Report Number: WUCS-95-21

1995-01-01

Efficient Demultiplexing of Network Packets by Automatic Parsing

Mahesh Jayaram and Ron K. Cytron

Packet filters are a mechanism for efficiently demultiplexing network packets to application endpoints. There is currently no general, formal specification method for packet filters that allows for easy or efficient composition of specifications. In this paper we present an automatic approach that achieves all of these goals. We approach packet filter specification as a language recognition problem: each filter is represented by a context-free grammar, whose language is the set of packets the filter should accept. Thus, packet filters can be formulated through a general, well defined specification; further, the grammar-based approach simplifies filter composition, which is essential where... **Read complete abstract on page 2.**

Efficient Demultiplexing of Network Packets by Automatic Parsing

Mahesh Jayaram and Ron K. Cytron

Complete Abstract:

Packet filters are a mechanism for efficiently demultiplexing network packets to application endpoints. There is currently no general, formal specification method for packet filters that allows for easy or efficient composition of specifications. In this paper we present an automatic approach that achieves all of these goals. We approach packet filter specification as a language recognition problem: each filter is represented by a context-free grammar, whose language is the set of packets the filter should accept. Thus, packet filters can be formulated through a general, well defined specification; further, the grammar-based approach simplifies filter composition, which is essential where scalability is important. However packet filters based on standard LR parsing techniques suffer from poor performance they touch every portion of the input, they check input bit by bit, they occupy large amount of space. We present new optimizations to LR parsing that enable our automatic approach to overcome the above problems and achieve performance rivalling hand-crafted approaches. We present results that compare our approach to the BSD packet filter for TCP connections; our approach shows significant improvements when there are multiple filters installed: for 50 TCP connections our approach is 6 times faster.

Efficient Demultiplexing of Network Packets by Automatic Parsing

Mahesh Jayaram
Ron K. Cytron

WUCS-95-21

July 19, 1995

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

This work supported by the National Science Foundation Grant NCR-9405444

Efficient Demultiplexing of Network Packets by Automatic Parsing

Mahesh Jayaram Ron K. Cytron

Abstract

Packet filters are a mechanism for efficiently demultiplexing network packets to application endpoints. There is currently no general, formal specification method for packet filters that allows for easy or efficient composition of specifications. In this paper we present an automatic approach that achieves all of these goals. We approach packet filter specification as a language recognition problem: each filter is represented by a context-free grammar, whose language is the set of packets the filter should accept. Thus, packet filters can be formulated through a general, well defined specification; further, the grammar-based approach simplifies filter composition, which is essential where scalability is important. However packet filters based on standard LR parsing techniques suffer from poor performance: they touch every portion of the input, they check input bit by bit, they occupy large amount of space. We present new optimizations to LR parsing that enable our automatic approach to overcome the above problems and achieve performance rivalling hand-crafted approaches. We present results that compare our approach to the BSD packet filter for TCP connections; our approach shows significant improvement when there are multiple filters installed: for 50 TCP connections our approach is 6 times faster.

1. Introduction and Motivation

Consider a computer that receives a variety of packets from a network. The computer is host to many applications (consumers), each of which is eligible to receive and process network packets. The kernel must decide, on a packet-by-packet basis, which application(s) should receive each packet; there are several conceptual schemes for accomplishing this:

Kernel level: When network protocols are implemented in the kernel, data is typically demultiplexed to the user endpoints by processing packets through the protocol layers in the kernel, as shown in Figure 1(Left). Each layer of the protocol stack looks at its corresponding header and demultiplexes the packet to the next higher layer of the stack, till finally the user process receives the packet. Such kernel demultiplexing is

efficient, requiring minimal context switching and system calls per packet. However this approach is not flexible, requiring the complete protocol stack to be installed directly in the kernel. As kernel code is difficult to develop, debug, and maintain, this approach can be burdensome on system and security administration. Moreover, this approach does not support fine-grained filtering of packets.

User level: In this approach, each packet is effectively demultiplexed by some user process, as shown in Figure 1(Middle). This approach allows user processes great flexibility in selecting packets and can support fine-grained filtering. It is also easier to implement, requiring little modification to the kernel. However performance suffers due to the overhead of context switches and system calls: with each received packet, the system must switch into the demultiplexing process and then switch again when the intermediate process transfers the packet to the final recipient process. As this is expensive, it would be more efficient to transfer the packet directly to the final destination process from the kernel. However this requires the kernel to determine the final endpoint of each received packet.

Packet filter:. To overcome the disadvantages of the above two approaches, the idea of *packet filtering* was developed [MRA87], as depicted in Figure 1(Right). A packet filter mechanism is essentially a kernel agent close to the network device that checks incoming packets and sends them to the appropriate user endpoints. Each endpoint provides a specification of the packets it wants, giving just enough enough information to be able to identify the desired packets. The filtering mechanism demultiplexes all packets that satisfy a particular specification to the associated endpoints. The specification is protocol independent, thus allowing the filter mechanism to demultiplex packets belonging to any protocol. Packet processing is left to the endpoint. The packet filter thus combines the advantages of kernel- and user-level demultiplexing.

Packet filters find use in many applications:

Network monitoring tools are used to capture and display packets exchanged between different hosts in a network. They help developers and maintainers of software to detect and solve network problems. They can also be used to monitor performance of communication software. To build such tools, the kernel must provide user programs access to unprocessed network traffic efficiently. Tcpdump [JLM89], Arpwatch [Ler92], and Sun's NIT [nit90] are examples of tools that use a packet filtering mechanism.

User level protocols: As has already been mentioned, it is difficult to implement protocols in the kernel, as kernel resident code is typically difficult to write, maintain, and debug. Thus it is desirable to implement protocols in user space. Not only are user-level implementations flexible, they help to decouple protocol code from the details of the kernel, leading to greater portability. Moreover, implementing protocols in the user space hides protocol-specific details from the kernel, greatly reducing its complexity. However, the performance of user-level protocols depends on how efficiently packets are demultiplexed to the user space. Packet filters help in producing efficient user-level protocols. In the Mach operating system [ABB+86], MPF supports protocol

processing in the user endpoint. The user process contains the whole protocol stack, and is responsible for the entire protocol processing. MPF demultiplexes incoming packets to the appropriate endpoint.

High-performance event filters: One application of packet filters which is as yet unexplored is as high performance event filters for dynamic multipoint(DMP) applications [Sch94]. Such applications are abundant, spanning diverse domains including satellite communication systems, network management systems, real-time market data analysis systems, on-line news services, distributed systems, and active databases [isi94, Bra88, nit90, MRA87, GJS94]. These applications display many characteristics which call for solutions involving packet filters: a high volume of messages; signalling events generated in real-time by suppliers; complex message formats containing a number of header fields; a number of consumers, each interested in a subset of the generated messages; and dynamic addition and deletion of consumers. The traditional applications of packet filters form a subset of the larger domain of DMP applications. It remains to be seen in what way packet filters can be modified to meet the demands of DMP applications.

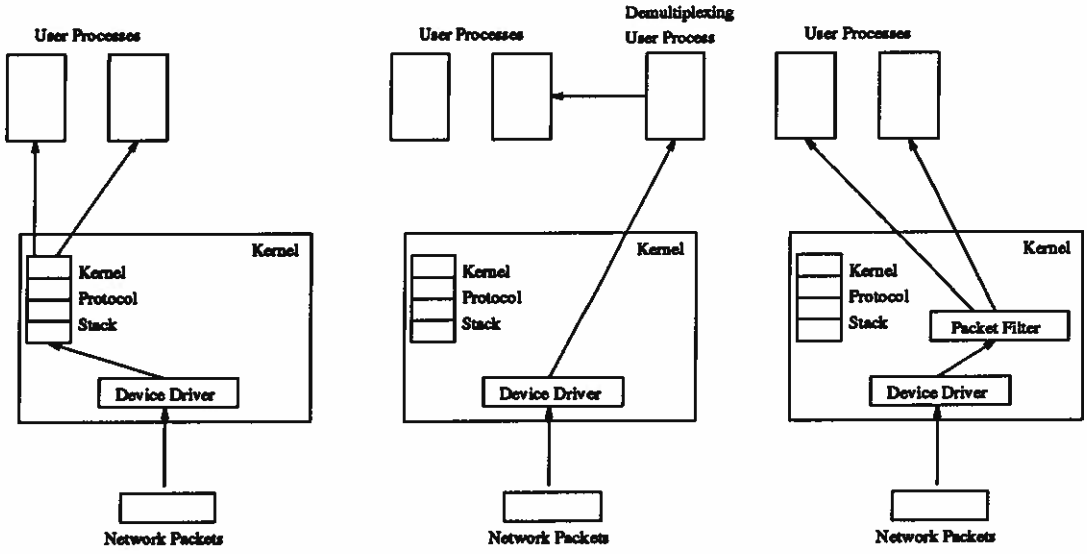

Figure 1. Demultiplexing of network packets to user processes.
Left: Demultiplexing in kernel-resident protocol code; Middle: Demultiplexing in a user-level process; Right: Demultiplexing in a kernel-resident packet filter.

Depending on the specification, filtering mechanisms are of two kinds:

- The specification can take the form of code in a filter language, which is interpreted by the filtering mechanism to test whether the packet is to be accepted [MRA87, MJ93, YBMM94]. The code could be compiled and executed for greater efficiency, if the security of executing the packet filter were not a concern.

- Alternately the specification can be in the form of a pattern, or a set of rules [BGS+94]. The filter mechanism then checks whether a packet conforms to the pattern. This is the approach followed in this paper.

There are a number of properties desirable in any packet filtering mechanism:

1. It must be protocol independent and able to support filters for different protocols.
2. The approach should be safe, avoiding undesirable interaction of the packet filter with other kernel or application components.
3. The filter specification language should be general and sufficiently expressive to specify different kind of filters. For example, the language should accommodate packet headers with variable length fields.
4. The filtering mechanism must be able to deal with packet fragmentation.
5. It should be scalable.
6. It must be efficiently implemented.
7. Incorporation of new specifications and removal of specifications should be efficient.

In this paper we address all the above properties, except the last: efficiency of filter specification installation is less critical than filtering efficiency, as the frequency of arrival of packets is expected to be much larger than changes to the installed filters. In Section 6 we discuss an approach allowing for efficient filter installation and removal.

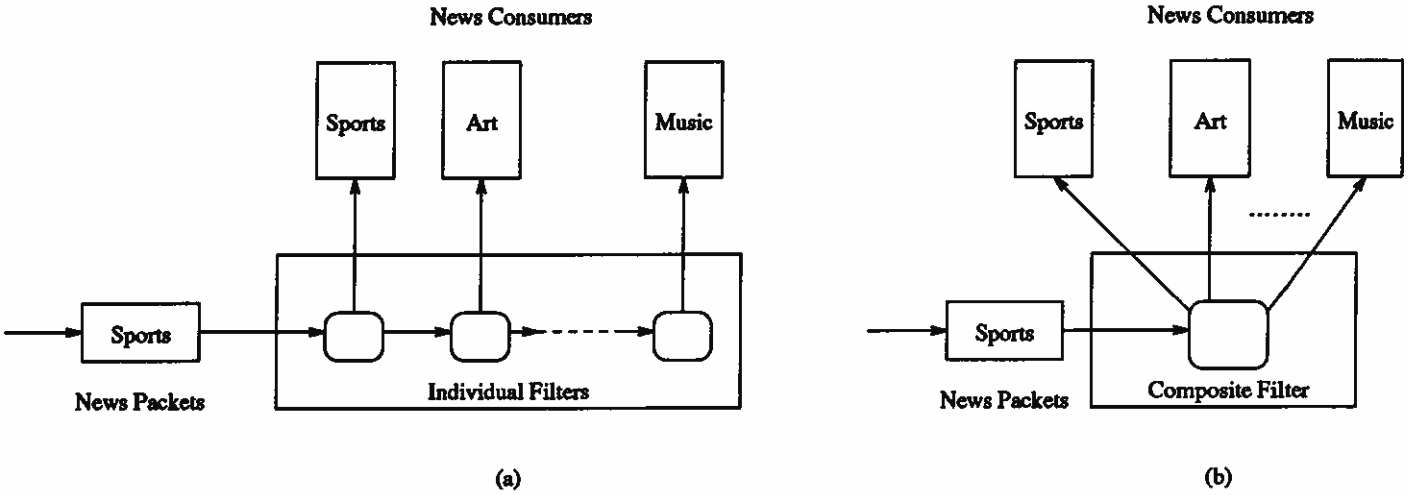

Figure 2. Demultiplexing news packets to consumers. (a) Separate filter per endpoint. (b) Composite filter.

Of the above properties, scalability is especially important when there are a large number of recipient endpoints: as the number of host consumer applications grows, the burden of

packet filtering increases. Consider a number of consumers connected to a news server as shown in Figure 2. A continuous stream of news packets arrives at the news server. Each consumer's interest in the incoming data is captured by an individual filter specification, which selects the data for that corresponding consumer. In the naive approach, the filter mechanism would test each incoming packet with every individual specification to find out which endpoints should receive the packet. Thus, the cost of demultiplexing inputs in this case grows linearly in the number of endpoints. It is not unusual for a thousand consumers to be connected to a news server; in this case the cost of demultiplexing becomes unacceptable. To improve scalability it is desirable to be able to combine individual specifications into a *composite specification.* This would enable the filter mechanism to determine which applications should receive the packet in just a single pass over a packet.

Existing packet filter mechanisms CSPF [MRA87] and BPF [MJ93] are interpreter-based filter mechanisms that do not support composition. MPF [YBMM94] is similar to BPF but adds support for composing filter specifications for special cases and for dealing with fragmented packets. PathFinder [BGS+94] is a pattern-based filtering mechanism that allows a greater degree of composition than MPF and allows for an efficient implementation. We discuss the above filter mechanisms in greater detail in Section 5.

A common deficiency in all the above filtering mechanisms is that they lack a formal approach and deal with different requirements like variable length fields and composition in an *ad hoc* way.

Our approach accommodates well defined, general packet filter specifications and also allows for easy composition. Our method derives its generality from the filter specification language of context-free grammar.The generality of the approach offers us two advantages:

- A powerful specification language allowing the description of a variety of filters and packet formats.
- A unified way of dealing with different requirements of a filtering mechanism.

Moreover, by optimizing the basic mechanism we retain efficiency without compromising on generality. These optimizations, described in Section 3, are new and apply to LR parsing in general and so are possibly of independent interest. Our experiments show that when multiple filters are installed our mechanism works much better than BPF. With 50 endpoints our mechanism is about 6 times faster.

The rest of the paper is organized as follows. We discuss our approach in Section 2. In Section 3 we describe the optimizations performed on the basic recognizer to yield an efficient implementation. Section 4 contains the performance analysis of the filtering mechanism. Section 6 discusses future work on further improving our mechanism. Section 5 describes related work in packet filtering as well as LR parsing. In Section 7 we present our conclusions.

2. Filtering Mechanism

At the heart of our filtering mechanism is the filter specification language, which is the set of LR grammars.[1] LR grammars describe deterministic context-free languages; like regular expressions, they formalize the notion of parsing. They are widely used, notably in defining and translating programming languages, as well as in other string-processing applications. LR grammars are more general than regular expressions and can describe arithmetic expressions with arbitrary nesting of parantheses and block structure in programming languages. In our filtering mechanism, LR grammars are used to specify packet filters. Figure 3 shows the basic filtering mechanism. The pattern to be matched in the packets is encoded in the form of an LR grammar, so that the desired packets form the language recognized by the grammar. A tool such as yacc [Joh79] then generates an LR parse table for the grammar, assuming the grammar presents no conflicts. As discussed in Section 3, we optimize the tables produced by yacc, in a manner quite different from optimizations typically seen in the context of parsing programming languages [Pen86, HW90]. The tables are in fact optimized for a modified LR parsing engine, which has the ability to *skip* tokens. It is this engine (driven by the optimized LR table) that acts as a packet filter, accepting packets conforming to the grammar specification.

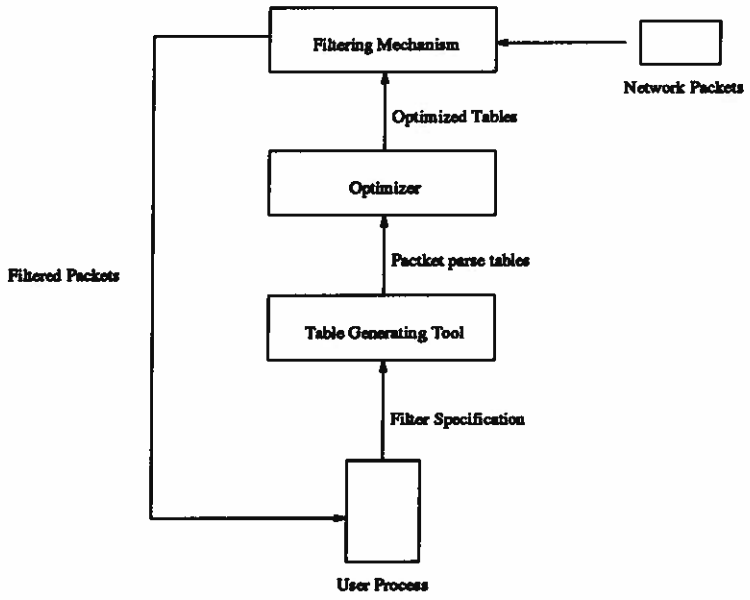

Figure 3. Basic filtering mechanism.

Figure 4 shows an example specification of a packet filter for a TCP connection. The grammar consists of a number of rules. Each rule contains a left-hand-side consisting of single non-terminal and a right-hand-side consisting of terminals and non-terminals. The grammar contains only two terminals (bits 0 and 1). The grammar is used by a yacc-

[1] Actually, because we use yacc, our grammars are really LALR(1); however, most LR(1) grammars of interest are also LALR(1), so there is not much point in differentiating the two in this paper: we use the shorter "LR" appellation.

```
TCPConnHeader : EtherType IPHeader TCPPortPair
EtherType :  #IP_TYPE
IPHeader  : Vers HlenPlusRest
Vers      : HalfByte
HlenPlusRest : 0 1 0 1 FixedRest
             | 0 1 1 0 FixedRest OneIPOption
             | 0 1 1 1 FixedRest TwoIPOption
             | 1 0 0 0 FixedRest ThreeIPOption
             | 1 0 0 1 FixedRest FourIPOption
             | 1 0 1 0 FixedRest FiveIPOption
             | 1 0 1 1 FixedRest FiveIPOption OneIPOption
             | 1 1 0 0 FixedRest FiveIPOption TwoIPOption
             | 1 1 0 1 FixedRest FiveIPOption ThreeIPOption
             | 1 1 1 0 FixedRest FiveIPOption FourIPOption
             | 1 1 1 1 FixedRest FiveIPOption FiveIPOption
FixedRest : ServiceType TotalLength Identification Flags \
            FragmentOffset TimeToLive Protocol HeaderChecksum IPAddrPair
ServiceType : Byte
TotalLength : TwoByte
Identification : TwoByte
Flags : bit bit bit
FragmentOffset : bit Byte HalfByte
TimeToLive : Byte
Protocol : #TCP_PROTOCOL
HeaderChecksum : TwoByte
IPAddrPair : #IP_SRC_DST_PAIR
FiveIPOption : ThreeIPOption TwoIPOption
FourIPOption : TwoIPOption TwoIPOption
ThreeIPOption : TwoIPOption OneIPOption
TwoIPOption : OneIPOption OneIPOption
OneIPOption : Option Padding
Option : ThreeByte
Padding : Byte
TCPPortPair : #TCP_PORT_PAIR
FourByte : TwoByte TwoByte
ThreeByte : TwoByte Byte
TwoByte : Byte Byte
Byte : HalfByte HalfByte
HalfByte : bit bit bit bit
bit : 0
    | 1
```

Figure 4. TCP connection filter specification.

like parser generating tool to form standard LR parse tables which are then optimized (as discussed in Section 3) for use in a modified LR parser.

We now discuss how our filtering mechanism satisfies the desired properties described in Section 1.

Protocol independence: As our filtering mechanism views packets simply as a string of bits, it is completely protocol independent. The packets of any protocol can be specified by describing the bit pattern corresponding to that protocol in the form of a context-free grammar.[2]

Safety: As each filter is specified as an LR grammar, the operation of an associated parser for such filters can be proven safe, if stack over- and under-flow can be prevented. In systems with virtual memory hardware, the cost of maintaining safety is minimal [App91], especially where safety is not violated.

Specification generality: The specification language, LR grammars, is powerful enough to allow easy specification of different kinds of filters. For example, no special treatment is needed for packets with varying length fields.

Variable length headers: Most packet-filtering methods handle fixed length headers with ease, but not all protocols have fixed length headers. It is therefore necessary for packet filtering mechanisms to handle variable length fields in protocol headers to parse packets of such protocols. In the example in Figure 4, the IP options field is of variable length depending on the number of options. Typically such fields are preceded by a length field that is used to detect the length of the field in a particular packet. The four bit header length field in the IP header serves that purpose. As shown in the example, it is simple to express such varying length fields in grammar form. Thus the power of the specification language allows us to deal with varying length fields without any special arrangement.

Fragmentation: Fragmentation occurs whenever the size of a packet to be transmitted is larger than the maximum transmission unit of the network. For example, UDP packets, which can be larger than the maximum Ethernet message size, may have to be sent as a number of IP fragments which have to be reassembled at the receiving end.

Demultiplexing fragmented messages is complicated. Firstly, full information to dispatch the packet to the final endpoint which is contained in the higher level protocol header is present only in the first fragment. Usually, a unique message id links different fragments of the same packet. Secondly, fragments may arrive out of order.

For the purpose of this paper we deal with fragmentation by diverting fragmented packets to an intermediate endpoint. This intermediate endpoint could then forward the fragments or the complete packet to the final endpoint, when it becomes known. Fragmented packets are typically detected by some flags set in the header. In the IP header, the Flags and Fragment Offset fields are used to detect fragmentation. The above approach of dealing with fragmentation is the simplest, and may involve some overhead for fragmented packets; however, with some modifications, other methods [YBMM94, BGS+94] can be incorporated in our filtering mechanism.

[2]One might be concerned that a packet specifications beyond context-free might be necessary. Because packets are finite, any packet filter is actually regular in theory; however, we find the context-free notation much more economical.

Composition: It is easy to give composite specifications using grammars. Grammars provide an *or* operator that can be used to match either of two different patterns. Thus if

$$S \longrightarrow A$$

matches pattern A, then

$$S \longrightarrow A \mid B$$

matches pattern A *or* B. By associating each endpoint with the rule matching its pattern we demultiplex packets to the appropriate endpoint depending on the rule which was used to parse the packet. The composite specification for two TCP connections is shown below:

```
TCPPortPair : #TCP_PORT_PAIR_1
            | #TCP_PORT_PAIR_2
```

3. Optimization

Though our approach is more general, it must perform well enough to be of practical use. While a filter could be realized by a standard LR parser which recognizes the filter specification grammar, it does not have the requisite performance. There are many reasons for its inefficiency:

- Usually only a small portion of the header of a packet is relevant to a filter. However, a standard LR parser traditionally "touches" every input, symbol whether it is relevant to the filter or not. This is because standard LR parsers sequentially scan through their input and cannot skip past portions of the input that are effectively of no interest. In contrast, hand-crafted packet filters typically touch only the fields required for rendering a decision on packet selection.
- A standard LR parser would match fields of interest one bit at a time. Hand-crafted filters would match values byte-wise or word-wise.
- Standard (LALR) parse tables for the specification grammar are large when compared with hand-crafted approaches.

In view of the above deficiencies, it is necessary to optimize the standard LR parser to overcome the above shortcomings. In the following subsections we discuss the optimizations performed on the standard parser to make it effective for packet filtering.

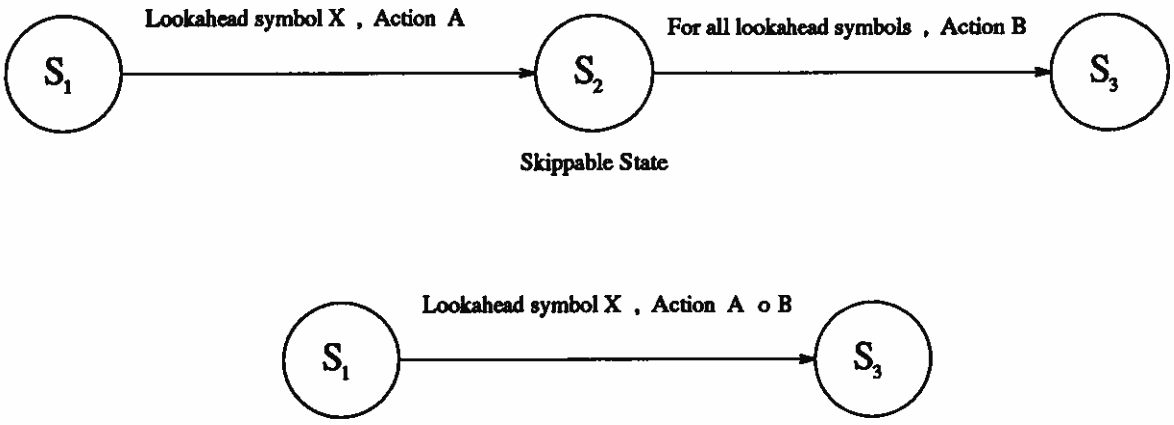

Figure 5. Skipping states.

3.1. Skipping LR parser

A standard LR parser can be viewed as a state machine equipped with a state stack. The parser selects the action to be performed from the parse table, given the current state and the current input symbol being examined. Actions, which may involve advancing to the next input symbol and/or manipulating the contents of stack, determine state transitions. A look at the parse tables generated for the filter specification grammars reveals that portions of the packet that are irrelevant for filtering, show up in what we call *skippable* states. These states are skippable in the sense that the action to be performed in those states is independent of the next input symbol. Thus, by chaining together (i.e., composing) actions of skippable states, it is possible to avoid examining portions of the input which are irrelevant for the parsing process. Skipping past irrelevant states also helps in reducing redundant stack activity performed in those states, further improving performance. Figure 5 illustrates the optimization. We now give details of this optimization.

We now describe the algorithm to construct a skipping LR parse table from a standard parse table. We also present a comparison of the skipping parser with the standard parser for the example specification of a TCP Connection in Figure 4. The comparison is on the basis of the number of accesses to an input string following the specification.

3.1.1. Notation and Terminology. The notation used in the rest of the section is as follows:
$s \in S$ the set of parser states
$a \in \Sigma$ the set of terminals
$X \in V$ the set of non-terminals
$x \in \Sigma \cup V$
$I \in \Sigma^*$
$\alpha \in S^*$
$\beta \in V^*$
$\gamma, w \in (\Sigma \cup V)^*$

A standard (canonical) $LR(k)$ parser scans its input $I\$$ from left-to-right, where \$ is a special end-of-string token and $I \in \Sigma^*$ is the string to be parsed. It has been shown in (Mickunas reference) that any $LR(k)$ language is also $LR(1)$. So, without any loss of generality, we will consider only $LR(1)$ languages in the rest of the paper.
One of the following actions is performed by an $LR(1)$ parser at each step:

shift s: State s is pushed onto the stack and the input pointer is advanced by one token.

reduce $X \rightarrow \gamma$: A number of states $\mid \gamma \mid$ is popped from the stack; a new state is pushed onto the stack. This can also be viewed as prepending X to the input string, so that it is the next symbol read from the input. Thus the next action is to shift the next state on to the stack depending on the top-of-stack and X, just like a shift on a terminal. This alternative allows a uniform treatment of terminals and nonterminals, and used in Incr[] and parallel parsing[] (References).

error: The input string is not accepted.

accept: The input string is accepted.

The parser uses lookahead to decide the appropriate action at parse-time. We denote the actions encoded in the parse table P as $action_P(s, x)$, where s is the state at the top of the stack and x is the current lookahead symbol. We are interested only in *recognizing* strings, not performing actions based on derivations. We now redefine each of the "macro" actions of the traditional parser into a sequence of "micro" instructions in a *skipping* LR parser. The microinstructions are as follows:

AdvanceInput n: The input pointer is advanced n tokens. Lookahead is adjusted appropriately.

Pop k: k states are popped.

Push α: The sequence of states $\alpha = s_1 s_2 \ldots s_m$ is pushed, with s_m on top-of-stack.

Prepend β: The string of nonterminals β is prepended to the input string.

Accept: same as standard LR.

Error: same as standard LR.

Henceforth, we encode actions in the skipping parser other than Accept or Error as a 4-tuple *compound microinstruction* (n, k, α, β) which denotes the sequence of microinstructions:
AdvanceInput n; Pop k; Push α; Prepend β
It is clear that any microinstruction can be represented by a compound microinstruction. We show later that any sequence of compound microinstructions can also be represented by one such compound microinstruction which is the *composition* of the actions of the sequence (Subsection 3.1.2). Traditional LR parsing macroinstructions can be represented by a compound microinstruction as follows:

shift s: AdvanceInput 1; Pop 0; Push s; Prepend ϵ, where ϵ denotes the empty string.

reduce $X \rightarrow \gamma$: AdvanceInput 0; Pop $\mid \gamma \mid$; Push λ; Prepend X, where λ denotes the empty sequence of states.

In this manner, it is clear from construction that a skipping LR parser can simulate a traditional LR parser. It also follows that any sequence of shift or reduce actions can also be captured by a compound microinstruction which is the composition of the shift or reduce actions in the above form.

If $\alpha = s_1 s_2 \ldots s_m$ is the sequence of states on stack with s_m on top-of-stack and $w = x_1 x_2 \ldots x_n$, $x_i \in (\Sigma \cup V)$ is the unexpended input with x_1 as the next lookahead, then we define the *configuration* C as the pair (α, w). The action applied at the above configuration by a parser with parse-table P is $action_P(s_m, x_1)$ and if the next configuration after applying the action is $C' = (\alpha', w')$ then we denote this by $(\alpha, w) \vdash_P (\alpha', w')$. There is also a special error configuration C_{error}. $(\alpha, w) \vdash_P C_{error}$ if either :

$action_P(s_m, x_1) =$ Error

$action_P(s_m, x_1) = (n, k, \alpha'', \beta'')$ and $n \geq \mid w \mid$ or $k > \mid \alpha \mid$

We denote the *parse* of string I by a parser with parse-table P by the following sequence of configurations :

$$(s_0, I\$) \vdash_P (\alpha_1, w_1) \vdash_P (\alpha_2, w_2) \vdash_P \ldots \vdash_P C_{last}$$

where s_0 is a distinguished start state and C_{last} is either C_{error} in which case the string is not accepted or $(f, \$)$, where f is a distinguished final state, in which case the string is accepted. Note that in parse-table P only $action_P(f, \$) =$ Accept.

In the rest of the paper we use parser and parse-table interchangeably and we drop the subscript from an *action* when it is clear from the context.

3.1.2. Composition. We now formally define the composition of two compound microinstructions $A \circ A'$, where A happens before A'.
If $A = (n, k, \alpha, \beta)$ and $A' = (n', k', \alpha', \beta')$
where $\alpha = s_1 s_2 \ldots s_{|\alpha|}$ and $\beta = X_1 X_2 \ldots X_{|\beta|}$
$A \circ A' =$

Case 1: if $\mid \beta \mid \leq n', \mid \alpha \mid \leq k'$
$(n + n' - \mid \beta \mid, k + k' - \mid \alpha \mid, \alpha', \beta')$

Case 2: if $\mid \beta \mid \leq n', \mid \alpha \mid > k'$
$(n + n' - \mid \beta \mid, k, s_1 s_2 \ldots s_{|\alpha| - k'} \alpha', \beta')$

Case 3: if $\mid \beta \mid > n', \mid \alpha \mid \leq k'$
$(n, k + k' - \mid \alpha \mid, \alpha', \beta' X_{n'+1} \ldots X_{|\beta|})$

Case 4: if $|\beta| > n', |\alpha| > k'$
$(n, k, s_1 s_2 \ldots s_{|\alpha|-k'} \alpha', \beta' X_{n'+1} \ldots X_{|\beta|})$

Case 5: if $A' =$ Error then $A \circ A' =$ Error

Case 6: if $A' =$ Accept then $A \circ A' =$ Accept

Case 7: if $A =$ Error then $A \circ A' =$ Error

Case 8: if $A =$ Accept then $A \circ A' =$ Accept

3.1.3. Algorithm. The algorithm for creating the skipping LR parsing table is described in Figure 6. The input for the algorithm is an $LR(1)$ table in which entry $action(s, x)$ refers to the action applied at state s with lookahead x. The goal is to reduce the number of actions in the parse of an input string by composing together as many actions of the table as possible always maintaining that the modified table accepts a string iff the original table does. Intuitively, our algorithm iteratively analyzes a traditional LR parsing table for opportunities to perform *safe* composition of actions of the table. Any configuration in the parse of an input string by a table modified by a safe composition is also a configuration in the parse of the input string by the unmodified table.
A composition of actions $A \circ A'$ is *safe* if either :

The state the parser will enter after performing action A, *nextstate*, is known, the next lookahead symbol, *nextsymbol*, is known and $action(nextstate, nextsymbol) = A'$

The state the parser will enter after performing action A, *nextstate*, is known and *nextstate* is *skippable.*

A state s is *skippable* if the choice of action in state s can be made irrespective of the next lookahead.
More formally, State s is skippable if either:

$\forall a \in \Sigma$, $action(s, a)$ is identical.

$\forall a \in \Sigma - \{\$\}$, $action(s, a)$ is identical and is of the form (n, k, α, β) where $n > 0$, and $action(s, \$) =$ Error.

3.1.4. Comparison of standard LR and skipping LR. The complexity of parsing a string I by a table can be thought as the number of configurations (or actions) in the parse of I by the table. Taking the specification of fixed length packets we can see that the sequence of actions performed in parsing any particular "don't care" portion of the packet, starting from the last "significant" token before the "don't care" portion corresponds to a sequence satisfying the properties of optimality. So the optimized table parses the "don't care" portion in one shot with the action for parsing the last "significant" token before the "don't care" portion. So the complexity of parsing a packet by the optimized table is

Algorithm[1] Skipping LR
Input: An unoptimized parse table *action*
Output: A Skipping LR parser
Method:

```
TableModified ← true
/*          Iterate until there is no change to the table          */
while (TableModified) do                                            ⇐ [1]
   TableModified ← false
   /*          Iterate through all table entries                   */
   foreach ( s ∈ S) do                                              ⇐ [2]
      foreach (x ∈ Σ ∪ V) do
         FinishedWithEntry ← false
         /*          Compose entry as much as possible             */
         while (not FinishedWithEntry) do                           ⇐ [3]
            nextst ← nextstate(s, action(s, x))
            nextsym ← nextsymbol(action(s, x))
            if (not nextst = Unknown and not nextsym = Unknown) then
               action(s, x) ← action(s, x) ∘ action(nextst, nextsym)   ⇐ [4]
               TableModified ← true
            fi
            elseif (not nextst = Unknown and skippable(nextst)) then
               action(s, x) ← action(s, x) ∘ action(nextst, a), where a ∈ Σ   ⇐ [5]
               TableModified ← true
            else
               FinishedWithEntry ← true
            fi
         od
      od
   od
od
end
```

Figure 6. Main algorithm.

independent of the portion of the packet which is "don't care". The same can be seen for the specification of variable length packets.

From the filter specification given in Figure 4 we can see that only a fixed portion of the fixed length part of the packet header is useful for filtering. These are the EtherType, Hlen, Protocol, IPAddrPair, TCPPortPair fields and 1 bit of the Flags field (a total of 125 bits). This is the portion of the header relevant to filtering irrespective of the total header length. The rest is "don't care" as far as filtering is concerned. The skipping parser attempts to find

```
Function skippable(State) : Skippable
    if (∀a ∈ Σ, action(State, a) is identical) then                    ⇐ [6]
        Skippable ← true
    fi
    elseif
    (∀a ∈ Σ − {$}, action(State, a) is identical and action(State, a) = (n, k, α, β), n > 0)
    then
        Skippable ← true
    else
        Skippable ← false
    fi
    return (Skippable)
end
```

Figure 7. Function skippable.

```
Function nextstate(CurrentState, Action) : NextState
    if (Action = (n, k, αs, β)) then                                   ⇐ [7]
        NextState ← s
    fi
    elseif (Action = (n, 0, λ, β)) then                                ⇐ [8]
        NextState ← CurrentState
    else
        NextState ← Unknown
    fi
    return (NextState)
end
```

Figure 8. Function nextstate.

```
Function nextsymbol(Action) : NextSymbol
    if (Action = (n, k, α, Xβ)) then                                   ⇐ [9]
        NextSymbol ← X
    else
        NextSymbol ← Unknown
    fi
    return (NextState)
end
```

Figure 9. Function nextsymbol.

such relevant portions of the string and access only those portions whereas the traditional parser has to look at the full input to render a decision.

The tables in Figure 10 below give the comparison between a traditional LR parser and a skipping LR parser for the above specification in terms of the number of accesses to a string before giving decision of whether to accept or not.

Valid Packet with no Options Length : 208 bits	
Parser	*String accesses before accept*
Traditional	208
Skipping	125

Valid Packet with 10 Options Length : 528 bits	
Parser	*String accesses before accept*
Traditional	528
Skipping	125

Invalid Packet with 10 Options Position of first invalid bit : 502	
Parser	*String accesses before reject*
Traditional	502
Skipping	99

Figure 10. Comparison between a standard parser and a skipping parser.

From these tables one can see that there is a big advantage in using the skipping parser over the traditional one.

3.2. Matching multiple bits

Packet fields do not necessarily correspond to a given computer's notion of word-length:

1. Packets are potentially sent between different makes and models of computers; it would be unfair to insist that a packet conform to one manufacturer's notion of a word.
2. Word alignment would waste packet space, which would increase transmission time and occupy unnecessary bandwidth.

As shown in the grammar of Figure 4, our basic token is therefore the "bit". A drawback of this approach is that each bit processed by the resulting standard LR engine causes the parser to shift state. Thus, when matching multibit fields of interest, this leads to high overhead in terms of the number of packet accesses and state stack manipulations. However, each state in the standard parse table which examines a bit in the field of interest has a

valid action only if the input symbol matches the required value. Thus a multibit field which is to be checked shows up as a sequence of states, each of which tests for a bit and reports a reject if the test fails. We therefore chain together such states into one state that tests for a string of bits, and reports a reject if the input does not match the string of bits. Thus it is possible to check for more than one bit at a time. A diagrammatic view of this optimization is shown in Figure 11. In fact, the packet filter we synthesize in this manner is biased toward the case of string match, improving its average performance since errors should be few.

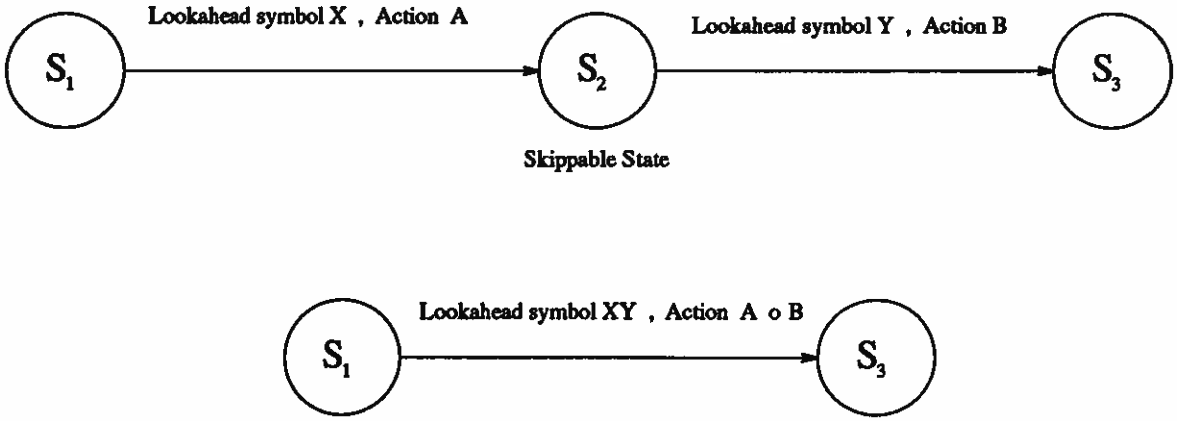

Figure 11. Matching multiple bits.

3.3. Reducing table size

Another problem of the standard LR tables is that they are large in size. The size of the tables grows roughly as the product of the number of states and the number of symbols used in the grammar. As the number of states as well as the number of symbols used in the specification grammar can be large, this is undesirable. However, it is unnecessary to keep all states and symbols in the table. It is necessary to retain only *live* states and symbols. Live states are those which are reachable during parsing and live symbols are those which are used in some live state. In a standard LR parse table, typically all states as well as symbols are live. However, after performing the optimizations to skip and to match multiple bits, a number of states become dead: when we skip past states, the skipped states become dead and when we chain together a number of states into one state to match multiple bits, only the first state in a sequence of states remains live. Moreover a number of symbols which are used only in dead states become dead as well. Thus by removing dead states and symbols it is possible to reduce the size of the parse tables significantly: we observed a 10-12 times reduction for the example specification.

4. Performance

In this section we discuss the performance of our packet filtering mechanism and compare it with BPF. In particular we are interested in the *scalability* of our approach. We measure the time to filter packets as a function of the number of recipient endpoints. The experiment was conducted on a SUN-4/300 (25MHz, Sun-4 MMU, 16 MIPS, 131072 byte cache) running NetBSD-1.0A

The experiment models the situation where there are n TCP connections between two machines and 100,000 TCP/IP packets are received by the host machine. Each of the received packets is intended for any of the recipient endpoints. The measured time reflects only the overhead to detect the endpoint(s) which are to receive the packet; this time depends only on the size of the packet header, and so it does not include the time required to dispatch the packet to the endpoint, which involves a packet copy and hence depends on the packet size.

To take into account the effect of the cache, we performed the experiment with both cold and warm caches. Cold cache measurements were made by flushing the cache before applying the filter. Warm cache readings were taken without flushing the cache. Cold and warm cache readings respectively represent the worst- and best-case measurements. Depending on the system load, and rate of packet arrival actual values should lie between these two values. The results are shown in Figure 12.

As seen in the results, the time taken by BPF to process a packet grows linearly with the number of connections, as there is a separate BPF filter for each connection. On the other hand, our filtering mechanism takes almost constant time irrespective of the number of endpoints. This is because we are able to compose all filter specifications into one. With 100 connections our mechanism performs about 10 times better.

To put the results in Figure 12 into context we give the time to copy packets in Figure 13. This gives an estimate of the time to dispatch a packet to its destination, once the destination is known. The results in Figure 12 show that our filtering mechanism takes time comparable to a packet copy. We discuss further optimizations to improve the efficiency of our mechanism in Section 6. We believe that if we incorporate these optimizations we can reduce the filtering time significantly.

In Figure 14 we show the time taken for filtering on different platforms (i.e., machine and operating system), for the case of 50 connections with warm cache.

5. Related Work

In this section we describe related work in the areas of packet filters and LR parsing . We discuss four major packet filter schemes: CSPF [MRA87], BPF [MJ93], MPF [YBMM94], PathFinder [BGS+94]. We also discuss related work in optimizing LR parsing [Pen86, HW90].

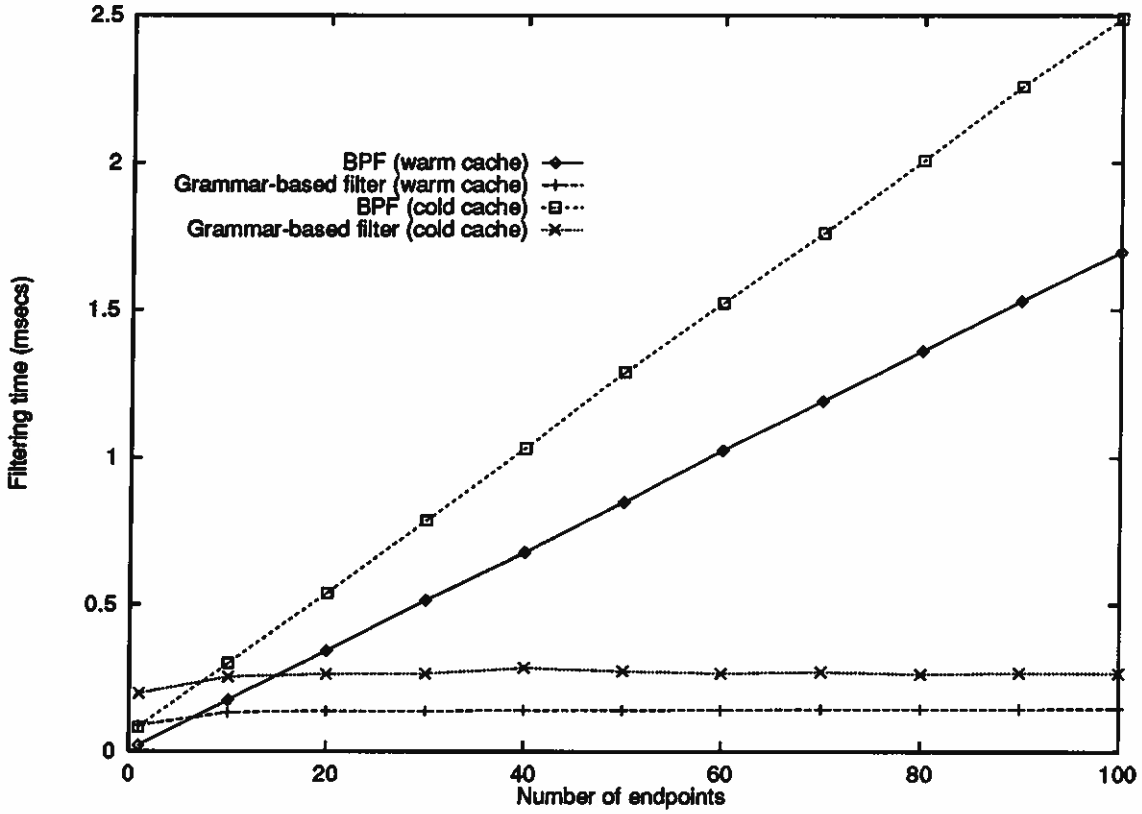

Figure 12. Comparison between BPF and our Grammar-based filter.

Packet size (bytes)	Time (msecs)
500	0.070
1000	0.125
1500	0.190

Figure 13. Time to copy packets.

Machine:	Sun 4/300	Sparc 5	Sparc 20
OS:	NetBSD	SunOS	SunOS
BPF	0.850	0.278	0.257
Grammar-based	0.140	0.044	0.042

Figure 14. Filtering time(msecs) on different platforms for 50 connections.

CSPF:. The CMU/Stanford packet filter (CSPF) is an interpreter based filtering mechanism. The filter specification language is stack based and contains binary operations

which are used to evaluate boolean expressions. The filter programs evaluate a logical expression on the fields of the header of the received packet, accepting packets based on the result. The expression to be evaluated is structured as a tree, with internal nodes performing boolean operations (OR and AND) and leaf nodes performing comparisons. CSPF has many shortcomings. The tree model of expression evaluation may involve redundant computations. The filter specification language does not contain an indirection operator, so it is restricted to deal with only fixed length fields. Moreover CSPF does not support composition of filters. The major contribution of CSPF was introducing the concept of packet filters for efficient demultiplexing.

BPF:. The BSD packet filter (BPF) addresses some of the shortcomings of CSPF and tries to remove them. BPF is also an interpreter based filter. However the specification language is register based and BPF filters are specified in an assembly like language. The language provides support for handling varying length fields. Unlike CSPF, BPF uses a directed acyclic control flow graph as the model to evaluate the filter expression. This helps BPF to avoid redundant computations. For the above reasons BPF performs better than CSPF. However like CSPF, BPF does not provide support for composing together different filters. Thus BPF does not scale well when there are a large number of endpoints.

MPF:. The Mach packet filter (MPF) adds support to BPF to handle endpoint-based protocol processing in the Mach operating system. The primary focus of MPF is on demultiplexing packets to multiple endpoints efficiently by composing filters and on handling fragmented packets efficiently. The filter specification language is same as that of BPF to which they add instructions for handling filter composition and fragmentation. Their method of filter composition is restrictive enabling them to compose filters only for the specific case where the filters share an identical prefix and followed by variation at a single point in the header. In such cases MPF associates a hash table with this field to demultiplex packets to different endpoints efficiently. Though this works fine for composing filters for different TCP Connections, this is not adequate for composing different kinds of filters. They are able to perform much better than BPF when there a large number of TCP Connections. Though MPF performs better than our filtering mechanism for TCP Connections, they lack the generality of our approach both in terms of the power of the specification language and the ability to compose in a general way.

PathFinder:. PathFinder is a pattern based packet filtering mechanism which allows for more general composition of filters with common prefixes than MPF. It also allows for an efficient software implementation and a fast hardware implementation. A filter is specified by supplying a sequence of cells. Each cell is given by a tuple consisting of : offset, length, mask and value. The filtering mechanism creates a DAG of cells. Whenever a new pattern is added the DAG is checked for the longest prefix match of identical cells already existing in the DAG. Only the remaining cells of the new pattern are added. If new cells differ only in the value from already existing cells they are coalesced using a hash table. Pathfinder performs better than MPF for TCP Connections. PathFinder also allows better

composition of specifications than MPF. However, as for MPF, both the filter specification language as well as the method of composition are less general than our filtering mechanism.

LR parsing optimizations:. There has been a lot of work in optimizing LR parsing. Some of the optimizations have been : optimization of the driver program of LR parsing [Gro88] , converting table-driven LR parsers to directly executable code [Pen86], reducing storage requirements for parsers [DDH84], making stack access optimizations [HW90]. As of now, there has been no attempt to skip past input tokens which are irrelevant to the process of parsing or matching multiple tokens simultaneously, as suggested in this paper. Though these optimizations make their greatest impact on grammars such as used for packet filter specifications they are general and are of independent interest to LR parsing.

6. Future Work

There some aspects of the filter mechanism which we can improve to make it more effective: filtering efficiency, efficiency of filter installation, and removal. To achieve this we have at our disposal the large amount of work done in LR parsing optimizations. In this section we discuss two techniques which promise to greatly improve performance.

A large amount of the time spent in parsing a packet is spent in state stack manipulation. However not all stack manipulation is unavoidable. Some states are pushed into the state stack only to be immediately popped out. In the case of TCP Connections a large percentage of pushes are redundant. This is due to the fact that after performing our optimizations, a number of states are skipped and never entered, though they may still be pushed onto the stack. [HW90] discusses a technique of avoiding such redundant activity . Significant performance improvement is reported even for grammars which do not have much redundancy. We expect that incorporating this optimization should improve performance significantly.

Another improvement to be made to the filter mechanism is to make dynamic installation and removal of filters efficient. For this we hope to profit from the work done in incremental parser generation [Hor90]. Incremental parser generating techniques address the problem of incrementally modifying parse tables when the grammar is modified. The motivation behind incremental parser generation is to be able to efficiently accomodate small changes to the grammar and avoid the computational effort involved in analyzing the full grammar. The problem of dynamically adding or removing filters is similar, involving addition or removal of rules to the specification grammar. Incorporation of incremental techniques can lead to reasonably efficient dynamic behaviour.

7. Conclusion

In this paper we have underlined the importance of packet filtering by putting it in the larger context of high performance event filtering. We examine the properties desirable in

a packet filtering mechanism and propose a general scheme for packet filtering . We also describe new optimizations to LR parsing which when applied to our scheme yields efficient filters which possess the desired properties. We give performance figures which show that our mechanism performs much better than the BSD packet filter for multiple endpoints.

Acknowledgements:. We thank: Jerome R. Cox Jr. for suggesting the importance of composition, Douglas C. Schmidt for pointing out the importance of event filtering for Dynamic Multipoint applications, Girish P. Chandranmenon, Chuck Cranor and R.Gopalakrishnan for their help in setting up the experiments. We also thank George Varghese for supporting the first author.

References

[ABB+86] M. J. Accetta, R. V. Baron, W. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian. Jr, and M. W. Young. Mach: A new kernel foundation for unix development. In *Winter 1986 USENIX Conference*, July 1986.

[App91] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, chapter 4. The MIT Press, 1991.

[BGS+94] M. L. Bailey, B. Gopal, P. Sarkar, M. A. Pagels, and L. L. Peterson. Pathfinder: A pattern-based packet classifier. In *Proceedings of the 1st Symposium on Operating System Design and Implementation.* USENIX, November 1994.

[Bra88] R. T. Braden. A pseudo-machine for packet monitoring and statistics. In *Proceedings of the Symposium on Communications Architectures and Protocols(SIGCOMM)*, Stanford, CA, August 1988. ACM.

[DDH84] P. Dencker, K. Durre, and J. Heuft. Optimization of parser tables for portable compilers. *ACM Trans. Prog. Lang. and Systems*, 6(4), 1984.

[GJS94] N. Gehani, H. V. Jagadish, and O. Shmueli. Compose: A system for composite event specification and detection. In *Book chapter in Advanced Database Concepts and Research Issues (N. R. Adam and B. Bhargava, eds.), Lecture Notes in Computer Science.* Springer Verlag, 1994.

[Gro88] J. Grosch. Generators for high-speed front ends. Technical Report 11, University of Karlsruhe, September 1988.

[Hor90] R. N. Horspool. Incremental generation of LR parsers. *Computer Languages*, 15(4), 1990.

[HW90] R. Nigel. Horspool and Michael Whitney. Even faster LR parsing. *Software-Practice and Experience*, 20(6), June 1990.

[isi94] *Isis Distributed Systems Inc., Marlboro, MA, Isis User's Guide: Reliable Distributed Objects for C++*, April 1994.

[JLM89] V. Jacobson, C. Leres, and S. McCanne. *The Tcpdump Manual Page.* Lawrence Berkeley Laboratory, Berkeley, CA, June 1989.

[Joh79] S. C. Johnson. *YACC - yet another compiler compiler in UNIX Programmer's manual*, 7 edition, 1979.

[Ler92] C. Leres. *The Arpwatch Manual Page.* Lawrence Berkeley Laboratory, Berkeley, CA, Sept. 1992.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter. In *Winter USENIX*, pages 259–269, January 1993.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The Packet Filter: An efficient mechanism for user-level network code. In *Proceedings of 11th Symposium on Operating Systems Principles. ACM*, pages 39–51, November 1987.

[nit90] *Sun Microsystems Inc. Mountain View, CA, NIT(4P); SunOS 4.11 Reference Manual*, Oct. 1990.

[Pen86] T. J. Pennello. Very fast LR parsing. *ACM SIGPLAN Notices*, 21(7), 1986.

[Sch94] Douglas C. Schmidt. High-Performance Event Filtering for Dynamic Multi-point Applications. In 1^{st} *Workshop on High Performance Protocol Architectures (HIPPARCH)*, Sophia Antipolis, France, December 1994. INRIA.

[YBMM94] Masanobu Yuhara, Brian. N. Bershad, Chris Maeda, and J. Elliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter 1994 USENIX Conference*, January 1994.