DPF: A Dynamic High-Performance Network Packet Filter

by

Dominic Sartorio

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author .....
Department of Electrical Engineering and Computer Science
May 12, 1995

Certified by .....
M. Frans Kaashoek
Assistant Professor
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Theses

DPF: A Dynamic High-Performance Network Packet Filter

by

Dominic Sartorio

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1995, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Using *packet filters*, a network application can describe which incoming messages should be delivered to that application. Applications submit packet filters to a *packet demultiplexor*, which uses them to direct messages to the correct applications. Most current packet filters are written in low-level imperative languages, which are then interpreted in turn by the demultiplexor on each incoming packet. This framework can be very inefficient for two main reasons. First, the interpretation of low-level imperative code is inherently slow. Second, most demultiplexors today do not adequately recognize and exploit similarities between filters.

In this thesis, we describe the design and implementation of the *Dynamic Packet Filter* (DPF), a novel packet filtering system that significantly improves demultiplexing performance. DPF incorporates three ideas that contribute to fast demultiplexing. First, filters are written in a higher-level declarative language, allowing for more efficient recognition and exploitation of overlap between filters. Second, filters are merged such that no instruction is run more than once on any given packet. Third, instead of interpretation, dynamic code generation is performed on the merged filters, with the resulting machine code then run on incoming packets. As a result, DPF demultiplexes packets much faster than other existing demultiplexors. At the cost of a higher setup time, DPF performs at least six times to an order of magnitude faster than Pathfinder and the Mach packet filter, two high-performance packet filters.

Thesis Supervisor: M. Frans Kaashoek
Title: Assistant Professor

Acknowledgments

I would like to thank the following people, whose help in completing this thesis was invaluable:

To Frans, for the overall supervision of this project, and for all his help in shaping and writing this thesis. In particular, his careful reading of many drafts and his many helpful comments were invaluable. Finally, I thank him for challenging me to learn as much as I did.

To Dawson, who generated the core ideas behind DPF, including the use of a declarative language, the merging of filters regardless of where instructions appeared, and the dynamic code generation thereof. He also wrote the parser and provided the groundwork for DCG. Finally, I thank him for his continuous technical supervision. I could not have done this without him.

To Josh and Anthony, my officemates, for providing input and patiently answering my annoying questions.

To Willy, my roommate, for graciously accepting my complaining.

To my parents, for moral support.

To others, who remain unnamed (but you know who you are), for your timely support, advice, suggestions, and constructive criticism. Thank you all very much for your help.

Contents

Chapter 1

Introduction

This thesis discusses the design, implementation, and performance of the *Dynamic Packet Filter* (DPF), a novel network packet demultiplexor that provides an alternative over other user-level packet filters through superior demultiplexing performance. DPF accomplishes this by incorporating three main ideas. First, filters are written in a higher-level declarative language instead of an imperative language, allowing for more efficient recognition and exploitation of overlap between filters. Second, filters are merged such that no instruction is run more than once on any given packet. Third, instead of interpreting filters, dynamic code generation is performed on them, with the resulting machine code then run on incoming packets. At the cost of a higher setup time, DPF performs at least six times to an order of magnitude faster than Pathfinder and the Mach packet filter, two high-performance packet filters.

The remainder of this section will provide the background and motivation behind DPF and a brief overview of the remainder of this thesis. We define terminology and then state the problem with existing filtering systems. We then offer our solution, based on our observations regarding the packet filtering problem, and preview the most significant results. Lastly, an outline of the remaining chapters is provided.

1.1 Background

A multiprogrammed computer attached to a network needs a means by which it can decide which application will receive a network packet addressed to that computer. This is the role of the *packet demultiplexor.* Most often located in the kernel or some trusted server,

it demultiplexes an incoming stream of packets towards their intended applications. To do this, it requires specifications of the packets that the respective applications are interested in. These specifications are known as *packet filters* because their semantics are such that they filter a packet stream and allow only those packets that meet its specification to be directed towards its application. Alternatively, packet filters can be regarded as boolean functions performed on each packet by the demultiplexor, with "true" indicating that the given packet belongs to that filter's application (*accepted*), and "false" indicating otherwise (*rejected*). Packet filters can either be directly installed as part of the demultiplexor or provided during run-time by the applications themselves. This latter class of filters, known as *user-level filters*, is the topic of this thesis.

User-level filters are popular because they provide more flexibility and less maintenance headache than hand-coded implementations of network code. By having applications provide their filters to a protocol- and application-independent demultiplexor during run-time, one does not need to take the whole system off-line and manually update the demultiplexor whenever a new application or protocol needs to be supported [15]. In order to make demultiplexors as general-purpose as possible, filters are written in specialized safe languages and the demultiplexor itself consists of an interpreter that runs incoming packets through the instructions of each filter. These safe languages are usually stack-based [15] or register-based [13, 21] imperative languages, and the interpreter accepts and rejects packets by running these imperative instructions on them and returning the final boolean value. Each filter is run against a given packet in turn until one of them accepts it or all reject it.

This framework, however, has been too slow on some networks, especially faster modern networks such as ATM [4]. Although some recent work has improved the demultiplexing of user-level filtering systems to somewhat more tolerable levels of performance [4, 21], other recent work suggests that the user-level packet filter model may be inappropriate for fast networks because it is inherently too slow [7, 8]. This thesis proposes a packet filtering system that achieves the flexibility and maintainability of user-level packet filters, while significantly improving demultiplexing performance.

1.2 The Problems with Current Packet Filters

Three observations regarding the nature of packet filtering indicate where there is room for improvement in demultiplexing performance. First, many filters exhibit a good deal of overlap. They share many common instructions, mostly because they may look for packets of the same protocol. Others may differ by only a few constants. However, most demultiplexors today do not adequately take advantage of this overlap. The result is many redundant computations, contributing to poor demultiplexing performance. It should be possible to *merge* filters or otherwise recognize common computations. Merging filters results in fewer filters that must be run in turn through the interpreter, thus resulting in improved demultiplexing performance. Some more modern demultiplexors [4, 21] do this to some extent, but there is still much room for improvement. For example, modern demultiplexors merge filters if their first few instructions are identical, but do nothing to recognize identical instructions appearing after different instructions.

Second, the imperative languages in which most modern filters are written have semantics that lend themselves to inefficient interpretation [13, 15, 21]. In particular, they do not capture very well the notion of a packet filter being a boolean function on packets. We believe that imperative languages provide a too low-level semantics for the packet filter, and result in both inefficient interpretation and inefficient merging of filters.

Third, interpretation itself is slow. An interpreting demultiplexor with even just one active filter may not keep up with the maximum bandwidth on an ATM [4].

1.3 Our Solution

Based on these three observations of how to improve the performance of packet filters, we have designed and implemented DPF, the *Dynamic Packet Filter* system. DPF is based on three ideas. First, DPF performs more aggressive merging of filters that exhibit similarity. DPF recognizes common instructions regardless of location within a filter, and ensures that no instruction is executed more than once on a particular packet. This is a significant improvement over all other existing demultiplexors, in which the recognition of common instructions is location-dependent.

Second, DPF's filter language is a higher-level *declarative* language whose semantics more closely match the concept of a packet filter as a boolean function. A declarative

language allows filters to be easily written in a form that more closely matches the packet specifications they are intended to describe, without describing the imperative instructions that a machine should execute [4]. This allows a more efficient implementation of DPF's merging of common filters. Pathfinder [4], a recent packet filter system, also uses a declarative language, but does not exploit it fully for merging similar filters.

Third, DPF does not interpret filters. Instead, dynamic code generation is performed on filters upon their insertion and removal, and the resulting machine code is run on incoming packets. Since running machine code is inherently faster than interpretation, this results in much faster demultiplexing of packets. Dynamic code generation of packet filters has been suggested in previous work [9, 15, 20], but no filter systems have used it, to our knowledge.

We have implemented and tested DPF. Our implementation runs in user-space and tests were run by reading filters from UNIX files and applying them to sample packets of common networking protocols. Results were compared to the Mach packet filter (MPF) and Pathfinder, two recent high-performance packet filter systems. We have shown that DPF performs significantly better than both of these, demultiplexing at least an order-of-magnitude faster than MPF and at least six times faster than Pathfinder in most cases.

This improved demultiplexing performance comes at a cost — namely, the setup cost involved when filters are inserted and removed from the DPF system. Because DPF performs aggressive filter merging and dynamic code generation, which other filter systems do not perform, DPF requires more cycles during insertions and removals. However, we consider trading setup time for faster demultiplexing to be an acceptable tradeoff, mostly because we expect that setup operations happen infrequently on most systems.

1.4 Thesis Overview

The remainder of this thesis describes the background, design, implementation, testing and analysis of DPF, and is divided as follows. The next chapter describes the DPF language. Chapter 3 details DPF's filter-merging algorithms and dynamic code generation. Chapter 4 describes the test suite and results, and provides some analysis of those results. Chapter 5 analyzes DPF's results in some more detail by discussing how resolving each of our three observations contributed to DPF's performance. Chapter 6 discusses related work. Chapter 7 concludes this thesis by summarizing its main contributions.

Chapter 2

The DPF Language

2.1 Overview

The DPF language consists of a simple declarative instruction set. This instruction set is *declarative* because it is used to provide a specification of the values found in specific bitfields in an incoming packet in order for that packet to be accepted, without describing procedurally how a machine should go about demultiplexing that packet. A broad range of simple operations are supported, thus allowing DPF to handle a broad range of possible protocols. Fragmentation of packets is also supported.

The DPF language captures the notion of a packet filter being a conjunctive set of boolean operations on a packet's header. We call these boolean operations *atoms* and a conjunctive sequence thereof *predicates*. Predicates are intended to be complete specifications of packets, but sometimes this is difficult to do in a single conjunctive sequence. A complete *filter*, then, is a disjunctive sequence of one or more predicates. The whole filter is intended to return a boolean when checked against any given packet, indicating whether this packet is accepted or rejected by the filter.

The language also incorporates the semantics of fragment recognition and reassembly by allowing for two special kinds of predicates. First, one can write specifications of packets that should be postponed for later demultiplexing (in the form of *postpone predicates*). These predicates are intended for out-of-order fragments that arrive before the first one, which contains the header information required for demultiplexing. Second, one can write specifications of fragments arriving after the first one arrives (called *subsequent fragment predicates*). Most protocols supporting fragmentation include *message IDs* by which one

can easily identify fragments of a given packet. DPF provides a means for doing this by allowing one to write predicates that include a check for the message ID found in the first fragment.

The remainder of this chapter describes the DPF language in more detail. Section 2.2 describes the grammar, and section 2.3 expands on how the DPF language can be used to express fragment recognition and reassembly.

2.2 The DPF Language

The grammar of the DPF language is given in Figure 2-1. For readability and easy implementation of packet filters by the application writer, familiar C syntax is used wherever possible. The grammar and syntax are more fully explained in the following paragraphs.

An atom describes a boolean operation which the packet filter is intended to perform on a field of the packet. Its intended basic syntax, *(bitfield rel_op constant)*, captures these semantics. The grammar indicates and our implementation supports the possibility of more complicated formulations of atoms. Any number of functions on any number of bitfields and constants can appear on either side of the rel_op. In practice, we have found that only the formulation *(bitfield rel_op constant)* is really needed.

Any of the six basic relational operators and many different arithmetic operators can exist within an atom. While many of these operations will not be needed for most protocols, they are included for added programming flexibility. Furthermore, they are easy to support. In fact, our experience leads us to expect only two relational operators, "==" and "!=", to be in common use. Likewise, for arithmetic operations, we have found "+", "&", and the shift operators to be sufficient for most modern protocols.

Bitfields, values to be extracted from the specified bits in the packet, are represented by a factor and a constant seperated by a colon. The left factor must represent the offset in bytes of the base of the bitfield from the beginning of the packet. The right constant must represent the length of the bitfield in bits. Currently, only 4, 8, 16 and 32 are legal bitfield sizes.

The predicate is intended to represent a complete specification of the desired packet, consisting of a conjunctive set of boolean operations. Syntactically, it consists of a set of atoms joined by "&&" operators. Logically, the atoms are checked one at a time. If one is

```
filter:  '(' predicates ')' [ '{' predicates '}' ]?
       | ϵ

predicates:  predicate
       | predicates '||' predicate

predicate:  atom
       | predicate '&&' atom

atom:  factor rel_op factor
       | '(' factor rel_op factor ')'

factor:  term
       | [sign]?  factor
       | factor arith_op factor
       | '(' factor ')'

term:  constant
       | bitfield

bitfield:  factor ':'  constant

constant:  ['0'-'9']?  /* decimal */
       | '0x'['0'-'f'] /* hex */
       | 'bx'['0'-'1'] /* binary */
       | '0'['0'-'7'] /* octal */

sign:  '+' | '-'

rel_op:  '<' | '<=' | '>' | '>=' | '==' | '!='

arith_op:  '+' | '-' | '*' | '/' | '%' | '^' | '|'
       | '&' | '<<' | '>>'
```

Figure 2-1: Grammar for the DPF language.

```
(  /* Ethernet header  */
   ((2:16) == 0x0200) &&        /* Check if a PUP packet    */

   /* PUP header       */
   ((7:8) > 0) &&               /* Check if 0<PupType<=100  */
   ((7:8) <= 100) &&
   ((14:32) == 123456) &&       /* Check destination socket */
   ((19:8) == 12) &&            /* Check source host        */
   ((20:32) == 123456)          /* Check source socket      */
)
```

Figure 2-2: Simple DPF filter recognizing a PUP packet.

false, then the packet is immediately rejected by that predicate. All atoms must be true for the packet to be accepted by that predicate.

Figure 2-2 gives a simple example of a DPF packet filter. This filter accepts PUP packets [5] of a specified range of types arriving via an Ethernet from a specified socket and address, and being sent to a specified socket. The first atom verifies that the packet is a PUP packet by checking the appropriate bitfield in the Ethernet header. Whether it is within a given range of PupTypes is determined by the next two atoms, representing the two inequalities necessary to verify this condition. The next three atoms are simple equalities checking the source and destination of this packet.

Normally, an application's filter will consist of just one predicate, but occasionally an application may be interested in a set of packets that cannot be specified by one conjunctive set of booleans. Thus, filters consist of one or more predicates joined by "||" operators. If at least one predicate's atoms are all true, then the packet is accepted. If no predicate's atoms are all true, then the packet is rejected.

DPF supports indirect loads, in which the left argument to the ":" (bitfield) operator can itself be a function of other bitfields. This situation often arises in practice when one wishes to check a bitfield in a protocol header that appears after the variable-length header of a lower-level protocol. The Internet protocol [1] is a prime example. IP itself consists of a header which is variable in length due to the possible presence of some optional fields. Its header also includes a bitfield indicating the length (in 32-bit words) of the header. Thus, in order to check bitfields of higher-level protocols (such as TCP [2] and UDP [16]), one must find the header length in the IP header. DPF simplifies this process by allowing the packet

```
(
  /* Ethernet header  */
  (2:16 == 0x0800) &&           /* Check if IP datagram            */

  /* IP header  */
  ((10:16 & 0xbfff) == 0) &&    /* Check if unfragmented           */
  (13:8 == 6)  &&               /* protocol == 6 indicates TCP     */
  (16:32 == 0xc00c4501) &&      /* Check source IP address         */

  /* TCP header  */
  ( (((4:8 & 0xf) << 2) + 4) : 16 == 1234) && /* Source port       */
  ( (((4:8 & 0xf) << 2) + 6) : 16 == 4321)    /* Destination port  */
)
```

Figure 2-3: Sample DPF filter recognizing a TCP/IP packet.

filter writer to write the base offset of a bitfield as a function of these header-length bitfields. Figure 2-3 provides the case of TCP/IP as an example. The last four bits of the bitfield (4:8) contain the length of the IP header and must be used in defining bitfields in the TCP header, such as (in this example) the packet's source and destination port numbers. Here, we extract the relevant four bits by masking (4:8) with 0xf, left-shift the result twice to obtain the IP header length in bytes, and then add a constant to account for the Ethernet header's length and the location of the port numbers within the TCP header.

2.3 Fragment Recognition and Reassembly

The DPF language also supports packet fragment recognition and reassembly. Since most networks have a maximum message size, protocols (including the IP protocol [1]) break packets into multiple fragments. The task of reassembling these fragments usually falls on their recipient. As an additional complication, these fragments may appear out of order. DPF provides a facility for detecting fragments in any order, saving them away, and demultiplexing them once the first fragment arrives. This facility is modelled after the Pathfinder [4] packet classifier's fragment recognition paradigm. DPF's version has some weaknesses, which will be discussed in Chapter 5, but it sufficed for the purposes of this implementation of DPF.

Figure 2-4 illustrates this process, and the following paragraphs describe in detail how

the DPF language supports it.

The problem with out-of-order fragments is that most of the header information needed for demultiplexing appears only in the first fragment. It is usually possible to determine if an application *may* be interested in a packet's out-of-order fragment, even without having the packet's header yet. This is crucial if we hope to identify fragments for postponement. So, *postpone predicates* can be written that identify which fragments should be saved away for checking later, when the first fragment arrives. These predicates are marked with the "P" token. If one such predicate accepts a packet, that packet is presumed to be an out-of-order fragment and is saved by the DPF implementation for demultiplexing once the first fragment arrives. Care should be taken in writing the "P" predicate such that it does *not* catch the first fragment; one would like the first fragment to be accepted by other predicates in the filter instead of it being postponed itself.

The first fragment will indicate the unique message ID by which one can recognize the postponed and subsequent fragments. This message ID is found in a bitfield that is well-defined within that packet's protocol. The location of this bitfield should be defined in another predicate, the *subsequent fragment predicate.* This predicate appears in brackets at the end of the filter. In order to properly access the message ID, two constraints need to be set on this predicate. First, the bitfield in the last atom of this predicate's "&&" chain must define the location of the unique message ID. Second, this bitfield should have constant delimiters and not be part of an indirect load. Care should be taken in writing these subsequent fragment predicates because most of the fragmented packet's header information, including source and destination addresses and port/socket numbers, will appear only in the first fragment. Usually, it is only necessary to check for the correct protocol and the unique message ID.

When a packet filter is initially inserted, this bracketed subsequent-fragment predicate is saved away for later use. Only the unbracketed predicate(s), including the postpone predicate(s), are actually inserted into the DPF system. At least one of these predicates should be a postpone predicate, unless the filter writer is somehow certain that fragments will never arrive out of order. The bracketed subsequent-fragment predicate is not activated until the first fragment arrives.

Once the first fragment arrives, the DPF system lifts the value in the message-ID bitfield in the first packet and places it in the message-ID atom of the subsequent-fragment predicate

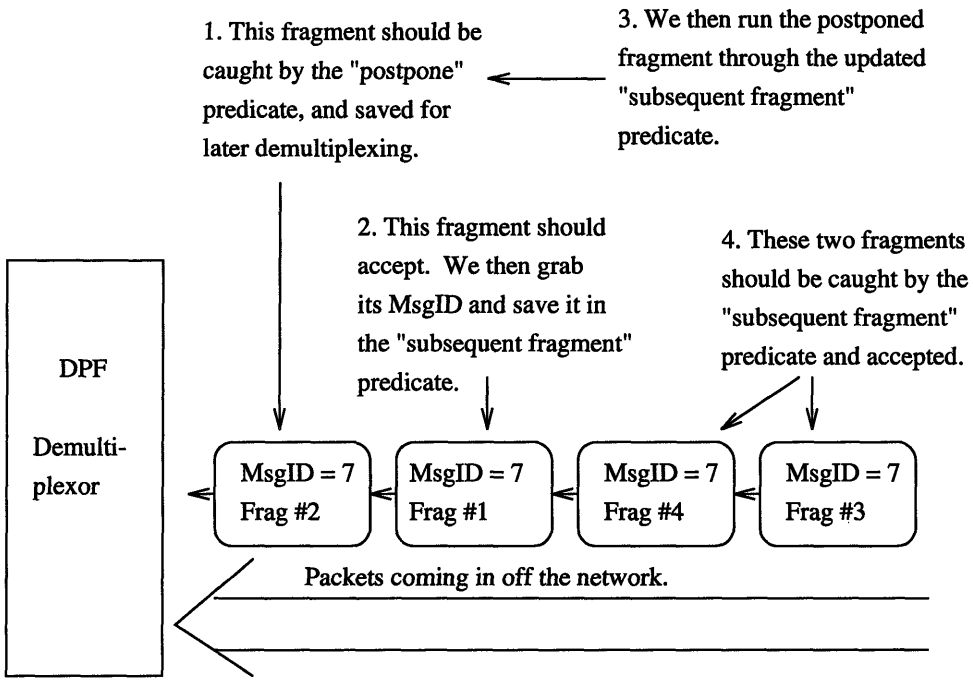


Figure 2-4: Illustration of fragment postponement and demultiplexing. Fragments arriving before the first fragment should be postponed. The first fragment should be accepted, and its message ID should be used to demultiplex postponed and subsequently arriving fragments.

as the constant to which we are comparing the message-ID bitfield, thus creating a boolean operation that checks subsequent fragments for that message ID. This predicate is then inserted, merged with the other filters, and dynamically compiled into machine code. It is then used to check all subsequently-arriving *and* previously arrived fragments that had been postponed by predicates from the same filter.

Postponed fragments are placed in a queue of fixed length. When the queue fills, the oldest fragment is discarded, and the next incoming fragment is placed at the end of the queue.

An example of a filter that checks for fragmented packets is given in Figure 2-5. The protocol for which this filter checks is the Reservation Protocol (RSVP) [22]. RSVP is a protocol intended for use in allocating network resources and setting up communication paths immediately before a multicast or unicast. RSVP packets are usually quite large and are often subject to fragmentation. In this filter, one predicate is intended to accept the first fragment, and contains information specifying a particular RSVP session. The

```
(  /* Predicate to accept without postponing  */

 (( /* Ethernet header  */
   ((2:16) == 0x0800) &&       /* Check if IP datagram           */

   /* IP header  */
   ((13:8) == 46)  &&          /* protocol == 46 indicates RSVP  */
   ((16:32) == 0x121a0009) &&  /* Check source IP address        */

   ((10:16 & 0x9fff) == 0) &&  /* Accept if frag count == 0      */

   /* RSVP header  */
   ( ((((4:8 & 0xf) << 2) + 4)  : 32) == 0x121a0040) &&
                               /* destination address            */
   ( ((((4:8 & 0xf) << 2) + 20) : 32) == 0x121a0041) &&
                               /* check previous hop address --  */
                               /* last machine to see this packet*/
)  ||

   /* Predicate to postpone  */

(P (((2:16) == 0x0800) &&      /* Make sure it's IP              */
    ((13:8) == 46)     &&      /* Make sure it's RSVP            */
    ((10:16 & 0x9fff) > 0)     /* Postpone if frag count > 0     */
)))

   /* Predicate to accept subsequent fragments -- inactive until */
   /* first fragment arrives.  */

{  ((2:16) == 0x0800) &&       /* Make sure it's IP              */
   ((13:8) == 46)     &&       /* Make sure it's RSVP            */
   ((8:16) == 0) }             /* Check msg ID given by 1st frag */
```

Figure 2-5: DPF filter recognizing a possibly fragmented RSVP packet. Fragments are postponed by checking the IP fragment field. Fragment reassembly is performed by recognizing all fragments that have the message ID given by the first fragment.

second predicate, ORed to the first, is the postpone predicate. Both of these predicates are activated upon the filter's insertion into the DPF system. The postpone predicate does not check for any information in the RSVP header. Since fragmentation should be done at the IP layer, the RSVP header will appear only in the first fragment. So, the postpone and subsequent fragment predicates should contain only atoms checking bitfields in the IP header. In this example, the postpone predicate need only check if the packet is of the RSVP/IP protocol and if the fragment count is greater than zero. This last atom will ensure that the postpone predicate does not catch the first fragment, whose fragment count will be equal to zero. Likewise, the subsequent fragment predicate will contain only atoms checking the IP header. The last atom contains the bitfield in which the unique message ID is located, thus meeting the constraints for a subsequent fragment predicate. This bitfield as written happens to be compared to zero, but any constant could be written here. It will only be overwritten by the actual message ID from the first fragment.

Chapter 3

DPF Implementation

3.1 Overview

This chapter describes the implementation of DPF's parsing, merging, and dynamic code generation (DCG) of filters. These are executed in sequential order in DPF's implementation, and will be described in the same order in this chapter. Figure 3-1 illustrates the process of parsing, merging, and generating code for DPF filters.

First, a DPF filter must be parsed into a tree-like structure before it is submitted to the demultiplexor. This parser's implementation is described in section 3.2.

Next, DPF attempts to merge the new filter with those that were previously inserted. This is done by (1) identifying the atoms and predicates in the new filter, (2) comparing these atoms to those of previously activated filters, (3) arranging the updated collective set of atoms into a queue, ordered in descending order of number of appearances, and (4) performing all the necessary bookkeeping to keep track of which predicates atoms belong to. We call this set of operations the *merging* of filters, because it is here where we find overlaps between all filters and try to take advantage of them by merging the similar instructions. These operations are described in section 3.3.

Immediately thereafter, code is generated for the whole updated collection of atoms. Atoms are dequeued and code is generated for them one at a time, in a specific order. This ordering is important, as it ensures that each atom is executed no more than once. The ordering is such that, when the code is subsequently executed, it follows a path down a *binary true-false tree* of atoms. This binary true-false tree has boolean operations (the atoms) as nodes, with each node having one branch representing the *true* condition and the

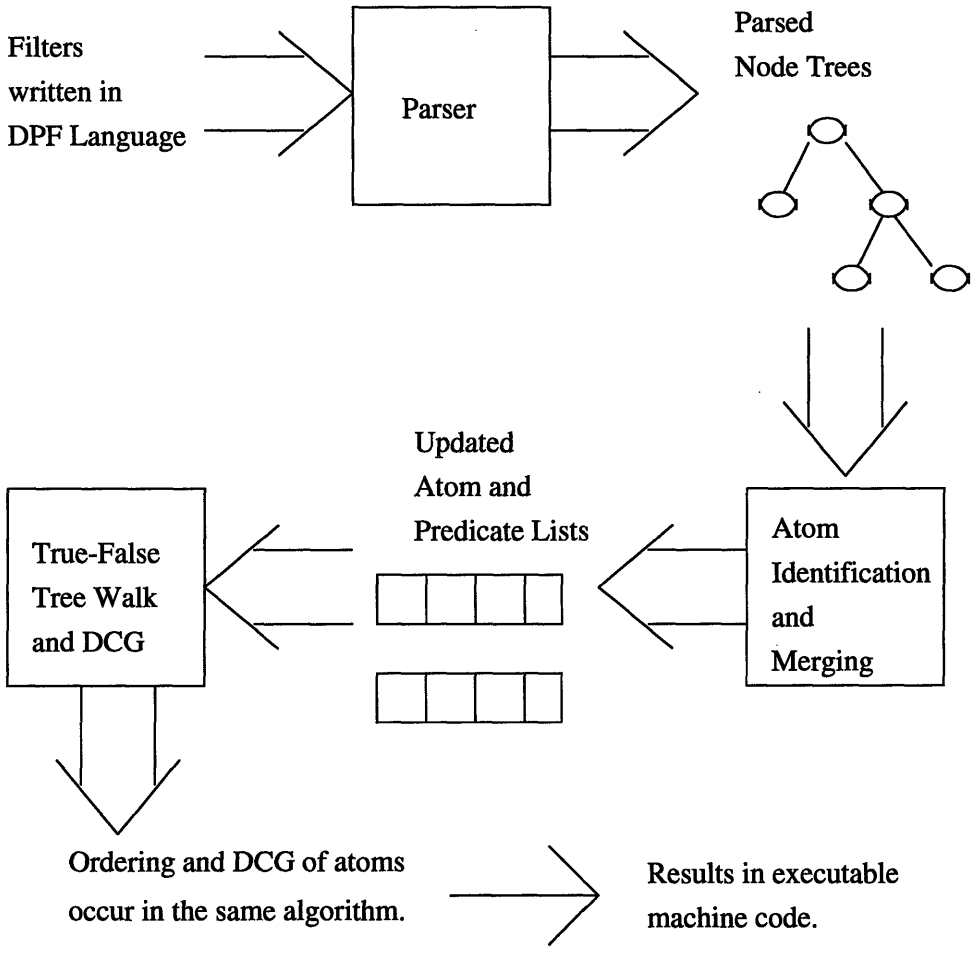


Figure 3-1: Illustration of the separate steps in DPF's implementation. Filters are parsed, then atoms are recognized and merged, and then dynamic code generation is performed on each atom in an order that traverses a true-false tree.

other branch representing the *false* condition. Atoms that appear in the most filters are at the top of this tree, with the least common atoms appearing as leaves. Execution begins at the top of this tree and continues until a leaf is reached, at which point it is known which filter (if any) should accept the packet. Traversal of this tree is described in section 3.4.

After laying the groundwork of how atoms are merged and how their code generation is ordered, we then discuss additional operations which were later added to these basic operations. The first is merging atoms that differ only by the constant to which packet bitfields are compared. These are called *disjunctive atoms*, because they are considered to evaluate to *true* if just one of the constants matches the packet and *false* if none do. Disjunctive atoms were added because they can vastly improve DPF's demultiplexing performance for a

majority of filters. The functions that build and manipulate disjunctive atoms are described in section 3.5

A second additional operation is handling filter priorities. Often, a packet can match several filters' specifications, but only one should accept it for security reasons. Most demultiplexors in the literature arbitrate with *priorities*, such that only the highest-priority filter actually accepts a packet. How this is done in DPF is described in section 3.6.

We then close with a discussion of per-atom dynamic code generation. The DCG functions are made as machine-independent as possible, with instruction emissions being performed by macros that are defined in machine-dependent header files. This was done to simplify porting DPF to different architectures. Some standard compiler optimizations that were applied to DCG are also described.

3.2 Parsing DPF

DPF filters must be first parsed into data structures that can be easily manipulated by the filter merging algorithms. In this section, we first discuss the parser's basic implementation, and then close with a discussion of how certain constraints on parsed DPF filters are enforced.

3.2.1 Parser Implementation

The parser module was built using the standard parsing tools `flex` and `bison`, with the grammar presented in Figure 2-1. DPF tokens are parsed into a `Node` structure, shown in Figure 3-2. It contains:

1. An integer denoting the operation type, if the token is an operator;
2. Two pointers to other node structures, representing the arguments to an operator token (none, one, or both may be initialized);
3. Two integers denoting the base and size of a packet bitfield, initialized only if this is a bitfield token;
4. An integer containing the constant, only if this is a constant token;
5. A set of bits used by the dynamic code generator for register allocation;

```
struct {
        int op;                         /* denotes operation type */
        Node *kids[2];                  /* optional argument tokens
                                kids[0] is left branch, kids[1] is right */

        unsigned        base,           /* base of bitfields */
                        size,           /* size of bitfield */
                        val,            /* constant value */
                        reg:5,          /* register number (codegen)*/
                        app_id;         /* which application's filter
                                           this node belongs to  */
} Node;
```

Figure 3-2: DPF node structure into which each DPF token is parsed.

6. An integer that uniquely identifies the application whose filter from which this token originates, supplied by the application, usually the process identifier (PID).

Collectively, these entries represent all the information needed to generate the code for a filter.

The parser will attempt to assemble all tokens' node structures into a single tree. Since a legal packet filter is really just one large boolean function, it should always be possible to do this. The only exceptions are packet filters that handle fragmentation; the bracketed predicates are assembled into a separate node tree, to be inserted and merged by DPF at a later time.

A legal DPF tree should consist of one long list of logical operator nodes ("&&" and "||"), whose arguments consist of relational operator nodes. The arguments to these rel_op nodes will consist of the functions on bitfields and the constants to which we are comparing them. A small sample filter and its parsed tree structure are given in Figure 3-3.

3.2.2 Parse Trees' Canonical Ordering

Immediately after a filter's parse tree is constructed, some constraints on the ordering of nodes within the parse tree are enforced. These constraints define the *canonical ordering* of nodes within a tree, and serve to make the task of identifying atoms and predicates easier. In particular, node trees should abide by the following:

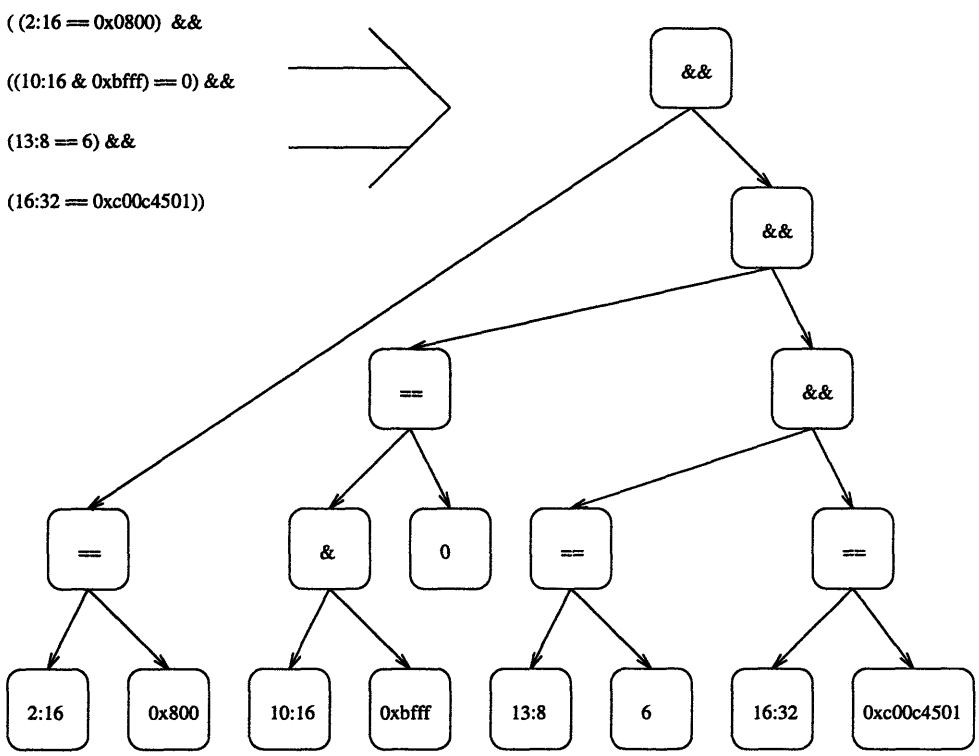


Figure 3-3: Sample TCP filter with parsed node structure.

1. Logical operators ("&&" and "||" tokens) have only relational operators and other logical operators as arguments, with the relational operators as the "left" argument and the logical operators as the "right" argument;

2. Bitfields (or functions thereof) always appear as the "left" argument and constants always appear as the "right" argument;

3. Constants are folded.

The sample parsed filter in Figure 3-3 abides by the canonical ordering.

This canonical ordering of argument nodes makes filter merging more efficient in two ways. First, atoms are easier to recognize in newly-submitted filters because the logical operations will always be expected along the right-most branch of a tree, with the relational operators always to the left. This way, atoms can be identified only by checking the left branch for rel_ops. Second, atoms are easier to compare with each other because constants will always be a rel_op's right branch and the bitfield will always be on the left. This cuts the time to compare atoms in half. Third, constant folding makes both atom identification

and dynamic code generation faster.

Parsed node trees are *canonicalized* by swapping "left" and "right" nodes as appropriate. In practice, most legal packet filters will already exhibit most of the requirements of this canonical ordering. We expect most filters to be written such that, for example, constants appear in the right branch and bitfields appear in the left branch of a rel_op node. However, we still enforce this ordering to prevent a poorly-written filter from hurting DPF's performance.

3.3 Merging Filters

This section describes the algorithm by which DPF finds and merges instructions found in multiple filters. The goal of DPF's filter merging algorithms is to ensure that all the common atoms in mergeable filters are executed at most once when demultiplexing a packet.

This algorithm is run whenever a new filter is inserted or an inactive filter is removed. An insertion or removal means that the collective set of active atoms has changed, and we should determine again how often and in which filters atoms appear. Immediately after the common atoms are identified and merged, dynamic code generation is run.

Subsection 3.3.1 describes the operations that occur whenever a filter is inserted. We will walk through an example to help illustrate these complex set of operations. Then, subsection 3.3.2 describes what happens when a filter is removed.

3.3.1 Inserting New Filters

Once the filter is parsed and canonicalized, its atoms are identified and compared with the atoms of previously-inserted filters. The predicates to which atoms belong are also identified. A relational operator is always the top node of an atom, and a list of atoms connected by "&&" operators constitutes a predicate.

After atoms are identified and compared, they are inserted into a global *atom queue*, ordered such that the most common atoms appear first and the least common atoms appear last. This queue is later used during the true-false tree walk to determine the order in which atoms are subject to dynamic code generation.

In the following subsection, the identification of atoms and predicates in a new filter is discussed in more detail. The next subsection then describes the data structures involved

```
struct {
  char active;          /* used by true-false tree walk           */
  short treedepth;      /* also used by true-false tree walk      */
  long pid;             /* app id associated with this pred       */
  Atom* atoms[MAXATOMS]; /* list of constituent atoms             */
  char postpone;        /* indicate if this is a postpone pred    */
  long natom;           /* number of dependent atoms (permanent)  */
  long refcnt;          /* number of dependent atoms remaining for
                           DCG (used by true-false tree walk) */
} Predicate;
```

Figure 3-4: DPF Predicate structure

in storing these new atoms and predicates in the DPF system, and how these new atoms are compared with previously-inserted atoms and inserted into the atom queue.

Identifying New Atoms and Predicates

First, we iterate down the node tree, identifying atoms and assembling them into predicates. At the beginning of this process, a `Predicate` structure is initialized. Space is allocated to hold pointers to the constituent atoms, along with some run-time information. This run-time information consists of:

1. The ID of the application that submitted this predicate's filter.
2. Two copies of the number of atoms in this predicate (length of the atom pointer array). One is a static count while the other is modified in the true-false tree walk.
3. A `postpone` boolean indicating if matching packets should be postponed.
4. An `active` boolean used in the true-false tree walk.
5. A `treedepth` integer, also used in the true-false tree walk.

See Figure 3-4 for the definition of the predicate structure.

The identification of atoms and predicates works as follows. When a "&&" operator is encountered, we look down its left branch for an atom, then continue down the right branch. When a "||" operator is encountered, then we know that another predicate must exist; another predicate structure is allocated, and the one currently being assembled is copied into it. We then continue assembling the first predicate by iterating down the left

```
struct {
  long npred;                 /* number of dependent preds (permanent)*/
  long refcnt;                /* number of dependent preds (used by
                                        true-false tree walk) */
  Predicate* dp[MAXPREDS];    /* list of dependent preds  */
  Node* relop;                /* the relop that this atom represents  */
} Atom;
```

Figure 3-5: DPF Atom structure

branch, and we assemble the second predicate by iterating down the right branch. If a "P" operator is found, then we know that the current predicate is a "postpone" predicate, and the `postpone` boolean is set to "true".

Atoms are identified when rel_ops are encountered. An `Atom` structure is initialized that contains the relational operator's subtree in addition to some other run-time information. Figure 3-5 shows the atom's structure. The run-time information consists of:

1. An array of pointers to the predicates that include this atom (called *dependent predicates*).

2. Two copies of the length of the predicate pointer array (the number of predicates containing this atom). Again, one is a static counter, while the other is modified during the true-false tree walk.

An example of the results of atom and predicate identification is shown in Figure 3-6. Here, the same filter as in Figure 3-3 is shown with the properly identified atoms and predicate. Some of the atom and predicate structure entries are also shown. In particular, the dependent predicate count is shown set to one and all pointers (rel_op nodes, atoms, and the predicate) are shown. The other entries (not included in the figure) will be filled in later on in the filter merging operations.

The Atom Queue and Predicate List

The last two data structures to be described in this section are the two global lists that keep track of all currently active atom and predicate structures. These two lists always represent the complete and current state of the DPF system. They collectively contain all the information needed to generate the machine code.

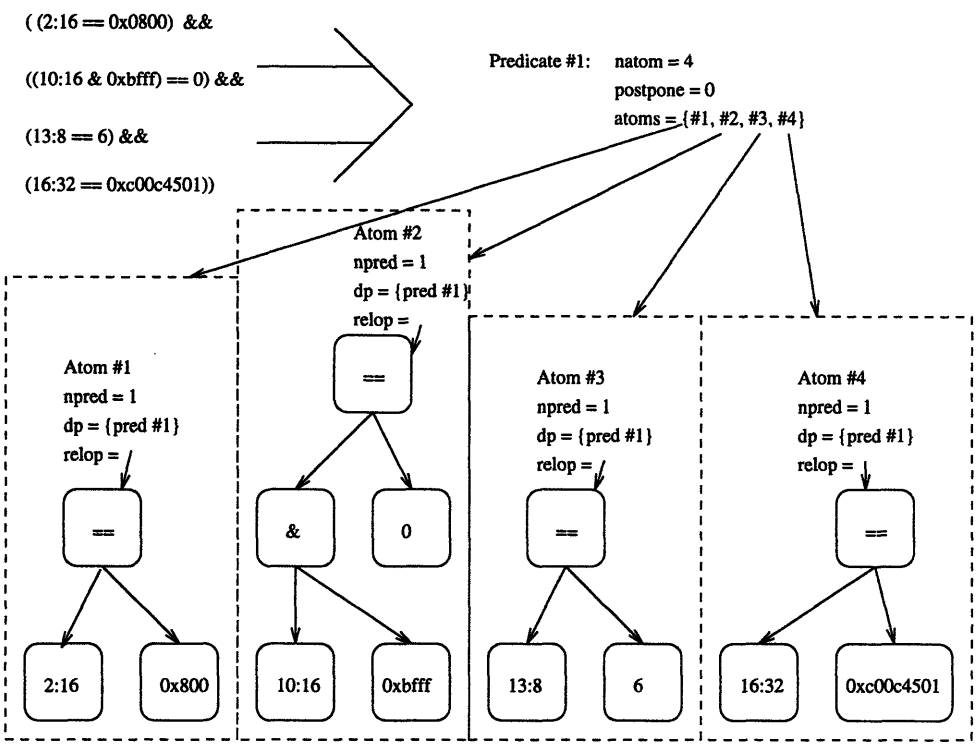


Figure 3-6: Sample TCP filter with atoms and predicate identified.

The predicate list is unordered, and acts only as a global repository of active predicates.

The atom list is treated as an ordered queue. Atoms in the atom queue are ordered in decreasing order of number of appearances in all active filters. So, the most commonly-occurring atoms are at the beginning of the queue, while atoms appearing only once are at the end.

Once an atom structure has been built and inserted into its predicate structure, it is compared with each atom that already exists in the atom queue. This is done by comparing the rel_op operators and their arguments. If the new atom does not match any existing atom, it is simply placed at the end of this queue. If a match is found, then the following occurs:

1. The existing atom's dependent predicate count is incremented.

2. The new atom's dependent predicate pointer is added to the existing atom's dependent predicate list.

3. The new atom structure is discarded.

4. The updated atom is swapped with the atom(s) ahead of it in the queue, if necessary,

until the queue invariant is again observed.

Once a node tree has been completely traversed and all atoms have been identified and inserted into the atom queue, the new predicate(s) are added to the predicate list, thus concluding the insertion operation. Execution then continues with the true-false tree walk.

Figure 3-7 shows an example of the effect of adding more filters to the system, illustrating what happens when filters with common atoms are inserted. In this example, all four filters share an atom comparing `2:16` to the same value. This atom appears at the front of the queue. It is followed by two atoms which each appear in two filters. The tail of the queue is occupied by an atom appearing in only one of the filters. All atoms contain their up-to-date multiplicity and list of pointers to dependent predicates. The predicate list is not intended to be in any particular order. But, the predicate structures in this list must contain the number of dependent atoms and pointers to those atoms. They must also contain their `postpone` status and the `pid` of the process that submitted them. Dummy `pids` are shown in this figure.

3.3.2 Removing Inactive Filters

Filters are removed when its application terminates. The application ID is simply passed to DPF, which iterates through both global lists and removes the relevant information.

First, we iterate down the global atom queue. For each atom, we iterate down its dependent predicate list. If a predicate's PID matches the given application ID, then that predicate's pointer is zeroed, the atom's dependent predicate counter is decremented, and atoms are rearranged in the queue as necessary to maintain the atom queue invariant. If the dependent predicate counter is decremented to zero, then this atom no longer appears in any active filters, and its pointer is freed.

Second, we iterate down the global predicate list. Whenever a predicate's PID matches the given application ID, its pointer is freed. If a filter consists of more than one predicate, all associated predicates are deleted, since they all share the common application ID.

After removing the inactive predicates and updating or removing their dependent atoms, dynamic code generation is performed on the resulting set of atoms.

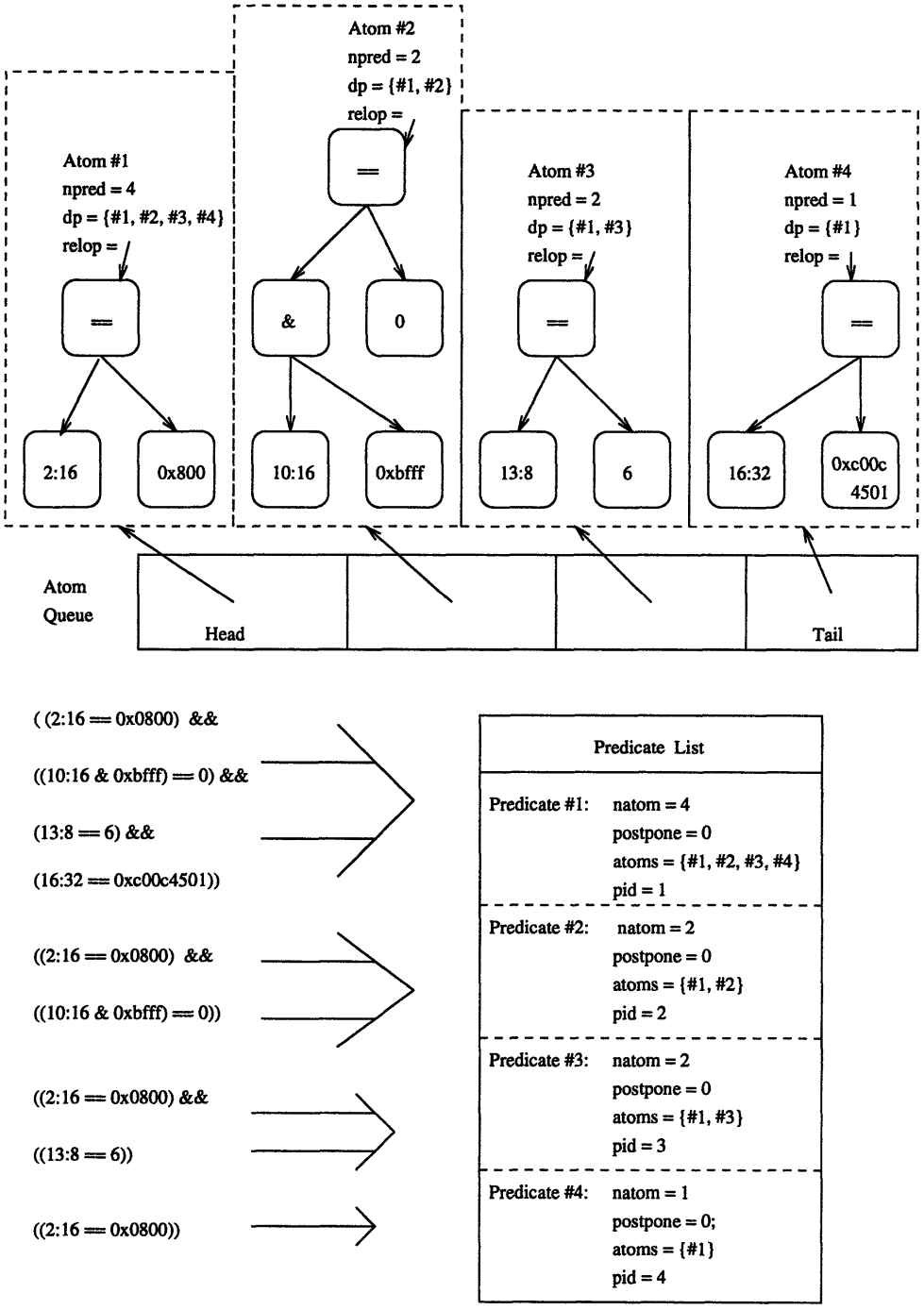


Figure 3-7: Four TCP/IP filters with the resulting atom queue and predicate list.

3.4 The True-False Tree

This section describes the traversal of the true-false tree of atoms. The true-false tree is a binary tree with boolean operations (the atoms) as nodes, "true" conditions as the left branches and "false" conditions as right branches. Execution of the machine code is intended to begin at the top of this tree and follow a single path to one of the leaves. The purpose of this is to ensure the following properties in the machine code:

1. The path of execution down the tree should not execute any atom more than once.

2. Atoms appearing in the most predicates are executed first, and those appearing least often are executed last. This keeps the tree small and ensures that the most rejected packets are rejected early.

3. Execution does not end until a leaf is reached. Here, and only here, can a judgment be made regarding all active predicates with respect to that packet. Also, only here can it be conclusively determined which predicate, if any, accepts the packet.

To help illustrate what a true-false tree for a particular set of filters looks like, Figure 3-8 shows the true-false tree for the filters given in Figure 3-7.

We detail the traversal of the true-false tree itself in the following subsections. The per-atom code generation that occurs during this traversal is described in section 3.7.

Brief Overview

In brief, there are three mutually recursive functions that perform the true-false tree traversal: `genatom`, `true_emit` and `false_emit`. `genatom` first dequeues an atom from the atom queue. If the atom exists in predicates that can still accept from the current location in the true-false tree, it emits the code for that atom, and then calls `true_emit` and `false_emit` to emit the "true" and "false" branches of that atom, respectively. If the atom exists in no predicate that can still accept, `genatom` will call just itself on the next atom in the queue. `true_emit` calls `genatom` again, after possibly emitting code that accepts the packet. Acceptance code is emitted if the last atom in a predicate had just been emitted. `false_emit` also calls `genatom` again. Both `true_emit` and `false_emit` perform bookkeeping on the atoms and predicates that keeps track of which predicates can accept at different points

in the tree. The operation of these functions is described in more detail in the following paragraphs. To help illustrate this algorithm, pseudocode is presented in Figure 3-9.

Setup

First, before calling **genatom**, all the atoms and predicates must be prepared for the tree walk. Each predicate's atom count is copied into its `refcnt` entry, as is each atom's predicate count. Also, each predicate's `active` boolean is set to "true". Being "active" means that, in a given location in the true-false tree, that predicate can still possibly accept a packet, but all of its atoms have not been executed yet. At this point, before any atoms are executed (the top of the tree), all predicates are active.

genatom

Then, **genatom** is called. **genatom** first dequeues the first atom in the atom queue. It then determines if this atom should be a node at this point in the true-false tree. It should be a node only if it is part of a predicate that is still active. **genatom** determines this by iterating down this atom's dependent predicate list, looking for predicates with a true `active` boolean. As soon as one is found, it knows that a predicate exists that is still a candidate for accepting the packet at this point in the true-false tree, but would not have done so yet, and so this atom should be a node at this point in the true-false tree. **genatom** then generates code for the atom and then executes `true_emit` and `false_emit`, and then ends by backpatching and re-enqueueing the atom.

If none of this atom's predicates are still active at this point in the true-false tree, then it should not be emitted. However, the next atom may need to be emitted. So, in this case, **genatom** simply calls itself, thereby dequeuing the next atom in the queue and possibly setting it up as a node in the tree. After returning from this call, the first atom is re-enqueued.

Each atom's block of machine code will end in a branch instruction. **genatom** emits a branch instruction such that the branch is taken if the relational operator is false. Execution will simply continue down the "true" branch if the atom is true. However, the offset is left out initially, in anticipation of `true_emit` emitting the "true" branch in the future. Later, when **genatom** knows the address that begins the false branch, backpatching will occur.

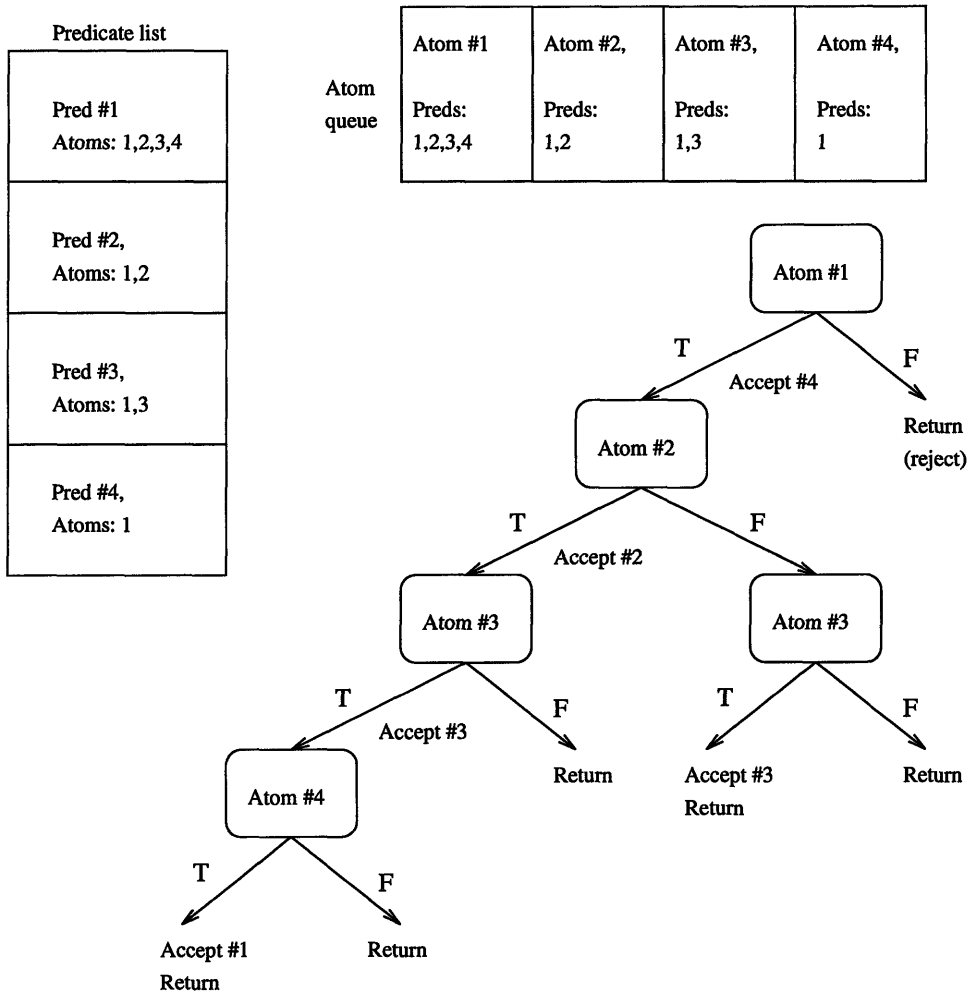


Figure 3-8: Sample true-false tree given the set of sample atoms and predicates in Figure 3-7. "(reject)" indicates a return with no acceptances having occurred, so the returned PID is zero. Otherwise, the PID returned is the PID of the most recently accepted predicate. Subsequent acceptances overwrite previous acceptances. Execution of machine code will always follow a sequence of arrows to one of the "returns" at the bottom of the tree.

```
tree-walk; arguments: none; returns: pointer
   foreach a:atom in atom queue do
      foreach p:pred in a.plist do
         p.refcnt := p.natom
         p.active := TRUE
         enqueue(genatom())        ;; Do tree walk
         backpatch_leaves()        ;; Leaves jump to return instruction
         return(code_handle)       ;; Return pointer to machine code

genatom; arguments: none; returns: atom
   execflag:int := 0
   a:atom := dequeue()        ;; Get next atom in queue
   if (a == NULL) then return(NULL)
   foreach p:pred in a.plist do        ;; Look for predicates that can still accept
      if (p.active == TRUE) then
         branchptr:instr := atom_emit(a)        ;; Emit atom's code and return its branch instruction
         enqueue(true_emit(a))        ;; Emit "true" branch
         patch(branchptr)        ;; Backpatch
         enqueue(false_emit(a))        ;; Emit "false" branch
         execflag := 1
         break
   if (not execflag) then enqueue(genatom())        ;; If this atom not emitted, try next atom
   return(a)

true_emit; arguments: a:atom; returns: atom
   foreach p:pred in a.plist do        ;; Look for predicates that accept at this point in tree
      if (p.active and -p.refcnt == 0) then
         accept_emit(p.pid,p.postpone)
         foreach q:pred in pred_masterlist do        ;; Deactivate predicates of same PID or lower priority
            if (q.active and (q.pid == p.pid or q.priority ≥ p.priority)) then
               q.active := FALSE
               q.depth := cur_depth
         ta:atom := genatom()        ;; Recursion: continue walking tree
         foreach p:pred in a.plist do        ;; Reactivate predicates
            if (cur_depth() == p.depth) then p.active := TRUE
            if (p.active) then p.refcnt++
         foreach q:pred in pred_masterlist do
            if (cur_depth() == q.depth) then q.active := TRUE
   return(ta)

false_emit; arguments: a:atom; returns: atom
   foreach p:preds in a.plist do        ;; Deactivate all dependent predicates
      if (p.active) then p.depth := cur_depth()
      p.active := FALSE
   fa:atom := genatom()        ;; Recursion: continue walking tree
   foreach p:preds in a.plist do        ;; Reactivate preds
      if (cur_depth() == p.depth) then p.active := TRUE
   return(fa)
```

Figure 3-9: Basic true-false tree-walking algorithm.

`true_emit`

After emitting code for an atom, `genatom` calls `true_emit`, which generates the true branch of the just-emitted atom. `true_emit` first iterates down the just-emitted atom's dependent predicate list, decrementing each predicate's `refcnt`. `refcnt` keeps track of the number of atoms in that predicate that have not been emitted yet, so this number must be decremented in *each* predicate that depends on the just-emitted atom. If a predicate's `refcnt` reaches zero, then all the atoms of that predicate have been emitted, and that predicate should accept. An instruction that places that predicate's PID into the return register is emitted. We also emit an instruction that places the `postpone` boolean in a predetermined address. However, the return instruction itself is not yet emitted. Actually, execution is intended to continue. Subsequent predicates may later accept, and their PIDs will overwrite this one. This may seem strange, but there is a good reason for doing this. Only one application should accept a packet. Even if several predicates can accept, there should be a way of arbitrating between them. Our way of arbitrating is by giving the packet to the application with the predicate having the most atoms — so, predicate acceptances at the bottom of the true-false tree overwrite earlier acceptances. This is an admittedly quick and dirty way of arbitrating between multiple accepting filters, but it also serves to cheaply enable a more intelligent way (prioritizing predicates) that will be described in section 3.6.

After emitting the code that accepts a predicate, `true_emit` sets the just-accepted predicate's `active` boolean to "false", and sets its `depth` to the current level in the true-false tree. This records where in the tree this predicate was deactivated. At the end of `true_emit`, this predicate is reactivated, but we must be sure that *only* this call of `true_emit` is the one that reactivates this predicate. (A global treedepth counter is incremented at the beginning of `genatom` and decremented at its end. When determining if a predicate should be reactivated, its `depth` is compared to this global counter. If they are equal, the predicate is reactivated.) Then, `true_emit` calls `genatom`. `true_emit` then finishes by reactivating all predicates that it deactivated.

`false_emit`

Next, the original call to `genatom` calls `false_emit`. `false_emit` simply deactivates all predicates depending on the just-emitted atom. Since the packet did not match that atom,

none of these predicates will accept. `false_emit` then calls `genatom`, and then reactivates all these predicates. The `treedepth` must again be saved away in each predicate, as a way of ensuring that only this call of `false_emit` actually reactivates these predicates, just as in `true_emit`.

Ending the Recursion

The bottom of these functions' mutual recursion is reached when `genatom` tries to dequeue an atom but finds an empty queue. This means that a leaf in the tree has been reached. In this case, a branch to a return instruction at the end of the machine code is set up, and will be backpatched at the conclusion of the true-false tree traversal.

When the top-level `genatom` call returns, the entire true-false tree has been emitted, and the atom queue has been reconstructed. At this point, the return instruction is emitted, and the unresolved branches at the leaves of the true-false tree are backpatched. Then, the DPF system returns the machine code handle as a pointer to a function, which can be subsequently called to check incoming packets. The prototype of this function pointer will be discussed in section 3.7.

3.5 Disjunctive atoms

Often, atoms will differ only by the constant to which a bitfield is being compared. The bitfield (or the function of that bitfield) and the relational operator are the same. This should happen very often in practice, because there will often be several filters that check for the same protocol stack, but differ by only an address or a port number. For example, in TCP/IP and UDP/IP, filters' IP portions are identical, but the port numbers in the TCP or UDP portions differ. Other filter systems described in the literature have been designed with this idea in mind [4, 21]. Combining these atoms that look up the same bitfield but compare against different values can significantly reduce the number of atoms that need to be manipulated by the DPF system, thus enhancing its demultiplexing performance. The resulting merged atom is called a *disjunctive atom*, because for it to evaluate to "true", the packet bitfield needs to match only one of the values. In fact, at most one value can be true.

This section describes how disjunctive atoms are implemented, and how the algorithms

```
struct {
  long npred;
  long refcnt;
  Predicate* dp[MAXPREDS];
  Node *relop;
  short disjunction;       /* how many disjunctive vals in this atom*/
  unsigned *vals;          /* list of disjunctive values            */
  Predicate*** disj_preds; /* list of dep. pred. lists              */
  long* disj_npreds;       /* list of dep. pred. list lengths       */
} Atom;
```

Figure 3-10: DPF Atom structure including entries for disjunctive atoms

described in the previous section are augmented to support this implementation. First, insertions and removals of filters are discussed. Then, we discuss some constraints that are placed on disjunctive atoms. Finally, the necessary enhancements to the true-false tree walk are discussed.

3.5.1 Filter Insertions and Removals with Disjunctive Atoms

First, some more entries were added to the "atom" structure:

1. A list of values, instead of just one value;
2. A list of dependent predicate lists, one corresponding to each constant;
3. A list of predicate list lengths to keep track of how long each of the new predicate lists are;
4. A `disjunction` integer field that indicates how many constants are in the disjunctive atom.

The updated atom structure is shown in Figure 3-10.

Upon insertion, when a new predicate's atoms are compared to existing atoms, a small check is also done to see if they are identical except for the constant. If they are not, then we proceed as before; either they are perfectly identical or quite different. But if they are, then one of two things can happen. First, the existing atom may not already be a disjunction, in which case one must be created. Memory is allocated for the additional entries in the atom structure (values, their associated dependent predicate lists, and the lengths thereof),

and these lists are added to one of the atoms. The two constants are placed in the `vals` list, and similarly for the two atoms' dependent predicate lists and the lengths thereof. The `disjunction` count is set to 2. The original `dp` list is now unused, but `npred` is still used as the atom's multiplicity counter. It should equal the sum of the entries in the `disj_npred` list. Also, the original value in the rel_op node (in the rel_op's right branch) is no longer used. DCG will compare the bitfield defined in the left branch only to those values in the `vals` list.

With the new predicate's atom's information now all encapsulated in the new disjunctive atom, it is discarded. Furthermore, the single constant and dependent predicate list are now unused; all the information is held in the new entries to the atom structure.

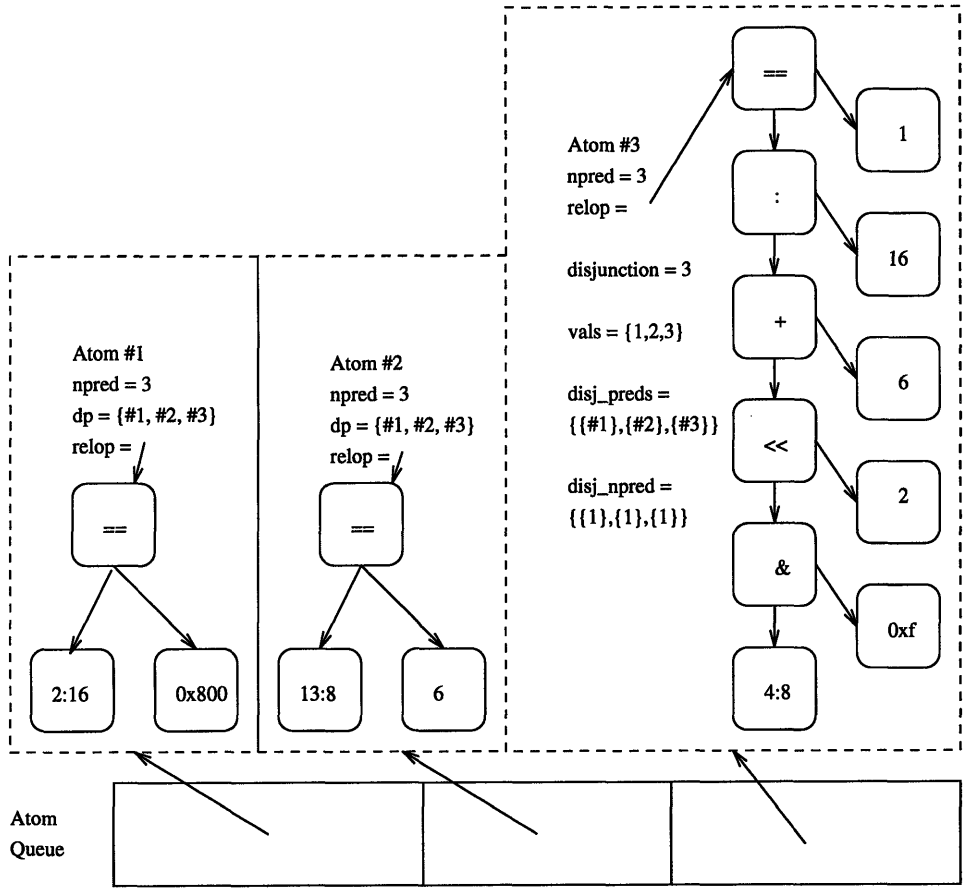
The second possibility is that the existing atom already is a disjunction. Then, the new atom's constant, dependent predicate list and the length of this predicate list are added to the corresponding lists in the disjunctive atom. The new atom is then discarded, since all its information has been encapsulated in the disjunctive atom.

Figure 3-11 illustrates the result of the above process. Three simple TCP filters are identical except for the port numbers in the last atom. These are merged into a disjunctive atom which is added to the end of the queue. All the relevant disjunctive atom structure entries are shown.

Upon a predicate's removal, and if one of its atoms had been merged into a disjunctive atom, the above process is reversed. If other constants still exist, just that constant is removed, its associated dependent predicate list is freed, and the corresponding length is deleted. If only one constant remains, then the atom must be returned to the non-disjunctive form. The remaining constant is copied back into the relational operator node (which is easily done because of its canonical form), the dependent predicates list is moved to the non-disjunctive dependent predicate list entry (`dp`), the length thereof is copied into the `npreds` entry, and `disjunction` is reset to zero.

3.5.2 Constraints Regarding Disjunctive Atoms

For disjunctive atoms to provide the most gain in demultiplexing performance, some constraints need to be enforced. First, disjunctive atoms should be found only at the end of



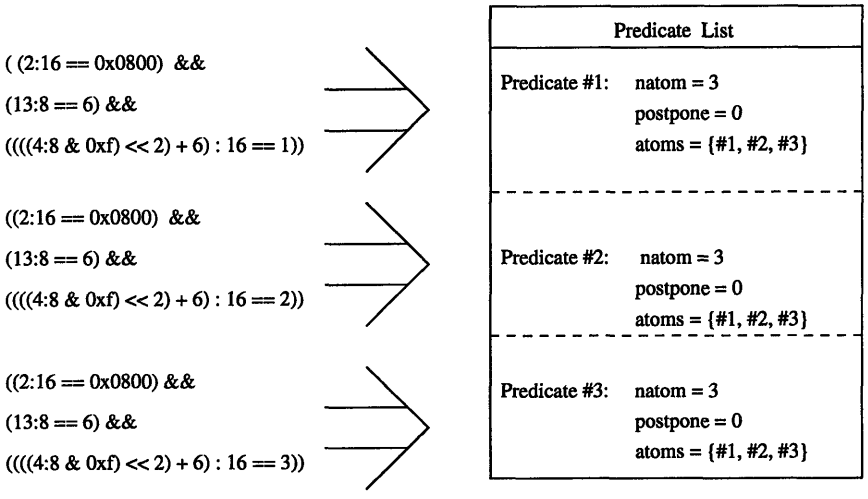


Figure 3-11: Example of a disjunctive atom. Three TCP filters differ only by the destination port number.

predicate. After emitting code for the portion of the tree below this node, these predicates (along with the predicate that just accepted) are reactivated. This way, a lower-priority predicate that would have accepted father down the tree will now not accept. Similarly, the currently accepting predicate would not be accepting if a higher-priority predicate had already accepted farther up the tree. Conversely, if a *lower*-priority predicate already accepted higher in the tree, then its PID and postpone-boolean would be overwritten by the current acceptance. Similarly, a *higher*-priority predicate accepting lower in the tree will overwrite this current acceptance.

3.7 Dynamic Code Generation

This section describes dynamic code generation (DCG). We begin with a discussion of the basic DCG algorithm. This discussion is followed by a description of standard compiler optimizations that were incorporated into DCG. We close with some implementation-specific details, including how the just-emitted demultiplexing function should be called.

3.7.1 The Basic DCG Algorithm

Machine code is generated for each atom as follows: The atom's relational operator node is passed to a function named **gen**. **gen** is a recursive function, calling itself on the zero, one, or two argument nodes in the node it receives. The basic functionality of **gen** is as follows, dispatched according to operator type:

1. No arguments indicate a constant or packet bitfield load into an available register.

2. For one argument, **gen** is first called on the argument node. Then, the argument node's register is deallocated and a register for the current node is allocated. Then, the unary instruction for the current node (not, neg, etc.) is emitted.

3. For two arguments, **gen** is first called on both argument nodes. Then, both argument nodes' registers are deallocated, and a register is allocated for the current node. Then, the binary instruction for the current node (add, and, conditional branches, etc.) is emitted. Branch offsets are not calculated here; this backpatching will occur at the end of the true-false tree walk.

This dynamic code generation implementation is based upon Engler's and Proebsting's "DCG" system [9]. The register allocation method is taken from Sethi-Ullman [3].

3.7.2 DCG Optimizations

Two optimizations have been implemented and added to the simple algorithm described above. They are the use of immediate-mode instructions and the optimization of bounds-checking on packet bitfield loads.

First, since many architectures support immediate/literal instruction modes, an instruction can be saved by taking advantage of them. When a unary or binary operator is recognized, the argument(s) can be checked for constants. If a constant exists, then the current node's instruction is emitted in immediate mode instead of calling **gen** again just to load the constant into a register.

The second optimization is bounds-checking only those bitfield load addresses that exceed all load addresses higher in the true-false tree. If a load for a larger offset has already occurred higher in the true-false tree, then there is no need to check it; if that load was legal, then so is this one. We also omit all bounds checks for packet load offsets that are less than a constant value; this value is defined in a header file and is currently set to the minimum length of a legal Ethernet packet. Keeping track of previous bounds-checked load addresses is done by keeping a set of global offset counters active during the true-false tree walk, one for each tree level, and updating them for each load as the true-false tree is traversed.

3.7.3 DCG Implementation Details

Finally, we discuss some important details regarding our DCG implementation. These are:

1. The use of machine-independent instruction macros;
2. Emitting code for hash tables;
3. The prototype of the returned function pointer.

Our implementation of **gen** does not use machine-dependent instructions. Instead, when an instruction is emitted, a machine-independent macro is called that is defined in a `binary.h` header file for that architecture. This allows for simpler ports of the DPF system to different architectures. However, some recoding is still necessary. In particular,

the atom queue, regardless of their multiplicity. Second, predicates can have at most one of their atoms merged into a disjunction. The following two paragraphs describe the reasoning and implementation of these constraints in more detail.

Once created or updated, a disjunctive atom is always placed at the *end* of the atom queue, regardless of its multiplicity or how many constants it holds. Although this breaks our atom queue invariant, there is a very good reason for this. This is necessary for keeping the size of the true-false tree small. If disjunctive atoms were near the top of the tree, then the true-false tree would no longer be binary. Instead there would be a branch for each constant, plus the "false" branch for when none of the constants were true. This would increase the tree size and worsen the performance of dynamic code generation, especially if the atom is very near to the the top of the tree. So, this atom is constrained to be at the bottom of the tree. Then, it will be the last atom to be checked before DPF returns. The packet is then either accepted with the matching constant's dependent predicate's PID being returned from the machine code, or we simply return without acceptance.

The second constraint is that only one disjunctive atom is allowed to exist in any one predicate. More than one can occur in all the active filters combined, but only one per predicate is allowed. This way, we ensure that a disjunctive atom is indeed always at the bottom of the tree, and is not followed by another disjunctive atom. This is enforced upon insertion when a new predicate's atoms are compared with existing atoms in the atom queue; only the first atom eligible for disjunction is actually merged into a disjunctive atom, and subsequent ones are treated as separate atoms. Although demultiplexing performance could improve by allowing multiple disjunctive atoms, they were disallowed to enable a much simpler implementation.

3.5.3 Disjunctive Atoms in the True-False Tree

Finally, we consider what must be done in the true-false tree walk to support fast demultiplexing with disjunctive atoms. When a disjunctive atom is dequeued, its values are stored in a quick-access hash table. An entry in this hash table is a structure containing the constant itself (for sanity checking), the predicate's PID, and a boolean indicating whether or not this constant is part of a "postpone" predicate. Thus, when executed, the machine code need only hash the value in the bitfield. If an entry exists, then all the information needed for acceptance is readily available. This hash table is implemented directly in the

machine code, including the hash function, collision checks, and traversing down the collision list. When a matching constant is found, the matching PID is saved in the return register. Otherwise, we simply jump to the return instruction without acceptance.

The hash function is the constant modulo twice the number of constants in the disjunction. We chose to modulo by twice the number of constants instead of just the number of constants (or any other smaller number) to make collisions less likely. Collisions will generate a list of entries down which one must traverse, and this can become very expensive if the list is long. So, we decided that reducing the number of collisions is worth the added memory overhead.

3.6 Handling filter priorities

This section describes priority-arbitrated demultiplexing in the DPF system. First, the reasoning behind using priorities is discussed. Then, we discuss how the implementation as described so far is augmented to support filter priorities.

A packet demultiplexor must be able to either reject a packet or forward the packet to just one application, even if several applications' filters accept the packet. While forwarding the packet to all the respective applications is possible (called *promiscuous mode*), one would like to somehow arbitrate in an intelligent manner such that only the correct application sees the packet. The DPF system as described so far returns the PID of the predicate with the most atoms, but this is clearly unacceptable for security reasons. Most packet filter systems do this by *prioritizing* filters [13, 15, 21]. Filters belonging to vital system applications are assigned higher priorities, while user applications are assigned low priorities. When several predicates can accept, we would like the one with the highest priority to be the one that forwards the packet to its application. However, in the implementation discussed so far, the final predicate that accepts is not necessarily the one with the highest priority.

The most direct way to augment the DPF implementation is as follows. First, the `predicate` structure is augmented by adding a `priority` integer field. The priority value is passed to DPF when the filter is inserted. We abide by the convention that smaller numbers indicate higher priorities.

Then, while emitting the "true" branch of an atom (in `true_emit`), if a predicate is accepting, all predicates with a lower priority are deactivated, in addition to the accepting

backpatching in the true-false tree walk needs to be recoded, since it depends on where in a branch instruction the offset field is located, and how large it is. (In particular, a port from MIPS [11] to Alpha [18] was simple, because the offsets were the same bits on both machines, but a port from MIPS to SPARC [10] was more involved, because the offsets were located in different parts of the branch instructions.) But, most of the porting work should be in writing a set of macro definitions (`binary.h`) for the new architecture. This should not be terribly time-consuming, as this work does not depend on what is happening at higher levels of abstraction (such as the tree walk). Macros can often be readily written by just consulting the instruction set manual for that architecture. This typically required only one or two days of intensive coding and debugging.

Hash table lookups must be emitted whenever **gen** is called on a disjunctive atom. After **gen** completes, we replace the final branch instruction with a set of instructions that compute the hash function and determine if any of the disjunction's values match the packet bitfield that was just looked up. If a match is found, acceptance code is emitted. Otherwise, a branch to the return instruction is set up. Backpatching occurs, as before, later in the true-false tree walk.

When all DCG has been completed, a pointer to the first instruction is returned. This pointer can be cast as a pointer to a function accepting three integer arguments and returning an integer. The three arguments are, in order:

1. The address of the first byte in the packet;
2. The length of the packet in bytes;
3. A pointer to an integer in which to place the **postpone** boolean.

The returned value is the PID of the accepting filter.

3.8 Summary

This chapter described the implementation of DPF's parsing, merging, and dynamic code generation of packet filters. The basic algorithms were discussed first, with the enhancements required by disjunctive atoms and priorities described later. Implementation details specific to dynamic code generation were discussed last. This section summarizes the major

aspects of each stage in DPF's operation, in the same order as presented throughout this chapter.

Filters are first parsed into a tree-like structure, in which the ordering of nodes are strictly defined in order to simplify the identification of atoms and predicates.

Then, parsed filters are inserted into the demultiplexor. Atoms are identified and compared with all previously-existing atoms. Identical atoms are completely merged, and atoms differing by just a constant are merged into disjunctive atoms. Atoms are stored in a queue in decreasing order of number of appearances. Unique predicates are stored in a separate list. Much bookkeeping is performed in both the atom queue and predicate list to keep track of which predicates atoms belong to. Filters are later removed by reversing these operations.

Third, code is generated for each atom in an order that traverses a true-false tree of atoms. Later execution of the resulting machine code is intended to follow a single path down this tree. Along this execution path, no atom is checked more than once, and the most common atoms are executed first. This ensures "optimal" demultiplexing. When a leaf is reached, one knows the result of applying each predicate to the given packet. Filter priorities arbitrate between predicates when more than one accept. This is done by using the given priority numbers to determine whether certain atoms should be emitted at given nodes in the tree.

Dynamic code generation is performed on each atom in a machine-independent way. When an instruction is emitted, instruction macros are used that can be defined differently on different architectures. Some simple optimizations are also implemented, including delay-slot filling, optimized bounds-checking, and immediate-mode instructions. Disjunctive atoms are supported directly in the machine code by looking up the values in a hash table. Postponement is supported by placing a "postpone" boolean in a predetermined address upon returning. A pointer to the resulting machine code is returned and can be cast as a pointer to a function. This function is subsequently called on incoming packets. It accepts a packet's base address, the packet's length, and a "postpone" boolean address as arguments, and returns the PID of the accepting or postponing predicate.

Chapter 4

Results of Experimental Evaluation

4.1 Introduction

This chapter describes and analyzes the results of testing DPF's functionality and performance.

Performance is compared head-to-head with that of the Mach packet filter (MPF) [21], a recent fast packet filter. Occasionally, comparisons to the "Pathfinder" packet classifier [4] will also be made. We could not run head-to-head comparisons with Pathfinder ourselves since we did not have access to its implementation, but their reported measurements are on the same architecture as the ones for DPF, and are included here.

All tests are run in user space on a 25 MHz DECstation 5000/200 running ULTRIX. This setup is used as the testbed for other packet filters in the literature [12, 21], and is used here to ensure that our test results can be compared to those reported in the literature [4, 21].

Unless otherwise indicated, sample filters to recognize TCP/IP [2] are used. DPF filters are stored in UNIX files and piped via standard input into a `dpf` executable, which contains the entire DPF implementation. The MPF implementation is lifted from the Mach kernel's source code and compiled into its own `mpf` executable. `mpf` initializes sample TCP/IP filters, inserts and deletes them, and uses them to demultiplex sample packets. For all tests, the DPF and MPF sample filters recognize the exact same sets of TCP/IP packets.

All reported timings are obtained as follows. `dpf` calls `getrusage`, an ULTRIX sys-

tem call that returns a given user's resource utilization, both before and after the timed operation. The difference between these two times is reported as the time spent in that operation. This timing method has a 3-millisecond resolution; this is problematic when microsecond times must be measured, and when the overhead of **getrusage** itself is a few milliseconds. So, all but one experiment consists of running 1,000,000 trials in a loop, and obtaining the results by dividing the time spent in that loop by 1,000,000. In this manner, reliable measurements to tenths of microseconds are obtained. Loop unrolling was done to minimize the overhead of the loop itself. The one exception was insertion and deletion of DPF filters, in which the operations' reported times were large enough to make multiple trials unnecessary. Times for inserting filters in MPF were taken from the literature [21]. These are the only MPF times which we could not measure ourselves. Running many trials in a loop and dividing the resulting time will not work for a filter insertion, since this operation has side-effects.

These results show that DPF demultiplexes packets by an order of magnitude or more faster than MPF in most cases. Both DPF's better filter merging and dynamic code generation account for this performance improvement. The use of disjunctive atoms also contributes to this performance improvement. However, our tests indicate that our DPF implementation yields this high demultiplexing performance at the expense of a high setup cost. The insertion and removal of filters in DPF takes considerably longer than in MPF. Yet, we expect that this setup cost, while high, can be an acceptable tradeoff for high demultiplexing performance. This tradeoff is discussed in more detail in Chapter 5.

Six different experiments are run, each testing the performance and functionality of a different part of the DPF system and comparing with MPF and Pathfinder, when applicable. They are:

1. Time to accept one packet;
2. Acceptance time of one filter as a function of protocol depth;
3. Acceptance time of similar vs. dissimilar filters;
4. Filter insertion/deletion time;
5. Functionality and overhead of fragment recognition/reassembly;
6. Functionality of priority-arbitrated demultiplexing.

Filter	Not Cached	Cached
MPF	71	36
Pathfinder	39	19
DPF	2.8	2.8

Table 4.1: Time for ten filters to recognize one TCP/IP packet. Times are in microseconds.

Detailed descriptions and the results of each test are described in the sections below.

4.2 Test #1: Time to Accept One Packet

This test measures the time required by ten filters to accept a TCP/IP packet. The filter in Figure 2-3 is used. This test is identical to performance tests done in the literature [4, 21].

The MPF equivalent is also tested on the same machine, and its results match those reported in the literature [21]. The Pathfinder results are copied from the literature [4].

Because interpretation is expensive, both MPF and Pathfinder maintain a cache of recently recognized filters; on packet arrival, the cached filters are checked first. Since packets intended for one application often come in bursts, this turns out to be a worthwhile optimization. In this experiment, we measure MPF's demultiplexing times for both cached and uncached filters. Given DPF's already superior performance, caching is not part of its current implementation.

For both MPF and DPF, the filter set was such that the last one inserted would be the last one checked. So, we inserted them such that *all* filters would be checked against the given packet, with the last one accepting the packet. This ensures a worst-case demultiplexing time, such that all possible rejecting filters are checked before the packet is finally accepted. We are interested in worst-case times for two reasons. First, it would have been more difficult to construct a fair average-case or best-case experiment that ensured that both packet filters executed the same set of instructions. Second, worst-case times better illustrate the differences between the different filters.

The results are in Table 4-1. DPF is 12 to 24 times faster than MPF and 7 to 14 times faster than Pathfinder, depending on whether or not the filter was cached. DPF is significantly faster than MPF for two reasons. First, MPF cannot merge these filters optimally; some redundant bitfield comparisons are performed. DPF's improved filter merging ensure that this does not happen. Second, DPF runs filters as machine code instead of interpreting

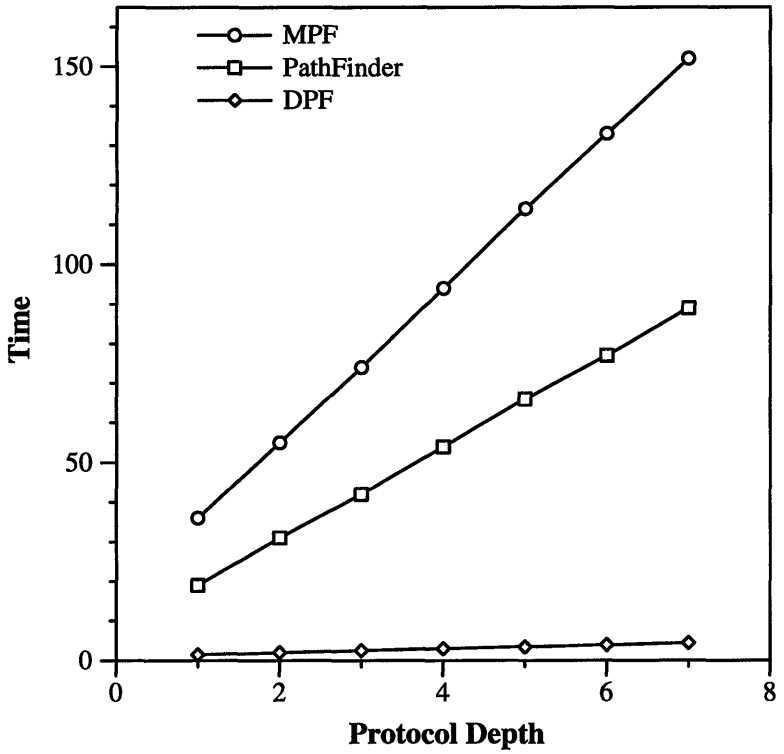


Figure 4-1: Time to recognize message versus protocol stack depth. Times are in microseconds.

them. Pathfinder, however, does perform optimal merging on this particular filter set. So, DPF's advantage over Pathfinder in this experiment is solely due to the running of filters as machine code.

4.3 Test #2: Acceptance Time vs Protocol Depth

This test measures the performance of DPF and other demultiplexors versus protocol stack depth. Each protocol header consisted of two fields of fixed length. In DPF, each field was checked by a simple atom that compared it with the correct constant. In MPF, this was done with a load instruction immediately followed by a conditional branch. We wrote and tested the MPF and DPF filters ourselves, while the Pathfinder results are copied without verification.

Figure 4.1 shows DPF's performance versus MPF and Pathfinder. The demultiplexing times for all packet filters grow linearly with the depth of the protocol stack, but DPF exhibits vastly superior performance due to running the filters as machine code. Here, DPF

was 12 – 20 times faster than Pathfinder, and 25 – 35 times faster than MPF, with the factor difference increasing with the protocol stack depth. Since only one filter is involved, the filters' merging algorithms do not come into play here. So, DPF's advantage here is solely due to running the filters as machine code instead of interpreting them.

The advantage increases with the number of instructions because the interpreted filters exhibit a constant overhead for each instruction. These interpreters run in a loop, and some instructions (incrementing instruction pointers and counters) must be performed at the end of each loop. In DPF's machine code, only a branch instruction is executed, thus saving some cycles.

4.4 Test #3: Similar vs. Dissimilar Filters

This test shows the advantage of the DPF optimizations over those of MPF. Both MPF and Pathfinder also attempt to merge similar filters, but filters are merged only when their *first* few instructions are identical. These contrast with DPF, in which filters can be merged based on identical atoms appearing anywhere in the filters.

DPF is compared with MPF by testing three different sets of filters:

1. A set of filters with completely dissimilar atoms;
2. A set of filters with many similarities occurring at their beginning, such that MPF can merge them;
3. A set of filters with similarities occurring elsewhere, such that MPF cannot merge them but DPF can.

Each filter is a variation of the TCP/IP filter shown in figure 2-3. Exactly how they differ is described in the respective subsections.

For each of these filter sets, we would like all filters to be checked on the given packet, with the last one checked accepting the packet. This ensures a worst-case measurement, as in test #1. We were careful to construct a set of packets that would be accepted by only the last filter to be checked (the last one to be inserted into the respective PF systems).

Comparisons to Pathfinder are not made here because the Pathfinder literature [4] does not include a test that is analogous to this experiment.

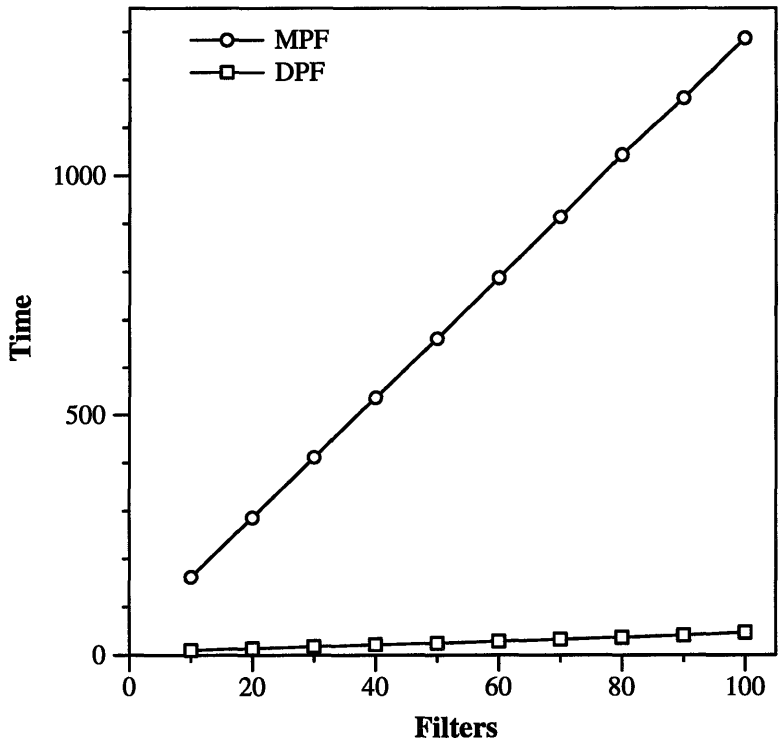


Figure 4-2: Time to recognize packets as a function of number of filters that cannot be merged by either MPF or DPF. Times are in microseconds.

4.4.1 Completely Dissimilar Filters

The first filter set, the set of *unmergeable* filters, was constructed by altering the atoms in the filter in Figure 2-3 by changing either the bitfield or the relational operator. Since both MPF and DPF require bitfields and relational operators to be identical for those instructions to be candidates for merging, neither will be able to merge any of these instructions. The last filter to be inserted is the original TCP filter, which is the only filter that can accept the test packets. Since it is the last filter inserted, it will be checked last in both MPF and DPF.

The results are in Figure 4-2. DPF shows a huge performance advantage, 15 to 28 times better, with the performance increasing with the number of filters. Since these filters are unmergeable in both demultiplexors, this improvement illustrates just the advantage of dynamic code generation. The difference increases with more filters because, in MPF, each failed filter results in its interpreter being called on the next filter. This represents a large constant overhead in the MPF software. On the other hand, DPF just issues a branch instruction to another filter's atom's machine code, resulting in much less overhead.

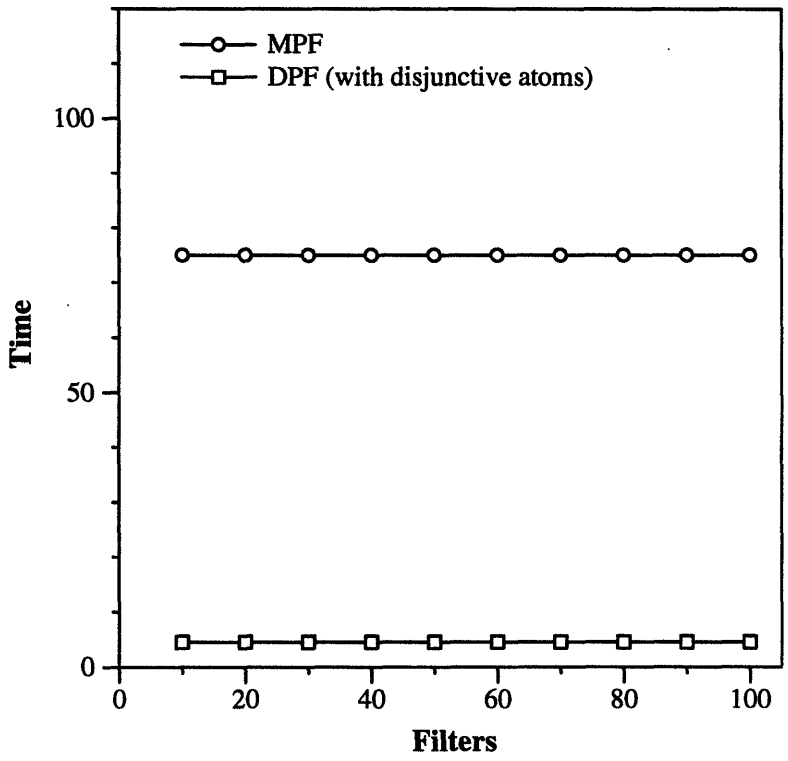


Figure 4-3: Time to recognize packets as a function of number of filters that can be merged into one filter by MPF. Times are in microseconds.

4.4.2 Similar Filters — Merged by MPF and DPF

The second filter set, the set of filters *mergeable* by *both* MPF and DPF, was constructed by altering the last atom appearing in the filter in Figure 2-3. Specifically, we changed the constant to which the last atom's bitfield is being compared. The resulting set of filters is ideal for MPF's optimizations, which attempts to merge filters with a common prefix and ending with comparisons that differ only by the constant to which a bitfield is being compared. MPF merges these into one filter, with the differing constants put into a hash table. DPF's disjunctive atoms are identical to this.

The results are shown in Figure 4-3. Both MPF and DPF exhibit constant acceptance times. MPF successfully merges these filters into just one filter plus a hash table with constant access time. DPF successfully merges these filters into a constant number of atoms, the last one of which is a disjunctive atom that stores the different constants. DPF's acceptance time is much faster than MPF's because DPF's demultiplexor is run as machine code. This accounts for a better than 15–fold improvement over MPF.

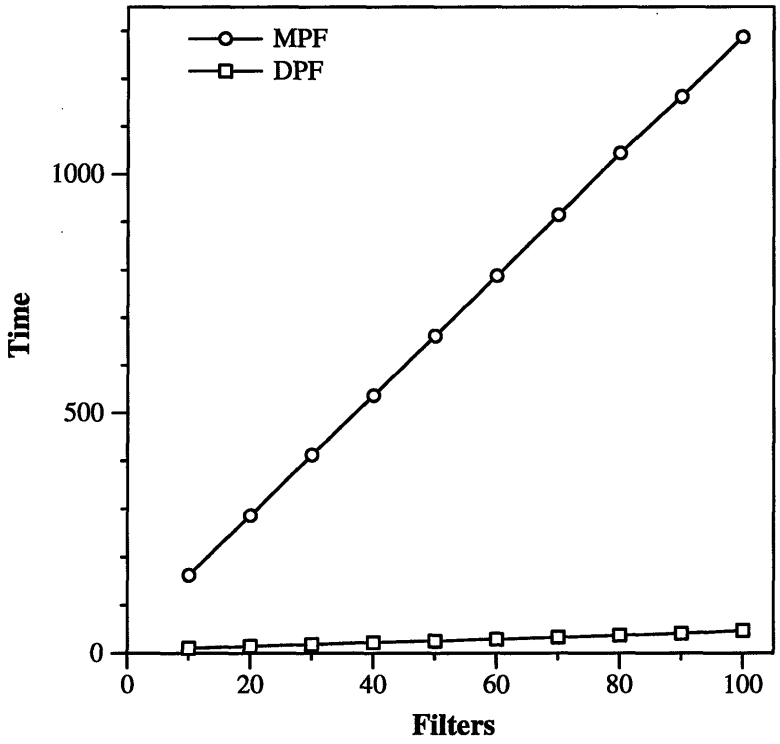


Figure 4-4: Time to recognize packets as a function of number of filters that cannot be merged into one filter by MPF, but can be merged by DPF. Times are in microseconds.

4.4.3 Similar Filters — Merged by DPF, Not Merged by MPF

The third set of filters, a set that is unmergeable by MPF but mergeable by DPF, was constructed by altering the *first* atom in the TCP filter shown in Figure 2-3. Thus, without a common prefix, MPF had to treat them as separate filters, while DPF could merge based on the common atoms appearing after the first.

The results are recorded in Figure 4-4. The results for MPF in Figure 4-4 are identical to the results in Figure 4-2, which showed the acceptance times with completely dissimilar filters. Because the first atom in each of these filter sets differs, MPF cannot merge them and treats each filter as separate. DPF, on the other hand, recognizes the identical atoms and arranges that they be executed first, with the dissimilar atoms being checked subsequently. Finally, the remaining true atom is the last atom to be checked.

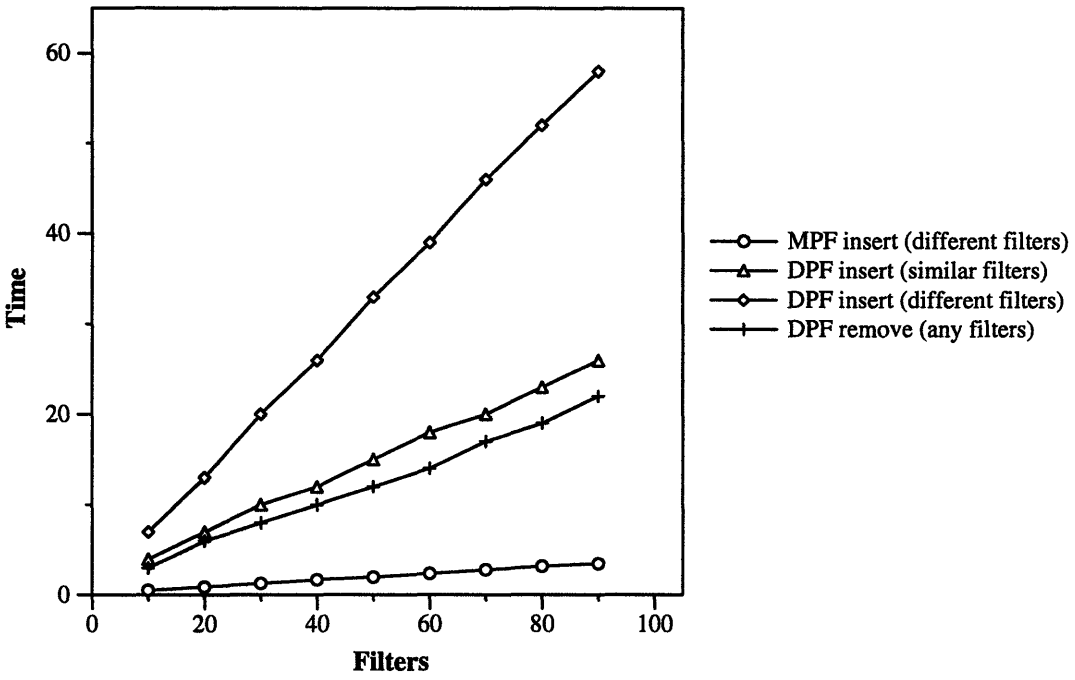


Figure 4-5: Time to insert and remove filters from demultiplexor. Times are in milliseconds. Poor DPF insert performance is exhibited, with an additional penalty if filters are not subject to merging. DPF remove performance has negligible dependence on whether filters are merged or not. Performance of MPF filter removal and insertion of similar filters is under 0.5 milliseconds regardless of the number of filters in the system.

4.5 Test #4: Insertion/Deletion Time

This test measures the time required to insert new filters into and remove inactive filters from the DPF system as a function of the number of filters active in the system. This test is performed with two sets of filters, both derived from the TCP filter in Figure 2-3. The first set consists of filters mergeable by both MPF and DPF, differing only by a constant in the last atom. The second set consists of filters unmergeable by both systems, with each atom having different bitfields. First, we insert and delete only mergeable filters into both filter systems and record the times. Second, as a separate test, we insert and delete only the unmergeable filters.

The results are shown in Table 4.2. They are also pictorially shown in Figure 4-5 to provide a better sense of how these times compare with each other. In general, DPF performed poorly for both insertion and deletion, with MPF outperforming DPF by approximately an order of magnitude. Also, unmergeable filters generally resulted in longer insert and remove

	Number of filters in system								
Operation	10	20	30	40	50	60	70	80	90
MPF Insert (diff filts)	0.5	0.9	1.3	1.7	2.0	2.4	2.8	3.2	3.5
DPF Insert (diff filts)	7	13	20	26	33	39	46	52	58
MPF Insert (sim filts)	0.20	0.22	0.24	0.26	0.28	0.29	0.31	0.33	0.35
DPF Insert (sim filts)	4	7	10	12	15	18	20	23	26
MPF Remove	N/A — Occurs implicitly while demultiplexing								
DPF Remove	3	6	8	10	12	14	17	19	22

Table 4.2: Table of times to insert and remove filters from DPF and MPF demultiplexors. Times are in milliseconds.

times for both systems, mostly due to the cost of manipulating the additional instances of data structures needed to represent them. Specific results of particular interest are itemized as follows.

1. The performance of insertion of similar filters using MPF is very small (under 0.5 milliseconds) and not shown in Figure 4-5. This is because all the filters were merged into just one filter, and whenever new filters are inserted, they are compared to just this one filter.

2. Using MPF, removal of inactive filters also takes negligible time, but this is because it is done during demultiplexing. While a packet is being run through the filters, each filter's process is checked if it is still alive; if not, that filter is quickly removed. Hence, filter removal in MPF is not a separate operation, and, as such, is difficult to measure.

3. Using MPF, insertion of unmergeable filters takes more time because the new filter must be compared to each filter in an increasingly long list of filters. Also, if it cannot be merged with any existing filter, memory must be allocated for it.

4. Insertion using DPF takes much longer than insertion using MPF because of the overhead of comparing each atom of a new filter with each unique atom that already exists, and of maintaining the longer list of predicates, regardless of mergeability. DPF's insertion of mergeable filters takes about eight times longer than MPF's insertion of unmergeable filters. Furthermore, if DPF cannot merge the filters, then there are more unique atoms to check, thus requiring more cycles. So, DPF's insertion of unmergeable filters takes slightly more than twice as long as DPF's insertion of

mergeable filters, and nearly 18 times longer than MPF's insertion of unmergeable filters.

5. Using DPF, filter removal also takes a significant number of cycles. However, it does not depend significantly on whether the set of filters in the system are mostly mergeable or mostly unmergeable. We conjecture that mergeable filters' advantage of having fewer unique atoms to check is offset by the additional bookkeeping that removals must perform on each atom.

6. Disjunctive atoms have no significant effect on DPF's insertion and removal times. Although disjunctive atoms reduce the number of unique atoms in the system, this seems to be offset by the additional manipulation of data within those atoms.

Some profiling was also done to determine where DPF spends most of its time during both insertions and removals. Approximately 70% of total cycles are used by the filter merging algorithms — specifically, identifying atoms in a new filter, comparing them with existing atoms, enqueuing them, and setting up the correct pointers between the new atom and predicate data structures — and the true-false tree walk. The other 30% was used by dynamic code generation. This 70-30 split was apparent regardless of how much overlap the set of filters exhibited or of how many filters were already in the system.

Table 4.3 shows some select profiling output, indicating where DPF spends most of its time during insertions and removals. This particular output is from an insertion with one hundred already-active filters, but the output does not change significantly for neither more nor less active filters. The tree-walk functions `true_emit` and `false_emit` take a little more than half of the total insert/delete execution time. `genatom`, which performs per-atom code generation and subsequent backpatching, requires another 16%. Collectively, these three mutually recursive functions account for a lot of cycles; one direction of future work should be reducing the cost of these or equivalent functions. The macros called within `genatom` account for another 17%. Lastly, the identification and comparison of atoms accounted for most of the remaining 15%.

Clearly, one pays for DPF's demultiplexing performance with a high setup cost. Running DPF's filter merging algorithms and dynamic code generation requires many machine cycles. Fortunately, insertions and deletions occur only when a network connection is opened and closed, actions which are usually infrequent.

Function	% Total Exec Time
`true_emit`	33
`false_emit`	18
`genatom`	16
DCG macros	17
Identify/compare atoms	15

Table 4.3: Percent of total insert/delete execution time spent by some select functions.

4.6 Test #5: Fragment Recognition/Reassembly

This test checks the functionality of DPF's fragment recognition and reassembly. The RSVP [22] filter in Figure 2-5 is used, along with some sample packet fragments that abide by IP's fragmentation paradigm [1]. Some other filters are inserted, some of whose atoms match those appearing in the RSVP filter. The fragments are then run through the DPF system in random orders. DPF successfully accepts and postpones fragments as specified. In particular, fragments appearing before the first fragment are successfully recognized by the "P" predicate and postponed. The first fragment is then correctly accepted, with the previously postponed predicates also being accepted. Then, later fragments are also properly recognized, presumably by the bracketed subsequent fragment predicate, which must have been correctly updated with the message ID and properly inserted.

This test also shows that the time required for a filter to postpone a packet is identical to the time required to accept a packet. This is because the same number of instructions are executed by the DPF implementation, regardless of whether or not a predicate is marked "postpone". A "postpone" boolean is always saved in a predetermined address upon acceptance, regardless of whether a postponement actually occurred.

However, there is some overhead to retesting previously postponed fragments once the first fragment arrives. This is the time required to check all postponed fragments with the "subsequent fragment" predicate. Since this predicate is small, this overhead is almost negligible: Approximately ten microseconds per fragment.

4.7 Test #6: Priorities

This simple test checks the proper handling of filter priorities whenever several filters can accept one given packet. For this test, several TCP filters, such as the one in Figure 2-3,

were used, each inserted with a different priority. Sample packets were carefully hand-coded so as to be acceptable by several filters. These packets were then run through the DPF system. DPF correctly returned the PID of the filter with the highest priority.

Priorities have no overhead on demultiplexing performance. Instead, priorities are handled during insertion and deletion of filters, during the true-false tree walk. Specifically, during `true_emit`, an extra `if` statement is executed for each predicate in the predicate list. However, profiling indicates that the overhead of this check is nearly negligible, adding less than 0.5% to the total execution time of an insertion or deletion.

4.8 Conclusions

Given these test results, we can conclude the following regarding the performance and functionality of the DPF implementation.

First, DPF is fast. In most cases, DPF demultiplexes packets at least an order of magnitude faster than MPF, the fastest packet filter in common use today. Test #1 indicates that dynamic code generation is responsible for much of this performance improvement, accounting by itself for 12 to 24 times more speed over MPF, depending on whether or not MPF cached the filter in question. Test #2 shows that this advantage improves even more for larger filters — up to 35 times faster than uncached MPF for filters with 16 atoms. Furthermore, test #3 shows that DPF's performance edge also improves when more filters need to be checked, from 15 times faster for 10 filters to 28 times faster for 100 filters. This is due both to DPF's dynamic code generation providing a quicker transition between filters, and to DPF's better merging of filters enforcing optimal demultiplexing, in that there are no redundant operations. MPF does not provide either of these advantages.

Second, insertion and removal of filters using DPF is slower than in the other filter systems against which DPF is compared. This is due to the overhead of performing filter merging and dynamic code generation. We expect that this is an acceptable tradeoff, considering the superior demultiplexing performance.

Chapter 5

Discussion

This chapter analyzes at a higher level the results reported in the previous chapter. We begin with a discussion of DPF's inherent tradeoff of obtaining high demultiplexing performance at the expense of a high setup cost. Then, we discuss how each of the three main ideas that DPF incorporates, dynamic code generation, complete filter merging, and a declarative language, contribute to DPF's superior demultiplexing performance. We close with a discussion of the drawbacks of DPF's current fragment recognition and reassembly framework, and offer suggestions for improvement in future implementations.

5.1 Performance Considerations

Clearly, DPF's demultiplexing performance is very good. It is much better than the demultiplexing performance of any other packet filtering system, and should be therefore able to handle the high bandwidths of modern fast networks.

The tradeoff for DPF's demultiplexing performance is the high setup cost of inserting and removing filters. Faster demultiplexing requires intelligent comparison, merging, and rearrangement of filter instructions, and this entails costly computation. And, mitigating this cost may result in slower demultiplexing. So, in determining whether DPF is appropriate for a particular system, one should take this tradeoff into consideration. Specifically, one should consider the following:

1. **The system's usage patterns.** If many network connections (i.e. more than hundreds) are often simultaneously open, but the rate of opening and closing connections is small, then DPF is most desirable. However, if connections are opened and closed

at a high frequency, then the setup cost may become problematic and one should tolerate slower demultiplexing.

2. **The nature of the network traffic.** If it is bursty and often fills all available bandwidth, faster demultiplexing may be necessary. One should then tolerate the higher setup cost.

3. **The speed of the network.** A relatively slow network (i.e. 1 Mbps Ethernet) does not require the fastest possible demultiplexing. A faster network (i.e. 155 Mbps ATM) may, thus necessitating toleration of a high setup cost.

Mitigating the setup cost is a suggestion for the direction of future research. In fact, work is already underway, in the form of another DPF implementation that reduces the setup cost by scaling back the filter merging algorithms without worsening demultiplexing performance.

5.2 The Effects of Dynamic Code Generation

Our tests show that dynamic code generation accounted for much of DPF's advantage in demultiplexing performance, accounting for an order of magnitude improvement in and of itself. Clearly, DCG was invaluable and is the way to go in future work. Moreover, we have demonstrated that DCG is not difficult to implement, contrary to opinions stated in the literature [15, 20].

An understandable concern is portability. One would like to install a DPF implementation on several different architectures without spending many man-hours retargeting DCG modules. Fortunately, there has been significant recent work in making DCG modules easier to retarget, and we made an effort to incorporate these techniques into our DPF implementation. In particular, our DCG is based upon the system built by Engler et al. [9], in which most machine dependency is hidden under a layer of instruction macros. Whenever an instruction needs to be emitted, a machine-independent macro is called which emits the code for the particular architecture. So, most of the work of a DPF port to a new architecture is simply to write a new set of macros. In our experience, an experienced programmer can do this in a day or two by simply consulting the reference manual for the new architecture's instruction set. The backpatching at higher levels may also need to be reimplemented, but

this is also a short and simple task.

5.3 The Effects of Filter Merging

Our tests also show an improvement in demultiplexing performance over "prefix-merging" demultiplexors (MPF and Pathfinder) due to DPF's ability to merge and reorder common instructions regardless of where they appear in the original filters. This ability results in an atom never being executed more than once on a particular packet, thus saving valuable cycles.

Also, the use of disjunctive atoms enables a finer-grained merging of atoms, namely, the very common occurrence of atoms differing by only the constant to which bitfields are being compared. This idea was originally implemented by MPF, but improved upon by DPF. While MPF allows disjunctions only at the end of filters, DPF can detect and merge disjunctions regardless of where the constituent atoms originally appeared in the original filters.

However, the price of these optimizations is the setup cost discussed above. Complex algorithms are required, and even though the use of a declarative language simplifies these algorithms, they still require many cycles. There is much room for improvement here. The profiling data given in Chapter 4 reveal the operations whose increased efficiency would mitigate the cost of global optimizations. In particular, attempts at simplifying the internal representations of atoms, predicates, and their global lists may be worthwhile. This would result in faster atom comparisons, queue management, and true-false tree traversal, operations which currently account for most of this setup cost.

5.4 The Effects of a Declarative Language

Although the overhead of filter merging is high, the use of a declarative language prevented it from being higher. It made comparing and merging instructions faster by providing a better means of describing what we are comparing — bitfield comparisons — in a simplified, standardized way. Enforcing a "canonical ordering" of tokens during parsing contributed to this. Comparing bitfield comparisons would have been more inefficient with an imperative language. Then, one would need to "intuit" what packet specification the filter writer "had in mind" when (s)he wrote a given non-standard, many-instructions-long piece of stack-

based or register-based imperative code. The declarative language intuitively captures the writer's intention, thus allowing for quicker identification of atoms and allowing a more efficient implementation of filter merging.

While it is difficult to quantify the advantage of a declarative language, one can still observe where it undoubtedly helps. First, atom and predicate recognition are faster. Since the syntax of the language matches the atom and predicate semantics very closely, traversing the parse tree in search of atoms and predicates is simplified. Second, atom comparisons are faster, for similar reasons. The parsed representation is a simple node tree that corresponds closely to our atom semantics, and our recursive comparison procedure is relatively fast and simple. While these are admittedly still absorbing many cycles during insertion and removal, they could easily absorb a lot more without the simpler internal representations made possible by the declarative language.

5.5 Improving Fragment Recognition and Reassembly

DPF's fragment recognition and reassembly paradigm can be improved. In particular, there are three problems in the current implementation that should be fixed in future implementations.

First, under the current framework, whenever a filter with a subsequent fragment predicate is submitted, that predicate is always activated after a packet is accepted, regardless of whether or not that packet is fragmented. This results in unnecessary computation when packets are unfragmented. A mechanism is needed by which DPF recognizes that an incoming packet is unfragmented and prevents activation of the subsequent fragment predicate. This is best done by augmenting the DPF grammar to allow defining a special atom that determines whether or not incoming packets are fragmented, thus determining whether the subsequent fragment predicate is necessary or not.

Second, the activation of a subsequent fragment predicate always results in a round of filter merging and dynamic code generation. This is problematic on networks in which many packets are fragmented, because *each* fragmented packet will result in the expensive activation of a subsequent fragment predicate. A better framework would be to keep these predicates separate, and allow subsequent fragments to continue to be caught by the postpone predicates. Then, fragments caught by the postpone predicates can be quickly

demultiplexed through the separate subsequent fragment predicates. While this would slightly worsen demultiplexing performance, it would save the high setup cost of inserting the subsequent fragment predicates for each fragmented packet.

Third, filters written for fragmented packets may reject unfragmented packets that should be accepted. This can occur whenever a header in an unfragmented packet differs from the header of the same packet subject to fragmentation. For example, in IP [1], a bit indicates whether the packet is fragmented or not. Filters not checking this bit may reject unfragmented packets. A solution would be to extend the grammar to express acceptance of unfragmented packets in filters written for fragmented packets.

Chapter 6

Related Work

This chapter discusses other work that has been done related to this thesis. We begin with Mogul's stack-based packet filter, which pioneered the notion of applications submitting filters to the kernel, and move on through the BSD filter, the Mach filter and Pathfinder, discussing how each was an improvement over its predecessors. Comparisons between DPF and these earlier filters are made. We also briefly discuss how dynamic code generation has related specifically to packet filtering.

6.1 CSPF

The CMU/Stanford packet filter ("CSPF") [15], designed and implemented by Mogul et al. in 1980, was the first attempt at a packet demultiplexing system in which filters were submitted from user space. Prior to this, filters were installed directly by hand, usually one per protocol.

Mogul et al. introduced such a user-level packet filter, in which applications submitted packet specifications written in a simple, safe, stack-based imperative language to a protocol-independent interpreter. Performance was sufficient to support the bandwidth of the Ethernet networks on which this packet-filter system was run. However, performance deteriorated rapidly as more filters were inserted into the demultiplexor, since each filter was treated separately and executed in turn on all incoming packets.

This performance deficiency was noted and improvements were suggested in the literature. First, it was noted that using a register-based language with branches instead of the stack language could improve interpreting speed. Furthermore, the idea of compiling filters

into machine code was raised, but then discarded for the reason that it would come "at the cost of greatly increased implementation complexity" [15].

Also, the earliest versions of CSPF did not support indirect loads, hence could not be used for protocols with optional fields (such as IP). This was supported in later versions, but required significantly complicating the stack language.

6.2 BPF

The Berkeley packet filter ("BPF") [13], introduced by McCanne and Jacobson in 1993, implemented the register-based language suggested by Mogul et al., supporting operations on an accumulator and an index register, loads and stores from scratch memory, and conditional branches. This packet filter was an improvement over CSPF on several fronts. First and foremost, it cut CSPF's demultiplexing time by more than half. Second, a variety of addressing modes were implemented that simplified packet bitfield loads, most notably indirect loads from variable-length protocol headers.

BPF's register-based interpreter was significantly faster than CSPF for several reasons. First, it took advantage of the fast register logic and arithmetic available on most modern RISC architectures. Second, it ensured fewer memory loads and stores, which are more expensive on modern machines and which occur often in stack-based interpreters. Third, the availability of branch instructions allowed the interpreter to jump over redundant code that was known to be true or false during run-time, something that the CSPF stack machine could not easily do. This last improvement was a first effort at ensuring that operations were executed at most once, although its scope was limited to one filter at a time. This effort extended across multiple filters in later filter systems and culminated in our DPF implementation.

Finally, the notion of an *atom* began here. The most common sequence of instructions was a bitfield load immediately followed by a conditional branch. These were readily recognizable as the boolean operations on packet bitfields to which we refer as atoms. Furthermore, since application writers usually used BSD system library functions to write their packet filters, these sequences became fairly standardized. However, these standards were not enforced anywhere.

6.3 MPF

In the beginning of 1994, Yuhara et al., noted two properties of existing user-level packet filters (mostly BPF) that could allow a faster implementation. First, many BPF filters were written using standard libraries, such that filters performing the same operations were usually written with the same instruction sequences. Secondly, many simultaneously-active filters on any given system performed similar operations, especially at their beginning (known as *prefixes*). For example, many filters recognizing TCP/IP [1, 2], UDP/IP [16], VMTP/IP [6], etc. would have an identical sequence of instructions at their beginning, namely the IP portion. Moreover, there were often several simultaneously active filters for the same full protocol stack (such as TCP/IP) with identical beginnings, and whose endings also had very similar instructions, in that they loaded the same bitfields (such as port numbers) but compared them to different values. These bitfield comparisons differing by just a constant are implemented in DPF as "disjunctive atoms".

The Mach packet filter (MPF), then, was designed as an optimization of BPF such that these similarities could be recognized and exploited by merging those filters together. Specifically, filters that were identical except for the disjunctions at the end were collapsed into one filter, with the differing constants stored in a hash table. Even though this seemed restrictive in that differences occurring anywhere other than a disjunction at the end of otherwise identical filters precluded merging those filters, it took advantage of the fact that application writers used standardized library functions to write their filters. Many BPF filters for common protocols were in a form that invited this kind of optimization.

The resulting demultiplexing performance improvement was significant, especially when many filters were active. In any large given set of filters, many would exhibit this similarity and be merged, thus reducing the number of filters that the interpreter would check in turn on incoming packets. With over 100 similar filters active, a fifty-fold improvement over BPF and a 100-fold improvement over CSPF were realized. However, a cost was incurred on filter insertions, in that a new filter had to be compared with each existing filter for possible merging. This cost was, however, considered an acceptable tradeoff for fast demultiplexing with many filters.

6.4 PathFinder

Pathfinder, published at the end of 1994 by Bailey et al, provided an improvement upon MPF's optimizations by virtue of using a declarative language. The Pathfinder language also captures the notion of bitfield comparisons as one unit of computation, similar to the DPF language's notion of atoms.

Pathfinder also extended the central idea behind MPF's prefix and disjunction merging. Specifically, what made MPF's optimizations work was that most filters would check a packet against both levels of a two-level protocol stack in turn. So, many merged filters would check for the same values in a lower-level protocol (i.e. IP), and would then look for different values in the second-level protocol (i.e. TCP). However, this paradigm broke down when deeper protocol stacks were realized; merging would then not occur, even if many similarities between filters existed in checking for the lower-level protocols. Pathfinder implemented efficient merging of filters for arbitrarily deep protocol stacks. Pathfinder's instruction sequence down this stack could be visualized as a directed acyclic graph (DAG), with branches diverging at each protocol level. For example, after the IP header, one can diverge to UDP or TCP, and after the TCP header, one can diverge to FTP [17] or RPC [14]. Pathfinder filters (or *classifiers*, as defined in the literature), could be merged into the existing DAG for as many levels as a similarity existed before diverging to another branch. This was an improvement over MPF, which could merge only two levels deep.

According to the literature [4], using a declarative language made a fast implementation of the interpreter possible. Since bitfield comparisons were treated as one instruction, this allowed for quicker comparisons between instructions and quicker subsequent merging into the DAG. Demultiplexing performance was cut in half, an impressive achievement considering that filters were still interpreted. However, the setup cost involved in managing the DAG was twice that of MPF. Still, Bailey et al. concluded that this was an acceptable tradeoff for faster demultiplexing.

We observe that Pathfinder's DAG bears a resemblance to our DPF implementation's true-false tree. Whereas DPF looks for the most common instructions explicitly, Pathfinder assumes that instructions checking lower-level protocol headers are the most common — a dangerous assumption, but often accurate. So, both gain performance by arranging instructions into a tree-like structure such that one executes the most common instructions

first, then diverges to less common instructions. But, while Pathfinder diverges at each protocol header, DPF diverges at each instruction. Also, the order in which Pathfinder checks protocol headers is determined by the order in which the related instructions appear in the original filters, and is assumed to be in increasing order of bitfield offset. So, a Pathfinder classifier writer must be sure that all bitfield comparisons appear in increasing order and that all instructions for a given protocol header must appear together in order for Pathfinder's optimizations to be effective. In contrast, since DPF can rearrange atoms regardless of where they appeared in the original filter, a DPF filter writer is not bound by such constraints.

6.5 Dynamic Code Generation

The idea of using dynamic code generation to improve demultiplexing performance is an old one. However, to our knowledge, DPF is the first packet demultiplexor to actually implement it. Mogul et al. [15] was the first to recognize the advantage of DCG in packet filtering, although they deemed its implementation "too complicated". Thekkath et al. [19, 20] also discussed DCG in the context of filtering ATM packets, but they did not implement it for similar reasons. Engler et al. [9] provided simple measurements of DCG applied to a simple PUP [5] packet filter, thus showing the vast potential of DCG for improving demultiplexing performance. It was this result that inspired the use of DCG in our DPF implementation.

Chapter 7

Conclusions

We have presented a novel user-level packet filter system that addresses the problem of demultiplexing performance on fast networks. This was done by incorporating the following three ideas:

1. **Dynamic code generation.** Compiling filters upon the insertion and removal of filters and then running the machine code on incoming packets proves to be desirable over the alternative: slow interpretation of filters. In fact, this alone accounts for nearly an order-of-magnitude speed improvement over existing packet demultiplexors.

2. **Declarative Language.** Previous packet filters' imperative languages provided a semantics that was too low-level for the task of packet demultiplexing and resulted in inefficient implementations. A declarative language provides a semantics that is closer to the notion of a packet filter really being a boolean function on a packet, and allows a more efficient implementation of DPF's filter merging and dynamic code generation.

3. **Complete Filter Merging.** Previous demultiplexors did not fully recognize and exploit overlap between subsequently active filters, at the expense of demultiplexing performance. DPF identifies and merges common instructions regardless of their context within a filter, and ensures that no instruction is executed more than once on a given packet.

Both the declarative language and dynamic code generation proved to be invaluable to providing fast demultiplexing. DPF's complete merging of filters also contributed towards its fast demultiplexing, but it required a set of costly algorithms that contributed to DPF's

large setup time on filter insertions and removals. Mitigating this time is a suggestion for the direction of future research. In fact, some work has already been done, in the form of an alternative DPF implementation that reduces the setup time by scaling back the filter merging algorithms without worsening demultiplexing performance.

Bibliography

[1] Defense Advanced Research Projects Agency. Internet protocol. RFC: 791, September 1981.

[2] Defense Advanced Research Projects Agency. Transmission control protocol. RFC: 793, September 1981.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[4] M. Bailey, B. Gopal, M. Pagels, L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 115–123, Monterrey, CA, November 1994.

[5] D. Boggs, J. Shoch, E. Taft, and R. Metcalfe. PUP: An internetwork architecture. Technical Report CSL-79-10, Xerox PARC, 1979.

[6] D. Cheriton. VMTP: Versatile message transaction protocol. RFC: 1045, February 1988.

[7] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the 1990 ACM SIGCOMM Symposium*, pages 200–208, Philadelphia, PA, September 1990.

[8] P. Druschel, M.B. Abbott, M.A. Pagels, and L.L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.

[9] D. Engler and T. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of ASPLOS 6*, pages 263–272, San Jose, CA, November 1994.

[10] SPARC International. *The SPARC Architecture Manual.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[11] G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[12] C. Maeda. Private Communication, October 1994.

[13] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269, San Diego, CA, January 1993.

[14] Sun Microsystems. RPC: Remote procedure call protocol specification. RFC: 1057, June 1988.

[15] J. Mogul, R. Rahid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987.

[16] J. Postel. User datagram protocol. RFC: 768, August 1980.

[17] J. Postel and J. Reynolds. File transfer protocol. RFC: 959, October 1985.

[18] R. Sites. *Alpha architecture reference manual.* Digital Press, Burlington, MA, 1992.

[19] C. Thekkath, H. Levy, and E. Lazowska. Efficient support for multicomputing on ATM networks. Technical Report TR93-04-03, University of Washington, April 1993.

[20] C. Thekkath, T. Nguyen, E. Moy, and E. Lazowska. Implementing network protocols at user level. In *Proceedings of the 1993 SIGCOMM Symposium on Communications Architecture, Protocols and Applications*, pages 64–73, San Francisco, CA, September 1993.

[21] M. Yuhara, B. Bershad, C. Maeda, and J. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the Winter 1994 USENIX Conference*, pages 153–165, San Francisco, CA, January 1994.

[22] L. Zhang, R. Braden, D. Estrin, S. Herzog, and S. Jamin. Resource reservation protocol. IETF Draft, Version 0.5, November 1994.