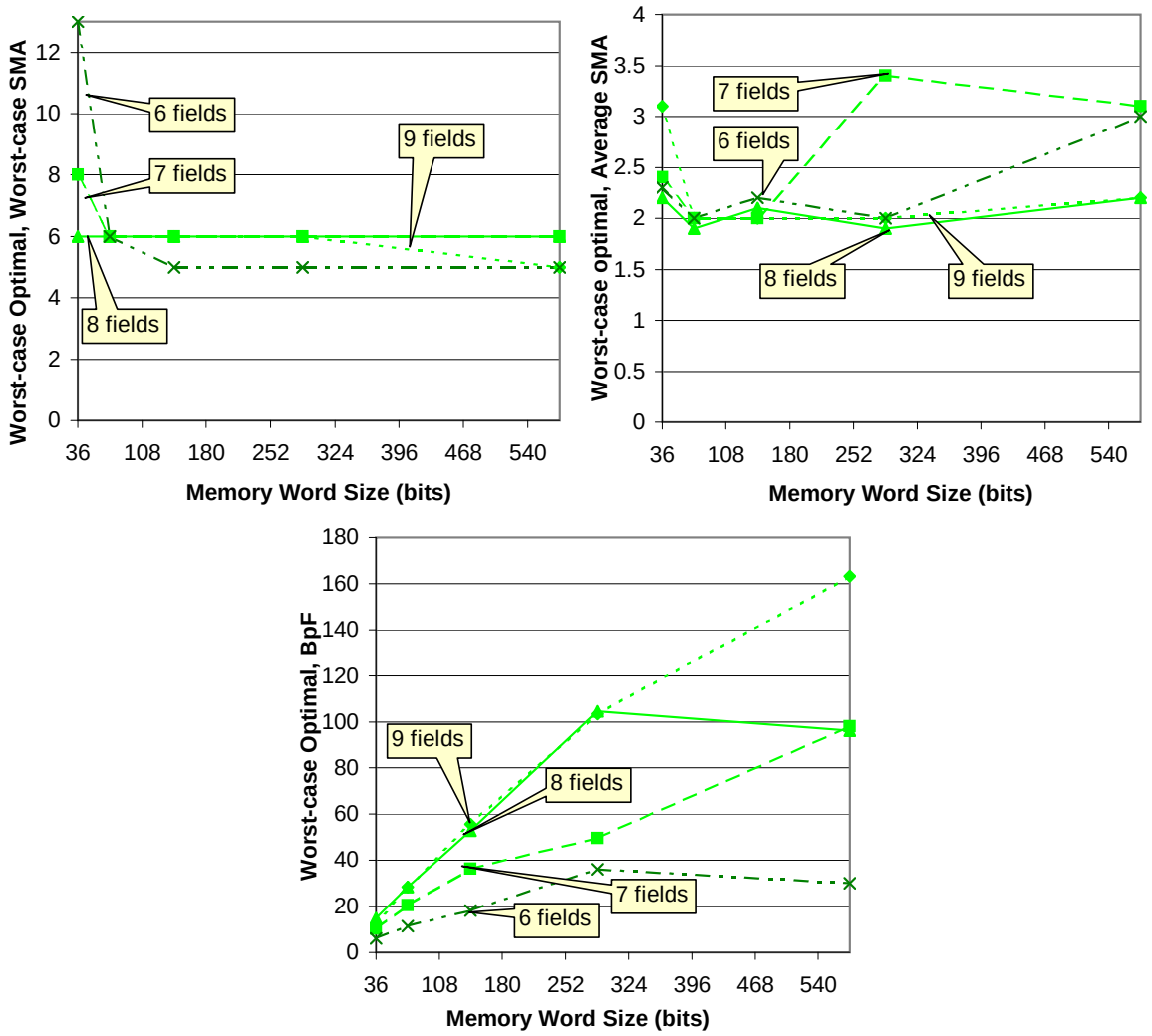

Figure 15: Performance results for synthetic filter sets containing 16k filters, generated with parameter file from filter set *acl5* with extra filter fields; call-outs highlight most pronounced effects (number of filter fields given in parentheses).

10 Related Work

In general, there have been two major threads of research efforts addressing the packet classification problem: algorithmic and architectural. A few pioneering groups of researchers posed the problem, provided complexity bounds, and offered a collection of algorithmic solutions [2, 21, 22, 9]. Subsequently, the design space has been thoroughly explored by many offering new algorithms and improvements upon existing algorithms [4, 18, 6]. 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) [23, 24, 25, 26]. While they provide sufficient speed, current TCAM-based solutions consume exorbitant amounts of power and hardware resources relative to implementations of efficient algorithms. Recent work has addressed many of the unfavorable aspects of current TCAM-based solutions [27, 28]; however, there remain fundamental limits to their scalability and efficiency.

The most promising algorithmic research embraces the practice of leveraging the statistical structure of filter sets to improve average performance [2, 4, 3, 21, 29]. Several algorithms in this class are amenable to high-performance hardware implementation. New architectural research combines intelligent algorithms and novel architectures to eliminate many of the unfavorable characteristics of current TCAMs [28]. We observe that the community appears to be converging on a combined algorithmic and architectural approach to the problem [5]. Our solution, *Distributed Crossproducting of Field Labels* (*DCFL*), employs this combined approach to provide a scalable, high-performance packet classifier. [1] provides a thorough survey of packet classification techniques using a taxonomy that frames each technique according to its high-level approach. In this section, we highlight the sources of the key ideas and data structures which we distill and utilize in *DCFL*. In order to demonstrate the value of our solution relative to the state of the art, we also contrast it with two leading solutions which are arguably the top solutions from the algorithmic and architectural threads.

As clearly indicated by the name, *DCFL* draws upon the seminal *Crossproducting* technique introduced by Srinivasan, Varghese, Suri, and Waldvogel [9]. *DCFL* avoids the exponential blowup in memory requirements experienced by *Crossproducting* by only storing the labels for field values and combinations of field values present in the filter table. It retains high-performance by aggregating intermediate results in a distributed fashion. Gupta and McKeown introduced *Recursive Flow Classification* (*RFC*) which provides high lookup rates at the cost of memory inefficiency [2]. 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. There is a subtle, yet powerful difference between the use of equivalence classes in *RFC* and field labels in *DCFL*. In essence, the number of labels in *DCFL* grows linearly with the number of unique field values in the filter table. 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. Another major difference between *DCFL* and *RFC* is the means of aggregating intermediate results. *RFC* lookups in "chunk" and aggregation tables utilize indexing, causing *RFC* to make very inefficient use of memory. The index tables used for aggregation also require significant precomputation in order to assign the proper *eqID* for the combination of the *eqID*s of the previous phases. Such extensive precomputation precludes dynamic updates at high rates. As we have shown, *DCFL* uses efficient set membership data structures which can be engineered to provide fast lookup and update performance. Each data structure only stores labels for unique field combinations present in the filter table; hence, they make efficient use of memory and do not require significant precomputation. In order to illustrate the differences between *RFC* and *DCFL*, we provide an example of an *RFC* search for two "chunks" of a search on n "chunks" in Figure 16. The squares $[a \ldots l]$ represent the unique projections of the two "chunks" x and y for all filters in a filter table. The number of *eqIDs* for the "chunk" lookups is 11 for each dimension x and y, as 11 unique sets of filters are formed by the projections onto the x and y axes. Since *RFC* utilizes indexing for lookups, each "chunk" table requires 2^b entries, where b is the size in bits of the "chunk". Note that if the number of unique projections were *labeled* as in *DCFL*, only six labels for each dimension would be required, and the set membership data structure would only need to store six entries. In order for *RFC* to aggregate the *eqIDs* from "chunks" x and y, it must compute all of the unique sets of filters for the two-dimensional overlaps. As shown in Figure 16, this results in 25 *eqIDs*. The aggregation table requires $2^{4+4} = 256$ entries, as *eqID(x)* and *eqID(y)* are four bits in size and *RFC* utilizes indexing to find *eqID(x,y)*. Note that in *DCFL*, a label would simply be assigned to each unique 2-d projection $[a \ldots l]$ and stored in a set membership data structure. In general, *DCFL* can provide line-speed lookups, like *RFC*, but with much more efficient use of memory and support for dynamic updates at high rates.

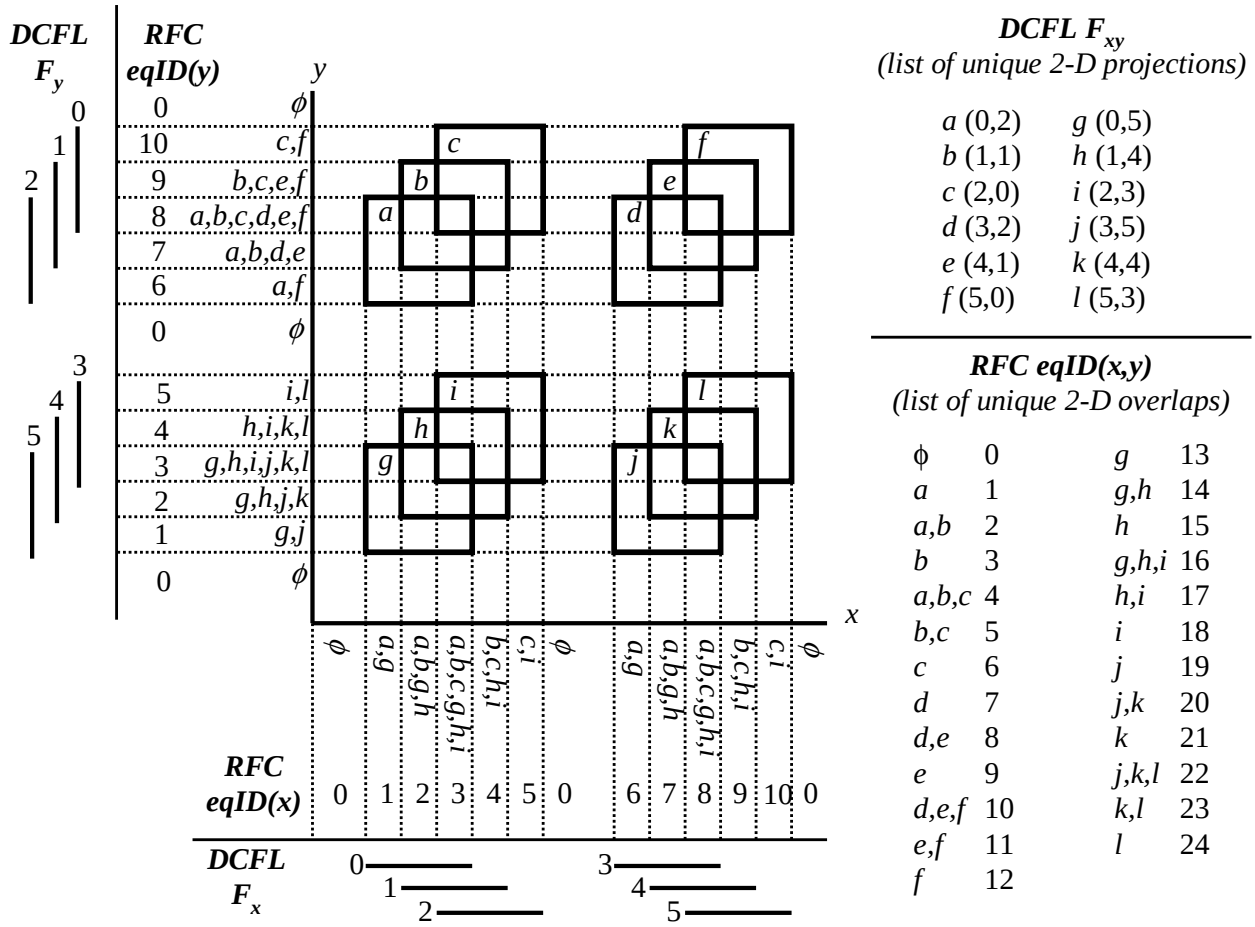

Figure 16: Contrast between unique field value labels in *Distributed Crossproducting of Field Labels* (*DCFL*) and equivalence class identifiers (*eqIDs*) in *Recursive Flow Classification*; example shows two fields of a d field search. Squares $[a \ldots l]$ represent the unique projections of two fields x and y for all filters in a filter table.

Our approach also shares similarities with the *Parallel Packet Classification* (P^2C) scheme introduced by van Lunteren and Engbersen [5]. Specifically, both *DCFL* and P^2C fall into the class of techniques using independent field searches coupled with novel encoding and aggregation of intermediate results. The primary advantage of *DCFL* over P^2C is its use of SRAM and amenability to implementation in commodity hardware technology; P^2C requires the use of a separate TCAM or a custom ASIC with embedded TCAM. *DCFL* also provides more efficient support of dynamic updates.

Given the volume of work in packet classification, we must show how our technique adds value to the state of the art. In our opinion, *HyperCuts* is one of the most promising new algorithmic solutions [29]. Introduced by Singh, Baboescu, Varghese, and Wang, the algorithm improves upon the *HiCuts* algorithm developed by Gupta and McKeown [21] and also shares similarities with the *Modular Packet Classification* algorithms introduced by Woo [6]. In essence, *HyperCuts* is a decision tree algorithm that attempts to minimize the depth of the tree by selecting "cuts" in multi-dimensional space that optimally segregate packet filters into lists of bounded size. According to performance results given in [29], traversing the *HyperCuts* decision tree required between 8 and 35 memory accesses, and memory requirements for the decision tree ranged from 5.4 to 145.9 bytes per filter. We assert that *DCFL* exhibits advantages in all metrics of interest:

worst-case *SMA*, memory requirements, and dynamic update performance. *DCFL* also provides the opportunity to strike a favorable tradeoff between performance and memory requirements, as the various parameters may be tuned to achieve the desired results. All new algorithmic approaches must make a strong case for their advantage relative to Ternary Content Addressable Memory (TCAM). Due to its performance, efficiency, scalability, and use of commodity hardware technology, *DCFL* has the ability to provide equivalent lookup performance at much lower cost and power consumption.

11 Conclusions

By transforming the problem of aggregating results from independent field search engines into a distributed set membership query, *Distributed Crossproducting of Field Labels* (*DCFL*) avoids the exponential increases in time and memory required by previous approaches. We introduced several new concepts including field labeling, *Meta-labeling* unique field combinations, and *Field Splitting*, as well as optimized set membership data structures such as *Bloom Filter Arrays* that minimize the number of memory accesses required to perform a set membership query. Using a combination of real and synthetic filter sets, we demonstrated that *DCFL* can achieve over 100 million searches per second using existing hardware technology. Furthermore, we have also shown that *DCFL* retains its lookup performance and memory efficiency when the number of filters and number of fields in the filters increases. Scalability to classify on additional fields is a distinct advantage *DCFL* exhibits over existing decision tree algorithms and TCAM-based solutions. We continue to explore optimizations to improve the search rate and memory efficiency of *DCFL*. We also believe that *DCFL* has potential value for other searching tasks beyond traditional packet classification.

References

[1] D. E. Taylor, "Survey & Taxonomy of Packet Classification Techniques," Tech. Rep. WUCSE-2004-24, Department of Computer Science & Engineering, Washington University in Saint Louis, May 2004.

[2] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *ACM Sigcomm*, August 1999.

[3] F. Baboescu, S. Singh, and G. Varghese, "Packet Classification for Core Routers: Is there an alternative to CAMs?," in *IEEE Infocom*, 2003.

[4] F. Baboescu and G. Varghese, "Scalable Packet Classification," in *ACM Sigcomm*, August 2001.

[5] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 560–571, May 2003.

[6] T. Y. C. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," in *IEEE Infocom*, March 2000.

[7] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Directions in Packet Classification for Network Processors," in *Second Workshop on Network Processors (NP2)*, February 2003.

[8] C. Bormann, et. al., "RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed." RFC 3095, July 2001. IETF Network Working Group.

[9] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel, "Fast and Scalable Layer Four Switching," in *ACM Sigcomm*, June 1998.

[10] Xilinx, "Virtex-II Pro Platform FPGAs: Introduction and Overview." DS083-1 (v3.0), December 2003.

[11] IBM Blue Logic, "Embedded SRAM Selection Guide," November 2002.

[12] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. McGraw-Hill Book Company, 1990.

[13] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Proceedings of 40th Annual Allerton Conference*, October 2002.

[14] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 281–293, June 2000.

[15] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching using Bloom Filters," in *ACM SIGCOMM'03*, August 2003.

[16] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing table lookups," in *Proceedings of ACM SIGCOMM '97*, pp. 25–36, September 1997.

[17] D. E. Taylor, J. S. Turner, J. W. Lockwood, T. S. Sproull, and D. B. Parlour, "Scalable IP Lookup for Internet Routers," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 522–534, May 2003.

[18] A. Feldmann and S. Muthukrishnan, "Tradeoffs for Packet Classification," in *IEEE Infocom*, March 2000.

[19] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," Tech. Rep. WUCSE-2004-28, Department of Computer Science & Engineering, Washington University in Saint Louis, May 2004.

[20] Xilinx, "Virtex-II Platform FPGAs: Introduction and Overview." DS031-1 (v2.0), August 2003.

[21] P. Gupta and N. McKeown, "Packet Classification using Hierarchical Intelligent Cuttings," in *Hot Interconnects VII*, August 1999.

[22] T. V. Lakshman and D. Stiliadis, "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching," in *ACM SIGCOMM'98*, September 1998.

[23] R. A. Kempke and A. J. McAuley, "Ternary CAM Memory Architecture and Methodology." United States Patent 5,841,874, November 1998. Motorola, Inc.

[24] G. Gibson, F. Shafai, and J. Podaima, "Content Addressable Memory Storage Device." United States Patent 6,044,005, March 2000. SiberCore Technologies, Inc.

[25] A. J. McAulay and P. Francis, "Fast Routing Table Lookup Using CAMs," in *IEEE Infocom*, 1993.

[26] R. K. Montoye, "Apparatus for Storing "Don't Care" in a Content Addressable Memory Cell." United States Patent 5,319,590, June 1994. HaL Computer Systems, Inc.

[27] D. Shah and P. Gupta, "Fast incremental updates on ternary-cams for routing lookups and packet classification," in *Hot Interconnects (HotI-8)*, p. 6.1, Aug. 2000.

[28] E. Spitznagel, D. Taylor, and J. Turner, "Packet Classification Using Extended TCAMs," in *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, 2003.

[29] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," in *Proceedings of ACM SIGCOMM'03*, August 2003. Karlsruhe, Germany.

A Introduction to Bloom Filters

A Bloom filter is essentially a bit-vector of length m used to efficiently represent a set of messages. Given a set of messages X with n members, the Bloom filter is "programmed" as follows. For each message x_i in X, k hash functions are computed on x_i producing k values each ranging from 1 to m. Each of these values address a single bit in the m-bit vector, hence each message x_i causes k bits in the m-bit vector to be set to 1. Note that if one of the k hash values addresses a bit that is already set to 1, that bit is not changed. Querying the filter for set membership of a given message x is similar to the programming process. Given message x, k hash values are generated using the same hash functions used to program the filter. The bits in the m-bit long vector at the locations corresponding to the k hash values are checked. If at least one of the k bits is 0, then the message is declared to be a non-member of the set. If all the bits are found to be 1, then the message is said to belong to the set with a certain probability. If all the k bits are found to be 1 and x is not a member of X, then it is said to be a false positive. This ambiguity in membership comes from the fact that the k bits in the m-bit vector can be set by any of the n members of X. Thus, finding a bit set to 1 does not necessarily imply that it was set by the particular message being queried. However, finding a 0 bit certainly implies that the the message does not belong to the set, since if it were a member then all k-bits would have been set to 1 when the Bloom filter was programmed.

The following is a derivation of the false positive probability. The probability that a random bit of the m-bit vector is set to 1 by a hash function is simply $\frac{1}{m}$. The probability that it is not set is $1-\frac{1}{m}$. The probability that it is not set by any of the n members of X is $(1-\frac{1}{m})^n$. Since each of the messages sets k bits in the vector, it becomes $(1-\frac{1}{m})^{nk}$. Hence, the probability that this bit is found to be 1 is $1-(1-\frac{1}{m})^{nk}$. For a message to be detected as a possible member of the set, all k bit locations generated by the hash functions need to be 1. The probability that this happens, f, is given by

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \tag{12}$$

for large values of m the above equation reduces to

$$f \approx \left(1 - e^{\frac{-nk}{m}}\right)^k \tag{13}$$

Since this probability is independent of the input message, it is termed the *false positive* probability. The false positive probability can be reduced by choosing appropriate values for m and k for a given size of the member set, n. For a given ratio of $\frac{m}{n}$, the false positive probability can be reduced by increasing the number of hash functions, k. In the optimal case, when false positive probability is minimized with respect to k, we get the following relationship

$$k = \frac{m}{n} \ln 2 \tag{14}$$

The false positive probability at this optimal point is given by

$$f = \left(\frac{1}{2}\right)^k \tag{15}$$

It should be noted that if the false positive probability is to be fixed, then the size of the filter, m, needs to scale linearly with the size of the message set, n.

B Port Range Analysis

We examined the port ranges specified by filters in the 12 real filter sets and the distribution of filters over the unique values. In order to observe trends among the various filter sets, we define five classes of port

Table 4: Distribution of filters over the five port classes for source and destination port range specifications; values given as percentage (%) of filters in the filter set.

Set	*Source Port*					*Destination Port*				
	WC	*HI*	*LO*	*AR*	*EM*	*WC*	*HI*	*LO*	*AR*	*EM*
acl1	100.0	0.00	0.00	0.00	0.00	30.42	0.00	0.00	11.60	57.98
acl2	100.0	0.00	0.00	0.00	0.00	69.34	0.64	0.00	7.06	22.95
acl3	99.92	0.00	0.00	0.00	0.08	9.25	13.96	0.00	11.04	65.75
acl4	99.93	0.00	0.00	0.00	0.07	8.56	12.15	0.00	11.21	68.08
acl5	100.0	0.00	0.00	0.00	0.00	30.00	4.08	0.00	5.20	60.72
fw1	77.74	8.13	0.00	0.35	13.78	31.10	8.13	0.00	0.35	60.42
fw2	38.24	17.65	0.00	0.00	44.12	100.0	0.00	0.00	0.00	0.00
fw3	77.72	5.98	0.00	0.54	15.76	27.72	5.98	0.00	0.54	65.76
fw4	10.98	42.05	10.98	1.52	34.47	13.26	18.94	0.76	1.14	65.91
fw5	75.62	5.00	0.00	0.62	18.75	35.62	3.75	0.00	1.25	59.38
ipc1	82.84	0.35	0.00	2.00	14.81	55.46	6.52	0.00	2.53	35.49
ipc2	73.96	0.00	0.00	0.00	26.04	73.96	0.00	0.00	0.00	26.04
AVG	**78.08**	**6.60**	**0.92**	**0.42**	**13.99**	**40.39**	**6.18**	**0.06**	**4.33**	**49.04**

ranges:

- WC, wildcard
- HI, ephemeral user port range $[1023 : 65535]$
- LO, well-known system port range $[0 : 1023]$
- AR, arbitrary range
- EM, exact match

Motivated by the allocation of port numbers, the first three classes represent common specifications for a port range. The last two classes may be viewed as partitioning the remaining specifications based on whether or not an exact port number is specified. Table 4 shows the distribution of filters over the five port classes for both source and destination ports. We observe some interesting trends in the data. With rare exception, the filters in the ACL filter sets specify the wildcard for the source port. A majority of filters in the ACL filters specify an exact port number for the destination port. Source port specifications in the other filter sets are also dominated by the wildcard, but a considerable portion of the filters specify an exact port number. Destination port specifications in the other filter sets share the same trend, however the distribution between the wildcard and exact match is a bit more even. After the wildcard and exact match, the HI port class is the most common specification. A small portion of the filters specify an arbitrary range, 4% on average and at most 12%. Only one filter set contained filters specifying the LO port class for either the source or destination port range.

In the interest of designing efficient data structures, we now examine the number of unique specifications in the AR and EM classes. Checking for matches in the first three classes is trivial. As shown in Table 5, the number of unique specifications in the AR class is small relative to the size of the filter set. Consisting of 50 ranges, the largest set of arbitrary ranges may be efficiently searched using a simple interval tree. Likewise

Table 5: Number of unique specifications in the Arbitrary Range (AR) and Exact Match (EM) port classes for source and destination port ranges.

Set	*Size*	*Source Port*		*Destination Port*	
		AR	*EM*	*AR*	*EM*
acl1	733	0	0	34	73
acl2	623	0	0	1	24
acl3	2400	0	2	36	152
acl4	3061	0	2	50	183
acl5	4557	0	0	3	35
fw1	283	1	10	1	40
fw2	68	0	7	0	0
fw3	184	1	6	1	36
fw4	264	3	22	3	44
fw5	160	1	8	2	29
ipc1	1702	5	27	7	45
ipc2	192	0	2	0	2

the number of specifications in the EM class is also small, thus a simple hash table would be sufficient to search this set of ranges.

The combination of source and destination port range specifications has a significant effect on several packet classification techniques. This is especially true of TCAM due to the need to convert arbitrary range pairs into pairs of prefixes. In order to assess the effect of this conversion, we computed the number of TCAM entries required to store each filter set. We refer to the *Expansion Factor* as the ratio of TCAM entries to filter set size, which can be thought of as the average number of TCAM entries required by each filter in the filter set. As shown in Table 6, a filter set may require that a TCAM provide more than six entries for every filter. On average, the filter sets required 2.25 entries per filter. While this is considerably less than the worst case of 900 entries per filter, yet it remains a large source of inefficiency. The magnitude of the *Expansion Factor* is not the only challenge. Note the high variance in the *Expansion Factor* among the filter sets; this presents a challenge in designing systems, as the filter storage capacity varies widely with filter set composition.

Table 6: Number of entries required to store filter set in a standard TCAM.

Set	*Size*	*TCAM Entries*	*Expansion 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**