The BSD Packet Filter: A New Architecture for User-level Packet Capture*

Steven McCanne† and Van Jacobson†
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720
mccanne@ee.lbl.gov, van@ee.lbl.gov

December 19, 1992

Abstract

Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be minimized by deploying a kernel agent called a *packet filter*, which discards unwanted packets as early as possible. The original Unix packet filter was designed around a stack-based filter evaluator that performs sub-optimally on current RISC CPUs. The BSD Packet Filter (BPF) uses a new, register-based filter evaluator that is up to 20 times faster than the original design. BPF also uses a straightforward buffering strategy that makes its overall performance up to 100 times faster than Sun's NIT running on the same hardware.

1 Introduction

Unix has become synonymous with high quality networking and today's Unix users depend on having reliable, responsive network access. Unfortunately, this dependence means that network trouble can make it impossible to get useful work done and increasingly users and system administrators find that a large part of their time is spent isolating and fixing network problems. Problem solving requires appropriate diagnostic and analysis tools and, ideally, these tools should be available where the problems are—on Unix workstations. To allow such tools to be constructed, a kernel must contain some facility that gives user-level programs access to raw, unprocessed network traffic.[7] Most of today's workstation operating systems contain such a facility, e.g., NIT[10] in SunOS, the Ultrix Packet Filter[2] in DEC's Ultrix and Snoop in SGI's IRIX.

These kernel facilities derive from pioneering work done at CMU and Stanford to adapt the Xerox Alto 'packet filter' to a Unix kernel[8]. When completed in 1980, the CMU/Stanford Packet Filter, CSPF, provided a much needed and widely used facility. However on today's machines its performance, and the performance of its descendents, leave much to be desired — a design that was entirely appropriate for a 64KB PDP-11 is simply not a good match to a 16MB Sparcstation 2. This paper describes the BSD Packet Filter, BPF, a new kernel architecture for packet capture. BPF offers substantial performance improvement over existing packet capture facilities—10 to 150 times faster than Sun's NIT and 1.5 to 20 times faster than CSPF on the same hardware and traffic mix. The performance increase is the result of two architectural improvements:

- BPF uses a re-designed, register-based 'filter machine' that can be implemented efficiently on today's register based RISC CPU. CSPF used a memory-stack-based filter machine that worked well on the PDP-11 but is a poor match to memory-bottlenecked modern CPUs.
- BPF uses a simple, non-shared buffer model made possible by today's larger address spaces. The model is very efficient for the 'usual cases' of packet capture.[1]

In this paper, we present the design of BPF, outline how it interfaces with the rest of the system, and describe the new approach to the filtering mechanism. Finally, we present performance measurements of BPF, NIT, and CSPF which show why BPF performs better than the other approaches.

*This is a preprint of a paper to be presented at the 1993 Winter USENIX conference, January 25–29, 1993, San Diego, CA.

†This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

[1] As opposed to, for example, the AT&T STREAMS buffer model used by NIT which has enough options to be Turing complete but appears to be a poor match to any practical problem.

2 The Network Tap

BPF has two main components: the network tap and the packet filter. The network tap collects copies of packets from the network device drivers and delivers them to listening applications. The filter decides if a packet should be accepted and, if so, how much of it to copy to the listening application.

Figure 1 illustrates BPF's interface with the rest of the system. When a packet arrives at a network interface the link level device driver normally sends it up the system protocol stack. But when BPF is listening on this interface, the driver first calls BPF. BPF feeds the packet to each participating process' filter. This user-defined filter decides whether a packet is to be accepted and how many bytes of each packet should be saved. For each filter that accepts the packet, BPF copies the requested amount of data to the buffer associated with that filter. The device driver then regains control. If the packet was not addressed to the local host, the driver returns from the interrupt. Otherwise, normal protocol processing proceeds.

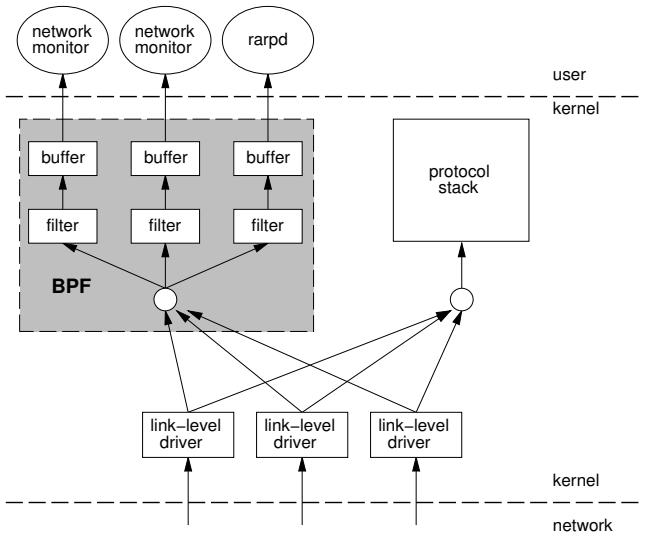

Figure 1: BPF Overview

Since a process might want to look at every packet on a network and the time between packets can be only a few microseconds, it is not possible to do a read system call per packet and BPF must collect the data from several packets and return it as a unit when the monitoring application does a read. To maintain packet boundaries, BPF encapsulates the captured data from each packet with a header that includes a time stamp, length, and offsets for data alignment.

2.1 Packet Filtering

Because network monitors often want only a small subset of network traffic, a dramatic performance gain is realized by filtering out unwanted packets in interrupt context. To minimize memory traffic, the major bottleneck in most modern workstations, the packet should be filtered 'in place' (e.g., where the network interface DMA engine put it) rather than copied to some other kernel buffer before filtering. Thus, if the packet is not accepted, only those bytes that were needed by the filtering process are referenced by the host.

In contrast, SunOS's STREAMS NIT [10] copies the packets before filtering and as a result suffers a performance degradation. The STREAMS packet filter module (*nit_pf(4M)*) sits on top of the packet capture module (*nit_if(4M)*). Each packet received is copied to an mbuf, and passed off to NIT, which then allocates a STREAMS message buffer and copies in the packet. The message buffer is then sent upstream to the packet filter, which may decide to discard the packet. Thus, a copy of each packet is always made, and many CPU cycles will be wasted copying unwanted packets.

2.2 Tap Performance Measurements

Before discussing the details of the packet filter, we present some measurements which compare the the relative costs of the BPF and SunOS STREAMS buffering models. This performance is independent of the packet filtering machinery.

We configured both BPF and NIT into the same SunOS 4.1.1 kernel, and took our measurements on a Sparcstation 2. The measurements reflect the overhead incurred during the interrupt processing — i.e., how long it takes each system to stash the packet into a buffer. For BPF we simply measured the before and after times of the tap call, *bpf_tap()*, using the Sparcstation's microsecond clock. For NIT we measured the time of the tap call *snit_intr()* plus the additional overhead of copying promiscuous packets to mbufs. (Promiscuous packets are those packets which were not addressed to the local host, and are present only because the packet filter is running.) In other words, we included the performance hit that NIT takes for not filtering packets in place. To obtain accurate timings, interrupts were locked out during the instrumented code segments.

The data sets were taken as a histogram of processing time versus packet length. We plotted the mean processing per packet versus packet size, for two configurations: an "accept all" filter, and a "reject all" filter.

In the first case, the STREAMS NIT buffering module (*nit_buf(4M)*) was pushed on the NIT stream with its *chunksize* parameter set to the 16K bytes. Similarly, BPF was configured to use 16K buffers. The packet filtering module which usually sits between the NIT interface and NIT buffering modules was omitted to effect "accept all" semantics. In both cases, no truncation limits were specified. This data is shown in Figure 2. Both BPF and NIT show a linear growth with captured packet size reflecting the cost of packet-to-filter buffer copies. However the different slopes of the BPF and

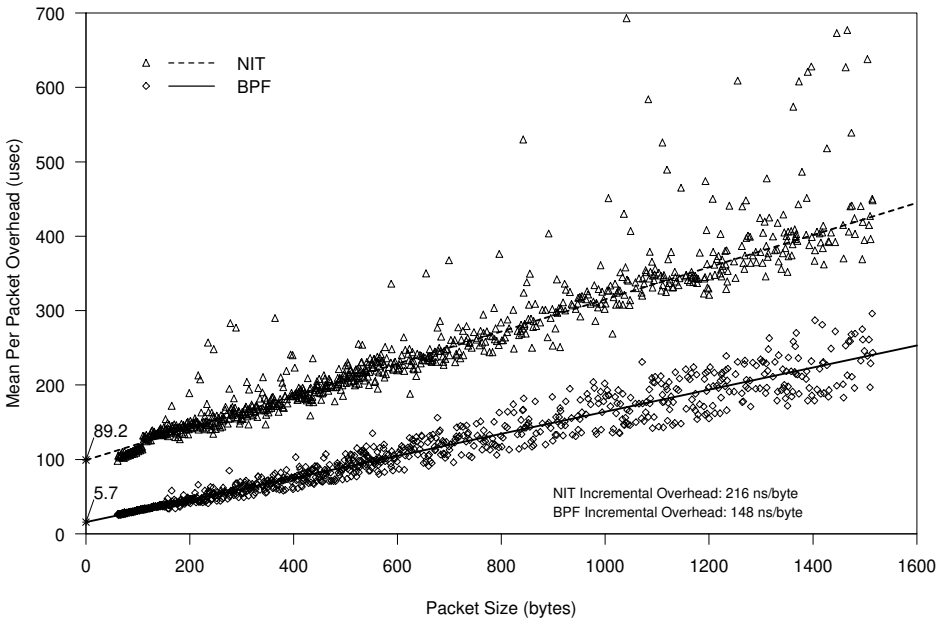

Figure 2: NIT versus BPF: "accept all"

NIT lines show that BPF does its copies at memory speed (148ns/byte) while NIT runs 45% slower (216ns/byte).[2] The y-intercept gives the fixed per-packet processing overhead: The overhead of a BPF call is about 6μs while NIT is 15 times worse at 89μs per packet. Much of this huge disparity appears to be due to the cost of allocating and initializing a buffer under the remarkably baroque AT&T STREAM I/O system.[3]

Figure 3 shows the results for the "reject all" configuration. Here the STREAMS packet filter module was configured with a "reject all" filter and pushed directly on top of the NIT interface module (the NIT buffering module was not used). Since the filter discards all packets, the processing time should be constant, independent of the packet size. For BPF this is true — we see essentially the same fixed cost as last time (5μs instead of 6 since rejecting avoids a call to the BPF copy routine) and no effect due to packet size. However, as explained earlier, NIT doesn't filter packets in place but instead copies packets then runs the filter over the copies.[4] Thus the cost of running NIT increases with packet size *even when the packet is discarded by the filter*. For large packets, this gratuitous copy makes NIT almost two orders of magnitude more expensive than BPF (450μs vs. 5μs).

The major lesson here is that filtering packets early and in place pays off. While a STREAMS-like design might appear to be modular and elegant, the performance implications of module partitioning need to be considered in the design. We claim that even a STREAMS-based network tap should include the packet filtering and buffering functionality in its lowest layer. There is very little design advantage in factoring the packet filter into a separate streams module, but great performance advantage in integrating the packet filter and the tap into a single unit.

3 The Filter Model

Assuming one uses reasonable care in the design of the buffering model,[5] it will be the dominant cost of packets you accept while the packet filter computation will be the dominant cost of packets you reject. Most applications of a packet capture facility reject far more packets than they accept and, thus, good performance of the packet filter is critical to good overall performance.

[2] This difference is due to the fact that NIT is not as careful about alignment as BPF. The network driver wants the IP header aligned on a longword boundary, but an Ethernet header is 14 bytes so the start of the packet is short-word aligned. Since NIT copies the packet to a longword aligned boundary, an inefficient, misaligned bcopy results. This oversight will be felt twice — once in this measurement, and again at the user-level, when for instance, a network monitor like *tcpdump* or *etherfind* must copy the network-layer portion of the packet to a longword aligned boundary.

[3] You might note anomalous behavior near the origin for the NIT data in both this and the following graph. STREAMS must allocate an mblk (a STREAMS buffer descriptor) for every packet. For small packets, the packet data is copied into a region of the mblk while large packets must use a more elaborate allocator involving additional dblk ('data block') allocations.

[4] The copy is required because the filter is a separate STREAM module pushed on top of the capture module and, thus, the capture module must copy data to STREAM buffers to send it up the stream to the filter module. As was the case with capture/buffer separation, the documentation notes this capture/filter separation is a feature, not a bug.

[5] E.g., *not* STREAMS.

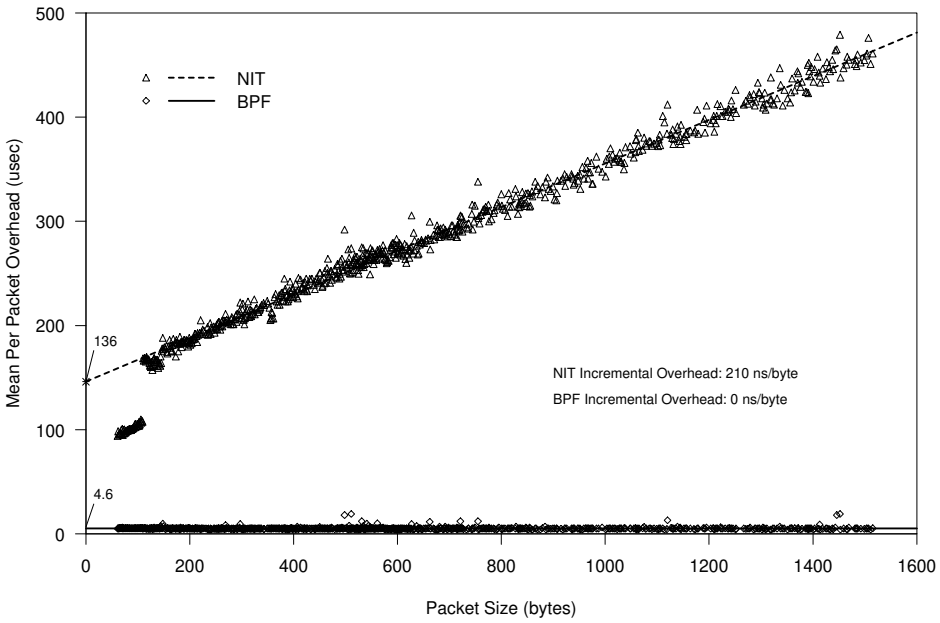

Figure 3: NIT versus BPF: "reject all"

A packet filter is simply a boolean valued function on a packet. If the value of the function is *true* the kernel copies the packet for the application; if it is *false* the packet is ignored.

Historically there have been two approaches to the filter abstraction: a boolean expression tree (used by CSPF) and a directed acyclic control flow graph or CFG (first used by NNStat[1] and used by BPF). For example, Figure 4 illustrates the two models with a filter that recognizes either IP or ARP packets on an Ethernet. In the tree model each node represents a boolean operation while the leaves represent test predicates on packet fields. The edges represent operator-operand relationships. In the CFG model each node represents a packet field predicate while the edges represent control transfers. The righthand branch is traversed if the predicate is true, the lefthand branch if false. There are two terminating leaves which represent *true* and *false* for the entire filter.

These two models of filtering are computationally equivalent. I.e., any filter that can be expressed in one can be expressed in the other. However, in implementation they are very different: The tree model maps naturally into code for a stack machine while the CFG model maps naturally into code for a register machine. Since most modern machines are register based, we will argue that the CFG approach lends itself to a more efficient implementation.

3.1 The CSPF (Tree) Model

The CSPF filter engine is based on an operand stack. Instructions either push constants or packet data on the stack, or perform a binary boolean or bitwise operation on the top two elements. A filter program is a sequentially executed list of instructions. After evaluating a program, if the top of stack has a non-zero value or the stack is empty then the packet is accepted, otherwise it is rejected.

There are two implementation shortcomings of the expression tree approach:[6]

- The operand stack must be simulated. On most modern machines this means generating add and subtract operations to maintain a simulated stack pointer and actually doing loads and stores to memory to simulate the stack. Since memory tends to be the major bottleneck in modern architectures, a filter model that can use values in machine registers and avoid this memory traffic will be more efficient.

- The tree model often does unnecessary or redundant computations. For example, the tree in fig. 4 will compute the value of 'ether.type == ARP' even if the test for IP is true. While this problem can be somewhat mitigated by adding 'short circuit' operators to the filter machine, some inefficiency is intrinsic: Because of the

[6]Note that it is not our intention to denigrate CSPF or its enormous contribution to the community — we simply wish to investigate the implementation implications of its filter model when run on modern hardware. The CSPF filtering mechanism was intended to support efficient protocol demultiplexing for user-level network code. The initial implementation achieved huge gains by performing user-specified demultiplexing inside the kernel rather than in a user-process. After this, the incremental gain from a more efficient filter design was negligible and, as a result, the designers of CSPF invested less effort in the filter machinery and, indeed, have pointed out that the "filter language is not a result of careful analysis but rather embodies several accidents of history"[8].

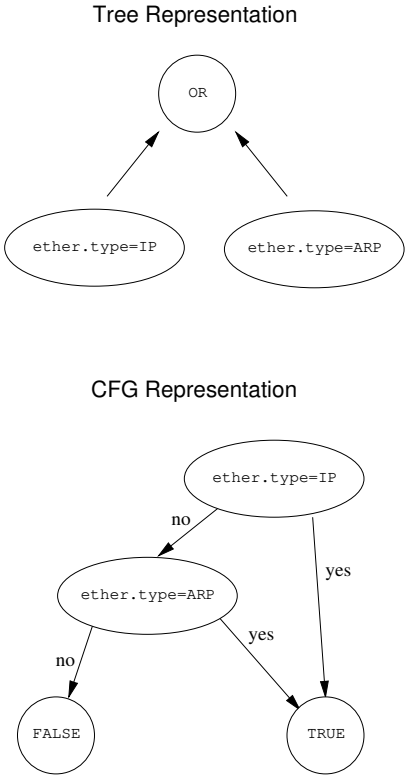

Figure 4: Filter Function Representations.

hierarchical design of network protocols, packet headers must be *parsed* to reach successive layers of encapsulation. Since each leaf of the expression tree represents a packet field, independent of other leaves, redundant parses may be carried out to evaluate the entire tree. In the CFG representation, it is always possible to reorder the graph in such a way that *at most* one parse is done for any layer.[7]

Another problem with CSPF, recognized by the designers, is its inability to parse variable length packet headers, e.g., TCP headers encapsulated in a variable length IP header. Because the CSPF instruction set didn't include an indirection operator, only packet data at fixed offsets is accessible. Also, the CSPF model is restricted to a single sixteen bit data type which results in a doubling of the number of operations to manipulate 32 bit data such as Internet addresses or TCP sequence numbers. Finally, the design does not permit access to the last byte of an odd-length packet.

While the CSPF model has shortcomings, it offers a novel generalization of packet filtering: The idea of putting a pseudo-machine language interpreter in the kernel provides a nice abstraction for describing and implementing the filtering mechanism. And, since CSPF treats a packet as a simple array of bytes, the filtering model is completely protocol independent. (The application that specifies the filter is responsible for encoding the filter appropriately for the underlying network media and protocols.)

The BPF model, described in the next section, is an attempt to maintain the strengths of CSPF while addressing its limitations and the performance shortcomings of the stack-based filter machine.

3.2 The BPF Model

3.2.1 CFGs vs. Trees

BPF uses the CFG filter model since it has a significant performance advantage over the expression tree model. While the tree model may need to redundantly parse a packet many times, the CFG model allows parse information to be 'built into' the flow graph. I.e., packet parse state is 'remembered' in the graph since you know what paths you must have traversed to reach to a particular node and once a subexpression is evaluated it need not be recomputed since the control flow graph can always be (re-)organized so the value is only used at nodes that follow the original computation.

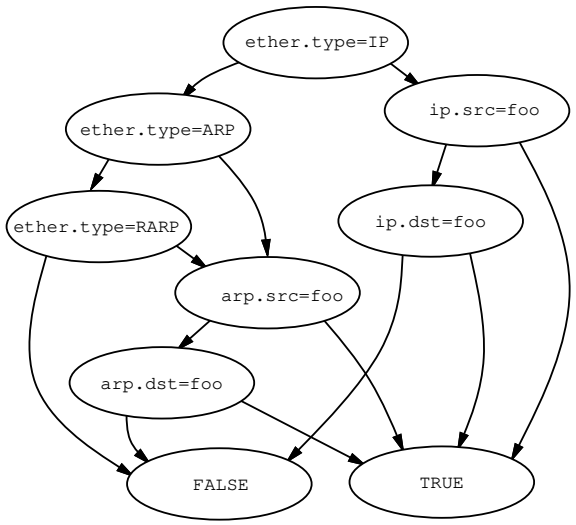

Figure 5: CFG Filter Function for "host foo".

For example, Figure 5 shows a CFG filter function that accepts all packets with an Internet address *foo*. We consider a scenario where the network layer protocols are IP, ARP, and Reverse ARP, all of which contain source and destination Internet addresses. The filter should catch all cases. Accordingly, the link layer type field is tested first. In the case of IP packets, the IP host address fields are queried, while in the case of ARP packets, the ARP address fields are used. Note that once we learn that the packet is IP, we do not need

[7] This graph reordering is, however, a non-trivial problem. Our BPF compiler (part of tcpdump[4]) contains a fairly sophisticated optimizer to reorder and minimize CFG filters. This optimizer is the subject of a future paper.

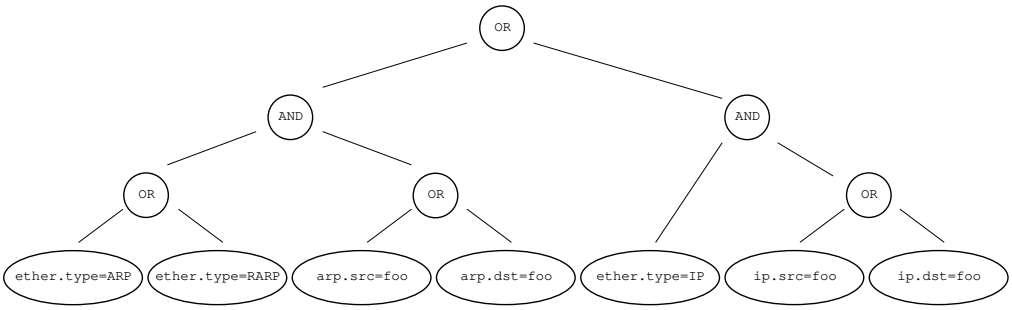

Figure 6: Tree Filter Function for “host foo”.

to check that it might be ARP or RARP. In the expression tree model, shown in figure 6, seven comparison predicates and six boolean operations are required to traverse the entire tree. The longest path through the CFG has five comparison operations, and the average number of comparisons is three.

3.2.2 Design of filter pseudo-machine

The use of a control flow graph rather than an expression tree as the theoretical underpinnings of the filter pseudo-machine is a necessary step towards an efficient implementation but it is not sufficient. Even after leveraging off the experience and pseudo-machine models of CSPF and NNStat[1], the BPF model underwent several generations (and several years) of design and test. We believe the current model offers sufficient generality with no sacrifice in performance. Its evolution was guided by the following design constraints:

1. It must be protocol independent. The kernel should not have to be modified to add new protocol support.
2. It must be general. The instruction set should be rich enough to handle unforeseen uses.
3. Packet data references should be minimized.
4. Decoding an instruction should consist of a single C switch statement.
5. The abstract machine registers should reside in physical registers.

Like CSPF, constraint 1 is adhered to simply by not mentioning any protocols in the model. Packets are viewed simply as byte arrays.

Constraint 2 means that we must provide a fairly general computational model, with control flow, sufficient ALU operations, and conventional addressing modes.

Constraint 3 requires that we only ever touch a given packet word once. It is common for a filter to compare a given packet field against a set of values, then compare another field against another set of values, and so on. For example, a filter might match packets addressed to a set of machines, or a set of TCP ports. Ideally, we would like to cache the packet field in a register and compare it across the set of values. If the field is encapsulated in a variable length header, we must parse the outer headers to reach the data. Furthermore, on alignment restricted machines, accessing multi-byte data can involve an expensive byte-by-byte load. Also, for packets in *mbufs*, a field access may involve traversing an *mbuf* chain. After we have done this work once, we should not do it again.

Constraint 4 means that we will have an efficient instruction decoding step but it precludes an orthogonal addressing mode design unless we are willing to accommodate a combinatorial explosion of switch cases. For example, while three address instructions make sense for a real processor (where much work is done in parallel) the sequential execution model of an interpreter means that each address descriptor would have to be decoded serially. A single address instruction format minimizes the decode, while maintaining sufficient generality.

Finally, Constraint 5 is a straightforward performance consideration. Along with constraint 4, it enforces the notion that the pseudo-machine register set should be small.

These constraints prompted the adoption of an accumulator machine model. Under this model, each node in the flowgraph computes its corresponding predicate by computing a value into the accumulator and branching based on that value. Figure 7 shows the filter function of Figure 5 using the BPF instruction set.

3.3 The BPF Pseudo-Machine

The BPF machine abstraction consists of an accumulator, an index register (x), a scratch memory store, and an implicit program counter. The operations on these elements can be categorized into the following groups:

1. LOAD INSTRUCTIONS copy a value into the accumulator or index register. The source can be an immediate value, packet data at a fixed offset, packet data at a variable offset, the packet length, or the scratch memory store.

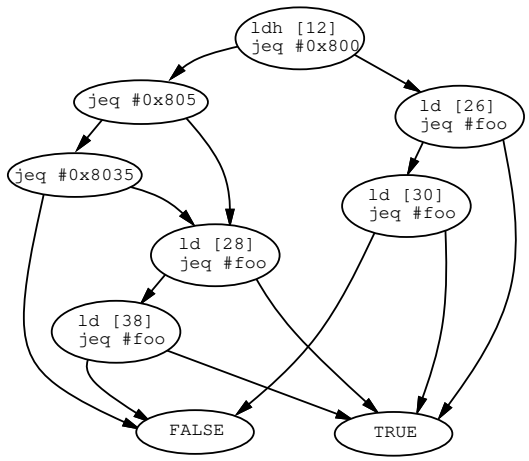

Figure 7: BPF Program for "host foo".

2. STORE INSTRUCTIONS copy either the accumulator or index register into the scratch memory store.

3. ALU INSTRUCTIONS perform arithmetic or logic on the accumulator using the index register or a constant as an operand.

4. BRANCH INSTRUCTIONS alter the flow of control, based on comparison test between a constant or *x* register and the accumulator.

5. RETURN INSTRUCTIONS terminate the filter and indicated what portion of the packet to save. The packet is discarded entirely if the filter returns 0.

6. MISCELLANEOUS INSTRUCTIONS comprise everything else — currently, register transfer instructions.

The fixed-length instruction format is defined by as follows:

opcode:16	jt:8	jf:8
k:32		

The *opcode* field indicates the instruction type and addressing modes. The *jt* and *jf* fields are used by the conditional jump instructions and are the offsets from the next instruction to the true and false targets. The *k* field is a generic field used for various purposes.

Table 1 shows the entire BPF instruction set. We have adopted this "assembler syntax" as a means of illustrating BPF filters and for debugging output. The actual encodings are defined with C macros, the details of which we omit here (see [6] for full details). The column labelled *addr modes* lists the addressing modes allowed for each instruction listed in the *opcode* column. The semantics of the addressing modes are listed in Table 2.

opcodes	*addr modes*				
ldb	[k]			[x+k]	
ldh	[k]			[x+k]	
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]&0xf)	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k		x		
sub	#k		x		
mul	#k		x		
div	#k		x		
and	#k		x		
or	#k		x		
lsh	#k		x		
rsh	#k		x		
ret	#k		a		
tax					
txa					

Table 1: BPF Instruction Set

The load instructions simply copy the indicated value into the accumulator (`ld`, `ldh`, `ldb`) or index register (`ldx`). The index register cannot use the packet addressing modes. Instead, a packet value must be loaded into the accumulator and transferred to the index register, via `tax`. This is not a common occurrence, as the index register is used primarily to parse the variable length IP header, which can be loaded directly via the `4*([k]&0xf)` addressing mode. All values are 32 bit words, except packet data can be loaded into the accumulator as unsigned bytes (`ldb`) or unsigned halfwords (`ldh`). Similarly, the scratch memory store is addressed as an array of 32 bit words. The instruction fields are all in host byte order, and the load instructions convert packet data from network order to host order. Any reference to data beyond the end of the packet terminates the filter with a return value of zero (i.e., the packet is discarded).

The ALU operations (`add`, `sub`, etc.) perform the indicated operation using the accumulator and operand, and store the result back into the accumulator. Division by zero terminates the filter.

The jump instructions compare the value in the accumulator with a constant (`jset` performs a "bitwise and" — useful for conditional bit tests). If the result is true (or non-zero), the true branch is taken, otherwise the false branch is taken. Arbitrary comparisons, which are less common, can be done by subtracting and comparing to 0. Note that there are no `jlt`, `jle` or `jne` opcodes since these can be built from the

`#k`	the literal value stored in *k*
`#len`	the length of the packet
`M[k]`	the word at offset *k* in the scratch memory store
`[k]`	the byte, halfword, or word at byte offset *k* in the packet
`[x+k]`	the byte, halfword, or word at offset *x*+*k* in the packet
`L`	an offset from the current instruction to L
`#k, Lt, Lf`	the offset to `Lt` if the predicate is true, otherwise the offset to `Lf`
`x`	the index register
`4*([k]&0xf)`	four times the value of the low four bits of the byte at offset *k* in the packet

Table 2: BPF Addressing Modes

codes above by reversing the branches. Since jump offsets are encoded in eight bits, the longest jump is 256 instructions. Jumps longer than this are conceivable, so a *jump always* opcode is provided (`jmp`) that uses the 32 bit operand field for the offset.

The return instructions terminate the program and indicate how many bytes of the packet to accept. If that amount is 0, the packet will be rejected entirely. The actual amount accepted will be the minimum of the length of the packet and the amount indicated by the filter.

3.4 Examples

We now present some examples to illustrate how packet filters can be expressed using the BPF instruction set. (In all the examples that follow, we assume Ethernet format for the link level headers.)

This filter accepts all IP packets:

```
      ldh   [12]
      jeq   #ETHERTYPE_IP, L1, L2
L1:   ret   #TRUE
L2:   ret   #0
```

The first instruction loads the Ethernet type field. We compare this to type IP. If the comparison fails, zero is returned and the packet is rejected. If it is successful, TRUE is returned and the packet is accepted. (TRUE is some non-zero value that represents the number of bytes to save.)

This next filter accepts all IP packets which did not originate from two particular IP networks, 128.3.112 or 128.3.254. If the Ethernet type is IP, the IP source address is loaded and the high 24 bits are masked off. This value is compared with the two network addresses:

```
      ldh   [12]
      jeq   #ETHERTYPE_IP, L1, L4
L1:   ld    [26]
      and   #0xffffff00
      jeq   #0x80037000, L4, L2
L2:   jeq   #0x8003fe00, L4, L3
L3:   ret   #TRUE
L4:   ret   #0
```

3.5 Parsing Packet Headers

The previous examples assume that the data of interest lie at fixed offsets in the packet. This is not the case, for example, with TCP packets, which are encapsulated in a variable length IP header. The start of TCP header must be computed from the length given in the IP header.

The IP header length is given by the low four bits of the first byte in the IP section (byte 14 on an Ethernet). This value is a word offset, and must be scaled by four to get the corresponding byte offset. The instructions below will load this offset into the accumulator:

```
ldb   [14]
and   #0xf
lsh   #2
```

Once the IP header length is computed, data in the TCP section can be accessed using indirect loads. Note that the effective offset has three components:

- the IP header length,
- the link level header length, and
- the data offset relative to the TCP header.

For example, an Ethernet header is 14 bytes and the destination port in a TCP packet is at byte two. Thus, adding 16 to the IP header length gives the offset to the TCP destination port. The previous code segment is shown below, augmented to test the TCP destination port against some value *N*:

```
      ldb   [14]
      and   #0xf
      lsh   #2
      tax
      ldh   [x+16]
      jeq   #N, L1, L2
L1:   ret   #TRUE
L2:   ret   #0
```

Because the IP header length calculation is a common operation, the `4*([k]&0xf)` addressing mode was introduced. Substituting in the `ldx` instruction simplifies the filter into:

```
      ldx   4*([14]&0xf)
      ldh   [x+16]
      jeq   #N, L1, L2
L1:   ret   #TRUE
L2:   ret   #0
```

However, the above filter is valid only if the data we are looking at is really a TCP/IP header. Hence, the filter must also check that link layer type is IP, and that the IP protocol type is TCP. Also, the IP layer might fragment a TCP packet, in which case the TCP header is present only in the first fragment. Hence, any packets with a non-zero fragment offset should be rejected. The final filter is shown below:

```
      ldh   [12]
      jeq   #ETHERPROTO_IP, L1, L5
L1:   ldb   [23]
      jeq   #IPPROTO_TCP, L2, L5
L2:   ldh   [20]
      jset  #0x1fff, L5, L3
L3:   ldx   4*([14]&0xf)
      ldh   [x+16]
      jeq   #N, L4, L5
L4:   ret   #TRUE
L5:   ret   #0
```

3.6 Filter Performance Measurements

We profiled the BPF and CSPF filtering models outside the kernel using *iprof* [9], an instruction count profiler. To fully compare the two models, an indirection operator was added to CSPF so it could parse IP headers. The change was minor and did not adversely affect the original filtering performance. Tests were run on large packet trace files gathered from a busy UC Berkeley campus network. Figure 8 shows the results for four fairly typical filters.

Filter 1 is trivial. It tests whether one 16 bit word in the packet is a given value. The two models are fairly comparable, with BPF faster by about 50%.

Filter 2 looks for a particular IP host (source or destination) and shows more of a disparity—a performance gap of 240%. The larger difference here is due mostly to the fact that CSPF operates only on 16 bit words and needs two comparison operations to determine the equality of a 32 bit Internet address.

Filter 3 is an example of packet parsing (required to locate the TCP destination port field) and illustrates a yet greater performance gap. The BPF filter parses the packet once, loading the port field into the accumulator then simply does a comparison cascade of the interesting ports. The CSPF filter must re-do the parse and relocate the TCP header for each port to be tested.

Finally, filter 5 demonstrates the effect of the unnecessary computations done by CSPF for a filter similar to the one described in figures 6 and 5.

4 Applications

BPF is now about two years old and has been put to work in several applications. The most widely used is *tcpdump* [4], a network monitoring and data acquisition tool. Tcpdump performs three primary tasks: filter translation, packet acquisition, and packet display. Of interest here is the filter translation mechanism. A filter is specified with a user-friendly, high level description language. Tcpdump has a built in compiler (and optimizer) which translates the high level filters into BPF programs. Of course, this translation process is transparent to the user.

Arpwatch [5] is a passive monitoring program that tracks Ethernet to IP address mappings. It notifies the system administrator, via email, when new mappings are established or abnormal behavior is noted. A common administrative nuisance is the use of a single IP address by more than one physical host, which arpwatch dutifully detects and reports.

A very different application of BPF has been its incorporation into a variant of the Icon Programming Language [3]. Two new data types, a *packet* and a *packet generator* have been built into the Icon interpreter. Packets appear as first class record objects, allowing convenient "dot operator" access to packet headers. A packet generator can be instantiated directly off the network, or from a previously collected file of trace data. Icon is an interpreted, dynamically typed language with high level string scanning primitives and rich data structures. With the BPF extensions, it is well suited for the rapid prototyping of networking analysis tools.

Netload and *histo* are two network visualization tools which produce real time network statistics on an X display. Netload graphs utilization data in real time, using tcpdump style filter specifications. Histo produces a dynamic interarrival-time histogram of timestamped multimedia network packets.

The Reverse ARP daemon uses the BPF interface to read and write Reverse ARP requests and replies directly to the local network. (We developed this program to allow us to entirely replace NIT by BPF in our SunOS 4 systems. Each of the Sun NIT-based applications (etherfind, traffic, and rarpd) now has a BPF analog.)

Finally, recent versions of NNStat[1] and nfswatch can be configured to run over BPF (in addition to running over NIT).

5 Conclusion

BPF has proven to be an efficient, extensible, and portable interface for network monitoring. Our comparison studies have shown that it outperforms NIT in its buffer management and CSPF in its filtering mechanism. Its programmable pseudo-machine model has demonstrated excellent generality and extensibility (all knowledge of particular protocols is factored out of the kernel). Finally, the system is portable and runs on most BSD and BSD-derivative systems[8] and can interact with various data link layers[9].

[8] SunOS 3.5, HP-300 and HP-700 BSD, SunOS 4.x, 4.3BSD Tahoe/Reno, and 4.4BSD.

[9] Ethernet, FDDI, SLIP, and PPP are currently supported.