APF : A Modular Language for Fast Packet Classification *

H. Dan Lambright *Saumya K. Debray*
Department of Computer Science
University of Arizona
Tucson, AZ 85721, USA
{hdlambri, debray}@cs.arizona.edu

August 30, 1996

Abstract

Fast packet classification—that is, the determination of the destination of a network packet—is of fundamental importance in high-performance network systems. Additionally, for flexibility reasons, it is desirable to allow applications to use their own high-level protocols where appropriate. Taken together, this demands a mechanism that permits the specification of protocols in a simple, flexible, and modular way without sacrificing performance. This paper describes APF, a language for specifying packet classifiers. The simple declarative syntax of this language makes it easy to specify even fairly complex packet structures in a clean and modular way, thereby improving reliability and maintainability. It also effects a clean separation between the specification of packet classifiers and their implementations, thereby making it possible to choose from a variety of implementations with different performance tradeoffs. In particular, as our experimental results illustrate, it can be compiled to efficient code whose speed can surpass that of well-known packet classifiers such as Pathfinder and BPF.

* The work of S. Debray was supported in part by the National Science Foundation under grant number CCR-9502826.

1 Introduction

A fundamental activity in any networked system is the identification of the destination process for a network packet as it arrives off the network interface. This component of the network driver's critical path is referred to as *packet classification*. If this activity cannot be performed with a high degree of efficiency, there is little hope that any network-oriented application program will be able to accomplish acceptable performance. For this reason, packet classification has been targeted by numerous researchers as a potential performance bottleneck that may be amenable to optimization in various forms (see, for example, [1, 3, 7, 9, 10]).

Because of the rapid spread and evolution of networked applications in recent years, the quest for high-performance packet classification has been paralleled by a demand for the ability to specify and use flexible application-specific protocols. The use of application-specific protocols, in turn, implies that packet classifiers—also referred to in the literature as packet filters—must be able to deal with packets whose structures are defined in application-specific ways. In other words, an application must be able to (dynamically) communicate the structure of its packets to the packet classifier; and the packet classifier must be prepared to handle such specifications, and recognize the corresponding packets, correctly and efficiently. This requires a language that can be used to specify which packets to recognize. Such a language should satisfy the following criteria:

Generality. The language must be general enough to specify arbitrary protocols whose packet formats may not be known ahead of time.

Simplicity. The specifications should be relatively simple, to simplify testing and debugging and enhance readability and maintainability.

Efficiency. The classifiers should compile into machine code that can execute at network speeds; this becomes both increasingly important, and increasingly difficult, with the introduction of gigabyte networks.

Safety. It should be possible to deal with situations such as fragmentation and out of order packet delivery.

Flexibility. It must be possible, at run-time, for new classifiers to be added or old classifiers to be deleted quickly, without interfering with the processing of incoming data. This requirement is especially important in today's high speed servers, which may have hundreds of transient connections active at once.

An essential requirement for meeting the first two requirements is that of modularity: if we want to be able to specify arbitrary protocols without sacrificing clarity, it is essential to be able to write modular specifications and to compose such specifications in different ways in order to achieve the desired effects. One might imagine that the third criterion listed above, namely, efficiency, would be difficult to attain using a modular specification language. In this paper, we describe a language for packet classifiers that shows that this is not the case: indeed, our results indicate that a declarative specification language that supports modularity and separates the specification from the implementation offers implementors a way to choose from among a number of implementation alternatives, with different tradeoffs between different performance parameters. Thus, such a language enhances the implementation flexibility of network packet classifiers considerably.

Historically, packet classifiers have typically been specified using virtual machine instruction sets: this is the case, for example, in the Berkeley Packet Filter (BPF) used in many Unix systems [9] and the Mach Packet Filter (MPF) used in the Mach operating system [10]. This approach has the disadvantage that the specification of a protocol in terms of a low-level instruction set can be tedious and potentially error-prone, and the hand-translated code may or may not exploit all of the optimizations that are possible in a given context. Moreover, the virtual machine instructions are typically interpreted at run time, and this incurs a performance penalty. Finally, since such virtual machine instruction sets are designed to abstract away from particular machine architectures, they make it difficult to take advantage of knowledge about a machine architecture (e.g., about caches or pipelines) or application mix (e.g., the frequency distribution of different kinds of packets) in a packet classifier implementation. A more declarative approach to protocol classification scheme is taken by the Pathfinder system [1], where packet classifiers are specified using directed acyclic graphs that specify, in essence, the transition diagram of a nondeterministic finite automaton. The specification language for Pathfinder focuses on performance, sacrificing some expressive power in the process. Cytron *et al.* have described a language for specifying protocols using context-free grammars: they propose using tools such as `yacc` to compile such specifications into executable code [4]; however, they do not argue convincingly either for the need for the full generality of context-free grammars, or for the practicality (with regards to speed) of the code that might be generated from their specifications.

We use an approach that is fundamentally different from these. We start from the observation that packet classifiers are, in essence, finite automata. However, instead of requiring that the automata be specified directly (e.g., as Pathfinder does), we feel that a simpler and more natural specification style is to use a regular grammar. Such grammars are compact and easy to read, understand and maintain, and are easy to compose with other grammars. Our compiler translates such a specification into a finite automaton. This automaton can be subjected to a variety of equivalence-preserving optimizing transformations (e.g., a sequence of transitions can be checked at once by using a word-compare instruction instead of a sequence of byte-compares). Moreover, there are a number of alternatives available for implementing a finite automaton—at a high level, it is possible to simulate a nondeterministic automaton, or to translate to a deterministic automaton (which may or may not be subjected to state minimization); at a low level, there are a variety of choices for translating the transitions of a finite automaton into machine code, e.g., using a tree of conditional branches or an indirect jump through a jump table. Since none of the implementation decisions are hard-wired to the specification, the implementor has considerable freedom in choosing an implementation with the tradeoff between different performance parameters that is appropriate to a given situation.

2 APF: Language Design Considerations

Our primary design goals for APF were to keep the set of language constructs small and simple, while maximizing their expressive power, so as to be able to represent the widest range of protocols possible. However, we were willing to sacrifice the ability to represent exotic features of some of the more obscure protocols rather then compromise our primary goal of simplicity.

In order to design a language for describing communication protocol packets, it is necessary to categorize certain common protocol construction techniques. Broadly speaking, information in network packets is grouped together into fields. Depending on the protocol, the length (in bytes) of these fields can be either *fixed*, i.e., known at protocol specification time, or *variable*, i.e., not known until run time; in the latter case, the protocol must specify some mechanism to derive the actual length of the fields at runtime. Correspondingly, the language describing such protocols should be able to specify the first situation in a simple way, while providing a means to handle the latter situation in a convenient and natural way. The number of such fields can also be unknown at protocol specification time: for example, many protocols allow for some number of optional fields to be specified. As in the variable length field case, the packet classifier language must somehow represent the decoding mechanism by which the number of fields is derived from the packet at run-time.

There are four commonly-used methods by which such run-time values can be encoded:

1. The value may be explicitly given in some field in the packet whose location is known at protocol specification time.

2. The length of the field may be encoded within a part of a field, and must be decoded using some transformation. This is the case, for example, with the IP protocol.

3. The length of the field may have been specified explicitly or implicitly in a previous packet; thus, a state variable must be set and then kept live between packet arrivals.

4. Instead of specifying the length or number of fields explicitly, the end of a variable-sized field (or sequence of fields) may be marked by a sentinel.

The first of these is straightforward to handle. The second and third cases can be handled by extracting the relevant information from a packet and using it to guide the subsequent computation. A similar mechanism suffices for the last case as well. This suggests that all but the most complex classifiers can be represented if the specification language supports pattern-matching using finite automata, augmented with a mechanism by which data can be extracted from a packet at run-time, transformed via logical or arithmetic operations, and then stored into a register. These registers can be used for the purposes of decoding the protocol further along in the classifier, or in a subsequent computation in an expression.

While finite automata are conceptually straightforward, large automata may not always be easy to describe directly (especially if we want to augment them to account for variable-length or optional fields, as discussed above). Nor are they especially modular: for example, determining the deterministic finite automaton obtained from composing two smaller finite automata can involve a nontrivial amount of work. Indeed, finite automata are better thought of as an *implementation* of packet classifiers than as a *specification*. For these reasons, we decided to use a simpler and more declarative means for specifying finite automata, namely, (right-linear) regular grammars. Such a grammar is a context-free grammar, with variables V and terminals T, where each production is of the form

$$\begin{array}{ll} X \longrightarrow a_1 \cdots a_k & a_i \in T, 1 \leq i \leq k; \text{ or} \\ X \longrightarrow a_1 \cdots a_k Y & a_i \in T, 1 \leq i \leq k; \; Y \in V \end{array}$$

Fundamentally, a script in APF is simply a list of such productions, written in the manner of `yacc` [8]: that is, a production '$X \longrightarrow a_1 \cdots a_k Y$' is written

```
X : a1 ... ak Y ;
```

As a first approximation, a packet will satisfy a classifier if a sequence of productions linking the start to the end states can be matched against the packet.

As a matter of convenience, APF supports the naming of common patterns in a "header portion" of the specification file. For example, an 8-bit pattern to mask out the two most significant bits of a byte could be defined as

```
Low6Bits = 0x3f:8
```

The value of this pattern can subsequently be obtained by writing '`#Low6Bits`'. It is also possible to specify a fixed number of repetitions of a pattern: the pattern 'P`*`k' denotes k repetitions of the pattern P. For example, '`bit`' stands for a "don't-care" bit, and '`bit*4`' represents 4 repetitions of '`bit`'—that is, four don't-care bits.

```
/*
 *  Constant definitions
 */
ETHERTYPE     = 0x8:16;
IPPROTO_TCP   = 20:8;
TCP_DEST_PORT = 0x2356:16;
%%
/*
 * Protocol classifier specification
 */
S : byte*12 B;
B : #ETHERTYPE C;  /* check for ip protocol in ethernet header */
C : bit*4 bit*4 {val=ptoint($2)*4-10);} D;
                   /* load ip length into val */
D : byte*8 E;
E : #IPPROTO_TCP F; /* check for tcp protocol */
F : byte*val;
G : byte*2 H;      /* skip past source port */
H : #TCP_DEST_PORT  {exit(1);};  /* check destination port */
```

Figure 1: `APF` Classifier for TCP Protocol packets

Semantic actions can be embedded within the bodies of productions. Such actions are of the form

$$\{\ stmt_1;\ \cdots\ stmt_k;\ \}$$

where $stmt_i$ is either a simple assignment or a conditional statement. This allows the extraction and manipulation of run-time information, which is necessary for handling variable-sized fields and optional fields. This is illustrated in Example 1, which shows the specification for the TCP protocol. This specification contains a production

```
C: bit*4 bit*4 {val = ptoint($2)*4-10);} D;
```

Here, the `$2` refers to the pattern that matched the second symbol in the body of the production, i.e., the second `bit*4`. By itself, this is just a bit sequence: the function *ptoint* converts this to a (signed) integer, and the statement '`val =` *ptoint*`($2)*4-11`' computes the displacement of the end of the variable-length field and stores the result in the variable `val`. We allow only a restricted class of statements within such semantic actions primarily for safety reasons.

The *width* of a variable in APF may be a byte, a halfword, a word, or some number of bits. Due to performance considerations, we limit the flexibility of bit-width variables: APF requires that these should occur in groups whose total width is a multiple of 8 (i.e., an integral number of bytes). We know of no important protocol that does not group data into byte octets.

The language features described above can be used in a straightforward way to support data-driven iteration, using conditional statements within semantic actions to terminate the iteration. This is useful in searching for a token in a list of fields when the field the token lies in is unknown. For example, optional headers in IPv4 may be specified in any order, so a classifier checking for just one of those headers must check each one. This is illustrated in Example 2, which specifies a complex filter that returns 1 if the packet is a TCP packet containing an optional timestamp header. A similar mechanism can be used for length fields in ASN.1, which may be constructed such that they are terminated by a sentenial bit: each byte in the field must be checked for this.

3 Implementation Issues

Correctness of an APF implementation requires only that the code generated for any input specification should correctly simulate the behavior of the corresponding finite automaton. Since the language does not predetermine the actual operational behavior of such an automaton, it is possible to have a wide variety of different implementations that offer the same functionality, but differ in the details of how the finite automaton is simulated, and therefore in the performance parameter tradeoffs they offer. Examples of high-level implementation alternatives include the following:

1. In the simplest approach, it is possible to translate the input specification to a nondeterministic finite automaton, then simulate the possible executions of this automaton at runtime. This approach offers quick installation, at the expense of increased runtime overheads.

2. The initial nondeterministic automaton can be transformed to an equivalent deterministic automaton using the subset construction. This is somewhat more expensive at compile time, with a concomitant improvement in runtime speed.

3. We can carry out state minimization on the deterministic automaton. This potentially leads to a reduction in code size, which can lead to improved cache utilization.

An orthogonal set of low-level implementation alternatives also present themselves:

1. The finite automaton may be encoded as a data structure (e.g., a transition table or a graph) that is interpreted. Such an approach would closely resemble that of Pathfinder [1].

```
/* IP version 4 classifier with optional headers */
/* Return 1 if packet is TCP with optional timestamp header */

/*
 *  Constant definitions
 */
ETHERTYPE       = 0x8:16;
IPPROTO_TCP     = 20:8;
END_OF_OP_LIST  = 0:5;
NO_OP           = 1:5;
TIMESTAMP       = 4:5;
%%
/*
 * Protocol classifier specification
 */
S : byte*12 B;
B : #ETHERTYPE C;
C : bit*4 bit*4 D {val=ptoint($2)*4;}
                     /* length is specified in 32 bit words */
D : byte*8 E;
E : #IPPROTO_TCP F;
F : byte*10 G     {val=val-20; if (val<=0) exit(0);};
                     /* no options specified */
G : bit*3 H;
H : #END_OF_OP_LIST; {exit(0);}       /* end of option list */
H : #NO_OP 1 I;   {val=val-1; if (val<=0) exit(0);};
                     /* no operation */
H : #TIMESTAMP  4 {exit(1);};         /* internet timestamp */
I : byte byte J   {val2=ptoint($2);};
J : byte*val2 G   {val=val-val2; if (val<=0) exit(0);};
```

Figure 2: APF Classifier for IP version 4 packets with optional headers

2. The automaton may be translated into an abstract machine instruction set that is then interpreted. The result would be similar, in principle, to the Berkeley Packet Filter [9] and the Mach Packet Filter [10]. However, our approach retains some additional flexibility: for example, a variety of different strategies for implementing abstract machines, such as byte code or threaded code, can be used.

3. The finite automaton may be compiled directly into executable machine code. This is potentially more expensive at compile time, though it is likely to be faster at runtime. This is the approach taken in the current prototype, though we plan to support other approaches in a more mature system. Here again, there are a host of further low-level alternatives to consider, such as low-level representation issues (e.g., the use of binary decision trees vs. jump tables in implementing the transitions out of a state), code generation strategies to improve locality, and transformations on the finite automaton to reduce memory traffic. We omit a more complete discussion due to space limitations.

4. The automaton may be compiled into abstract machine code that is then incrementally compiled to native code at runtime (see, e.g., [2, 5, 6, 7]). Such an approach would offer both quick installation and runtime efficiency.

Indeed, such alternative implementation strategies can be supported within the same compiler, and selected by the user, so as to offer a great deal of overall flexibility.

In the current prototype implementation, an input specification is first translated into a nondeterministic finite automaton, then converted to an equivalent deterministic automaton. A situation that very commonly arises here is that there may be a (large) group of classifiers that are identical except for at the very end, where they accept by checking against different constants [10]. This may be the case, for example, on a large server with many TCP connections: the classifiers for the different connections are identical except for the check at the end for the unique session key that identifies the particular connection. The transformation to a deterministic automaton effectively "coalesces" the individual classifiers into a single automaton that has a number of transitions based on the session key out of a single state at the end. A general-purpose transformation thus accomplishes the same result as a special-purpose optimization implemented in the Mach Packet Filter [10].

The resulting DFA can still have inefficiencies with respect to the memory traffic it induces. It is subjected to a tiling transformation aimed at minimizing the number of memory accesses. The idea is to try and group nodes together in such a way as to allow individual memory loads to access the largest chunk of data transferable at one time across the machine's bus (the word width of the machine can be specified as a compile-time option). As an example, consider the productions

```
A : byte*2 0x3f:8 0xff:8 ;
A : byte*3 0x0:8 ;
```

This specifies two transitions out of the state `A`: the first matches two don't-care bytes, followed by a byte matching the bit pattern `00111111`, followed by a byte containing only `1`'s; the second transition matches three don't-care bytes followed by a byte of `0`'s. A straightforward implementation might skip the first two bytes, load the third byte and test it, then load and test the fourth byte: a total of two memory accesses. Our tiling algorithm will transform this to a single 4-byte memory load, followed by mask operations and tests on the register containing the word just loaded. This can lead to significant reductions in the total amount of memory traffic. The subsequent masking operations that are necessary are, in many cases, essentially free, since most RISC architectures cannot operate on individual bytes or half-words, but must expand them to word-sized values in any case. However, this transformation can pose alignment problems: when there are loops in the DFA, and when there are variable length fields, it is impossible to predict the subsequent node's data alignment at compile time. Our solution to this problem is to construct different versions of the relevant subautomata that are specialized for the different alignments that may be encountered at a node. We believe that the improvement in speed resulting from the reduction in memory traffic is worth the resulting increase in code size.

The final aspect of code generation which needs to be considered is safety. An invalid length field loaded into the classifier engine could cause accesses to be made outside of the packet's memory buffer. We used DPF's method to mitigate the costs of checking the legality of memory references [7]. Briefly, Engler describes a scheme entitled "bounds-check aggregation" in which a check compares the most distant known offset in the DFA before a branch is made or a variable length field reached, rather then checking every memory reference individually.

4 Performance

For this prototype implementation of APF, we measured two performance metrics:

1. *latency* : this is the time taken to classify a packet (measured for an increasing number of classifiers); and
2. *scalability* : this measures how well APF scales as classifiers grow in size, i.e., the time taken to classify a packet as the number of different layers of protocols increases.

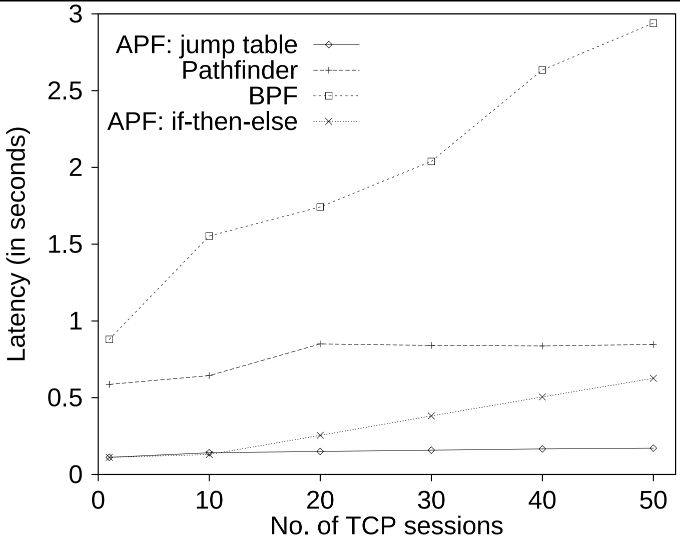

Figure 3: Experimental results: latency

All of our tests were done on a 100Mhz SPARC workstation. We employed the same testing code as had been used with Pathfinder, and used this to compare the performance of APF with that of Pathfinder [1] and BPF [9] (due to arcane implementation reasons, we were unable to run these tests on MPF [10]; however, it has been shown elsewhere that Pathfinder is faster than MPF [1]).

Our latency test measured the time to identify TCP packets destined for a particular session given an increasing number of active sessions. We timed how long it took to run each classification algorithm one million times, while testing with 10, 20, 30, 40, and 50 sessions, and divided that time by one million to yield the latency. Note that we isolate our measurements from the overhead incurred by the rest of the software (such as the device and protocol drivers): our numbers only show the time spent in the filter engine. The results are shown in Figure 3: two performance curves are given for APF, corresponding to two low-level implementation alternatives tested, where state transitions were implemented using conditionals and jump tables respectively. These data show APF to be the fastest of the three classification engines, with a speed that is approximately five times as great as that of Pathfinder.

Our scalability test measured the time to classify a packet as the size of the protocol stack is increased from 2 up to 6. Each layer appended to the stack effectively requires two additional comparisons per packet. This test shows that APF can be as much as six times faster then Pathfinder. Furthermore, note that the rate of increase in the classification overhead as the protocol stack lengthens

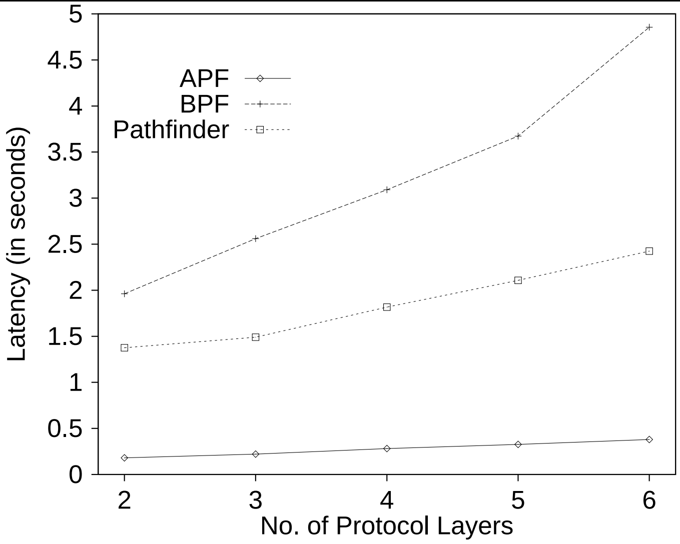

Figure 4: Experimental results: scalability

is smallest for APF.

The main reason why APF is so much faster than Pathfinder or BPF is that it compiles the automata to native code, while the other systems either interpret a representation of the automata [1] or use interpreted virtual machine instructions [9, 10]. This illustrates one of the advantages of using a declarative specification language that allows the specification of packet classifiers to be separated from their implementation, and thereby makes available a range of possible alternative implementation strategies.

Our prototype implementation currently translates packet classifier specifications into C programs that are then compiled using the C compiler. Because of this, the *insertion time*—that is, the time taken to produce an executable packet classifier from a specification—is fairly high. This issue can be addressed, for example, by using incremental compilation to native code [2, 5, 6, 7]. We intend to incorporate this into a future version of the system.

5 Conclusion

Fast classification of packets is fundamental in any networked system. Previous proposals for languages for specifying packet classifiers tended to be non-modular, low-level, and implementation-oriented, with adverse effects on readability, maintainability, and performance. This paper introduces a declarative language for the modular specification of network packet classifiers. It makes for enhanced readability and maintainability, and can compile into very fast code that is measurably superior to previous implementations.

References

[1] M. L. Baily, B. Gopal, L. L. Peterson, "Pathfinder: A pattern-based packet classifier", in *Proc. First Symposium on Operating System Design and Implementation*, November 1994, pp. 241–256.

[2] C. Chambers, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Ph.D. Dissertation, Stanford University, Stanford, CA, April 1992.

[3] D. Clark, "An Analysis of TCP processing overhead", in *IEEE Communications*, June 1989, 27(6):23-29 .

[4] R. K. Cytron and M. Jayaram, "Efficient Demultiplexing of Network Packets by Automatic Parsing", *Proc. Workshop on Compiler Support for Systems Software*, August 1996.

[5] L. P. Deutsch and A. Schiffman, "Efficient Implementation of the Smalltalk-80 System", *Proc. 11th ACM Symposium on Principles of Programming Languages*, Jan. 1984, pp. 297–302.

[6] D. R. Engler, "VCODE: a Retargetable, Extensible, Very Fast Dynamic Code Generation System", *Proc. SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996.

[7] D. R. Engler, *DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation*, Technical Report MIT/LCS/TM533, May 1996.

[8] S. C. Johnson, "Yacc – yet another compiler compiler", Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[9] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture", *Proc. Winter 1993 USENIX conference*, pp. 259-269, Jan. 1993.

[10] M. Yuhara, B. N. Bershad, C. Maeda and J. E. B. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages", *Proc. Winter 1994 USENIX Conference*, Jan. 1994.