Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages

Masanobu Yuhara
Fujitsu Laboratories Ltd.
1015 Kamikodanaka
Nakahara-ku
Kawasaki 211, Japan

Brian N. Bershad *Chris Maeda*
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213

J. Eliot B. Moss
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003

July 11, 1993

Abstract

This paper describes a new packet filter mechanism that efficiently dispatches incoming network packets to one of multiple endpoints, for example address spaces. Earlier packet filter systems iteratively applied each installed filter against every incoming packet, resulting in high processing overhead whenever multiple filters existed. Our new packet filter provides an associative match function that enables similar but not identical filters to be combined together into a single filter. The filter mechanism, which we call MPF (Mach Packet Filter), has been implemented for the Mach 3.0 operating system and is being used to support endpoint-based protocol processing, whereby each address space implements its own suite of network protocols. With large numbers of registered endpoints, MPF outperforms the earlier BSD Packet Filter (BPF) by over a factor of 4, resulting in a 33% shorter latency in the kernel. The MPF also allows a filter program to dispatch fragmented packets, which was nearly impossible with previous filter mechanisms.

1 Introduction

In this paper, we describe a new packet filter mechanism, called MPF (Mach Packet Filter) that efficiently handles simultaneously installed multiple filters. Our new packet filter also deals well with context-dependent demultiplexing, which is necessary when receiving multiple packets in a fragmented message. We have implemented our new packet filter in the context of the Mach 3.0 operating system [Accetta et al. 86]. The new packet filter improves performance by taking advantage of the similarity between filter programs that occurs when performing endpoint-based protocol processing. With 10 TCP/IP sessions MPF performs 7 times faster than the CMU/Stanford packet filter (CSPF) [Mogul et al. 87], and 4 times faster than the BSD packet filter (BPF) [McCanne and Jacobson 93]. The advantage of MPF increases as the number of sessions grows.

The original packet filters (CSPF and BPF) shared two primary goals: protocol independence and generality. The filters did not depend on any protocol, and future protocols could be accommodated without changing the kernel. Our new filter mechanism shares these two goals, and fits alongside the BPF and CSPF implementations. (MPF is implemented as an extension to the base BPF language.) Consequently, a packet filter program built for CSPF or BPF will work with our system. Although MPF has been implemented for the Mach operating system, it has not required any changes to the Mach microkernel interface, and indeed has no Mach-specific aspects. Therefore, other BPF implementations could be extended to support MPF programs, and our implementation should port easily to other operating systems that support packet filters.

1.1 Motivation

A packet filter is a small body of code installed by user programs at or close to a network interrupt handler of an operating system kernel. It is intended to carry an incoming packet up to its next logical level of demultiplexing through a user-level process. An operating system kernel implements an interpreter that applies installed filters

against incoming network packets in their order of arrival. If the filter accepts the packet, the kernel sends it to its recipient address space.

The packet filter mechanism was originally intended to support network monitoring [Mogul et al. 87], and primarily functioned to place the machine in "promiscuous mode" whereby all packets were routed to a single spy process. Two packet filters, CSPF and BPF, are in common use in today's systems. CSPF is based on a stack machine. A filter program can push data from an input packet, execute ALU functions, branch, and notify whether it accepts or rejects the packet. BPF is a more recent packet filter mechanism. The BPF abstract machine is register-based and has two registers (`A` and `X`), an input packet (`P[]`), and a scratch memory (`M[]`). It executes load, store, ALU, and branch instructions as well as a return instruction that can specify the size of the packet to be delivered. As shown in [McCanne and Jacobson 93], BPF admits a somewhat more efficient interpreter than CSPF.

With microkernel technology, where traditional operating system services such as protocol processing are implemented outside the kernel, the original packet filter provided a convenient mechanism to route packets from the kernel to a single protocol server. In such cases, relatively few packet filters would ever be installed on a machine (typically two: one to recognize IP traffic, one to recognize all other traffic), so the scalability of the packet filter was not important. Unfortunately, a single point of dispatch for all network traffic resulted in communication overhead for microkernel-based systems substantially larger than for monolithic systems (those in which the protocols are all implemented in the kernel) [Maeda and Bershad 92].

To address this problem, we have decomposed the protocol service architecture so that each application is responsible for its own protocol processing [Maeda and Bershad 93]. That is, every address space contains, for example, a complete TCP/IP stack. Figure 1 illustrates the structural differences between the two different protocol strategies.

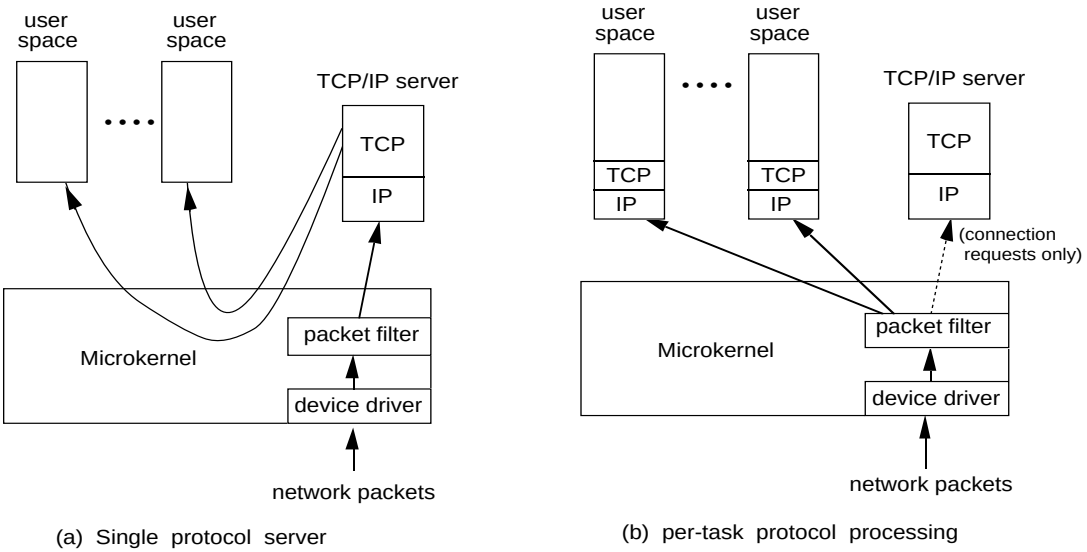

Figure 1: *Two ways to structure protocol processing. In the system on the left, all packets are routed through a central server and then on to their eventual destination. In the system on the right, the kernel routes an incoming, but unprocessed network packet directly to the address space for which the packet is ultimately intended.*

At its core, the new protocol architecture on the right relies on the kernel's packet filter to hand off incoming packets to the appropriate address space. For example, an address space for TCP/IP port 2371 would register its own packet filter that recognizes incoming packets destined for that port. Our distributed protocol architecture revealed two serious problems with the earlier implementations of the packet filter:

1. *The packet filter's programming interface was not designed to be scalable.* The dispatch overhead grew linearly with the number of potential endpoints. For even a workstation-class machine, it is not uncommon to have several hundred endpoints (ports) in use at a time, so scalability becomes critical for efficient demultiplexing.

2. *A packet filter is unable to efficiently recognize and dispatch multipacket messages.* Some protocols require information on the previous or future packets to dispatch a packet. For example, the IP protocol splits one large IP packet into several small IP packets when the underlying data link layer cannot accept a large

packet [RFC791]. Moreover, the fragmented packets may arrive out of order. The existing packet filters have no mechanisms for dealing with fragmentation, let alone out of order delivery. Therefore, they cannot dispatch fragmented packets to any of multiple endpoints. Instead, fragments must all be sent to a higher-level intermediary process using the "packet filter of last resort" at the expenses of substantially more kernel messages and boundary crossings.

1.2 Our solution

To deal with the scalability problem, our new packet filter takes advantage of the structural and logical similarity within a protocol, and dispatches all packets destined for that protocol in a single step. Typically, filter programs for a particular protocol consist of two parts: one that identifies the protocol and one that identifies the session in that protocol. (The code in Appendix A shows an example BPF program for TCP/IP dispatching.) The first part is exactly the same for all sessions within a protocol, while the second part differs only in the constant values that identify the particular session instance. Figure 2 contrasts the old packet filter mechanisms, which execute similar code repeatedly for each protocol for each session, with our own MPF.

The kernel's filter module internally transforms, or *collapses*, filter programs for the same protocol into a single filter program. A special new instruction in the packet filter enables associative lookup on packet data, and simplifies the identification and collapsing of similar code sequences.

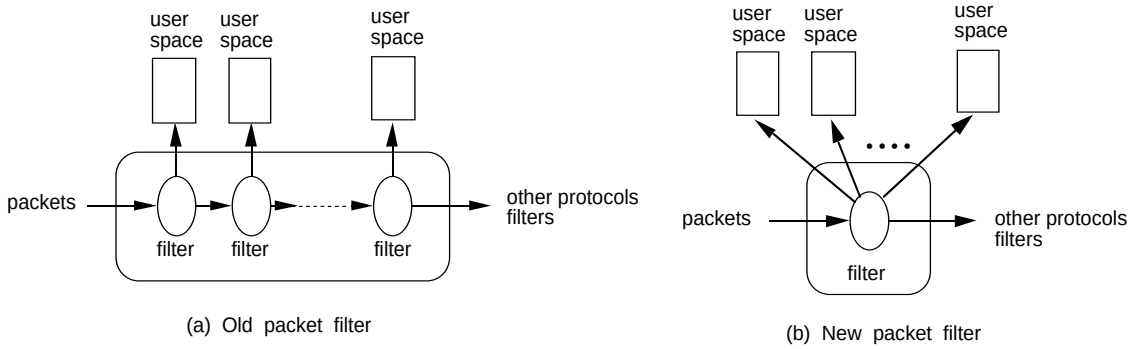

Figure 2: *Redundant (BPF and CSPF) vs. one-step filtering (MPF) for incoming packets*

To deal with the fragmentation problem, our new filter mechanism provides per-filter state that persists across the arrival of packets. Filter programs can use the state to record the dispatch information and use that recorded information to dispatch multipacket messages.

1.3 The rest of this paper

The rest of this paper is organized as follows. In Section 2 we describe the design and implementation of our single-pass filtering machinery. In Section 3 we discuss our support for dispatching fragmented messages. In Section 4 we review the performance of the new packet filter in the context of a large number of dispatchable endpoints per machines. Finally, in Section 5 we present our conclusions.

2 Fast dispatch of incoming packets to multiple endpoints

As mentioned in the previous section, filters generally consist of two levels of dispatch. The first level dispatches to a protocol, while the second level to an endpoint within that protocol. The logic for the first level of dispatch is identical for all packets destined to a particular protocol, while that for the second relies on mapping from some number of fields in the packet header to an actual address space/endpoint (for example, a Mach IPC port). We have introduced an associative match instruction (`ret_match_imm`) that allows MPF to exploit the fact that a packet is dispatched first to a protocol and then to a session within that protocol.

Figure 3 illustrates a sample MPF program that reveals the split-level dispatch and the use of the new instruction (the code sequence performs the same function as that in Appendix A). The filter accepts packets sent to a TCP/IP session specified by source IP address (`src_addr`), source TCP port (`src_port`), and destination TCP port (`dst_port`). The combination of these constant parameters is unique for a particular session. Other constant

parameters (including the destination IP address (**dst_addr**)) are the same for all TCP/IP sessions. The first part (A) of the MPF program checks if the packet uses the TCP/IP protocol. The second part (B) extracts the TCP session information from the packet and puts it into the scratch memory. Parts (A) and (B) are common to all TCP/IP filters. The last part (C) determines if the packet is in fact destined for this particular filter (session). The new **ret_match_imm** instruction makes it easier to identify the (possibly) common part (A and B) and the dispatch part (C). The **ret_match_imm** instruction is a combination of the packet filter's compare and return instructions, as shown in Table 1.

```
/* Part (A) */
        begin                           ; MPF/BPF identifier
        ldh     P[OFF_ETHERTYPE]        ; A = ethertype
        jeq     #ETHERTYPE_IP, L1, Fail ; If not IP, fail.
L1:
        ld      P[OFF_DST_IP]           ; A = dst IP address
        jeq     #dst_addr, L2, Fail     ; If not from dst_addr, fail.
L2:
        ldb     P[OFF_PROTO]            ; A = protocol
        jeq     #IPPROTO_TCP, L3, Fail  ; If not TCP, fail.
L3:
        ldh     P[OFF_FRAG]             ; A = Frag_flags|Frag_offset
        and     #!Dont_Frag_Bit         ; Clear Don't Fragment bit
        jeq     #0, L4, Fail            ; If fragmented, fail
L4:

/* Part (B) */
        ld      P[OFF_SRC_IP]           ; A = src IP address
        st      M[0]                    ; M[0] = A

        ldx     4 * (P[OFF_IHL] & 0xf)  ; X = offset to TCP header

        ld      P[x + OFF_SRC_PORT]     ; A = src TCP port
        st      M[1]                    ; M[1] = A

        ld      P[x + OFF_DST_PORT]     ; A = dst TCP port
        st      M[2]                    ; M[2] = A

/* Part (C) */
        ret_match_imm #ALL, 3           ; Compare keys and M[0..2].
        key     #src_addr               ; If matched, accept the
        key     #src_port               ; whole packet. If not,
        key     #dst_port               ; reject the packet.
Fail:
        ret     #0
```

Figure 3: *An MPF program for a TCP/IP session.*

The MPF implementation takes advantage of the new **ret_match_imm** instruction to "collapse" multiple filter programs into one, and to convert the **ret_match_imm** instruction into a fast associative lookup preceding the dispatch. We assume that in every filter using **ret_match_imm**, all instructions but the **ret_match_imm** are common with other filter programs with high probability. When a user task installs a new packet filter, the kernel's filter module checks whether the program contains any associative match instructions. If not, the filter is treated as either a CSPF or a BPF program (Mach 3.0 supports both kinds of filters). If a match instruction is found, the filter module then searches for a previously installed filter program with the same code except for the immediate values contained in the **key** instructions. These immediate values are used by the kernel to create a hash table used to dispatch incoming packets. Each installed filter has its own hash table by which the key fields in an incoming packet can be quickly matched to an endpoint (in our case, a Mach IPC port). The hash values are calculated using the immediate values of the key instructions. Each hash entry has the set of key values, and the corresponding receive port. Upon finding a similar filter, the kernel enters the new filter's immediate values into the base filter's hash table. If no similar filter exists, the kernel creates a new one. The just-installed filter program is now ready to receive packets.

In the example in Figure 3, the hashed values are the source IP address and the TCP source and destination port numbers. The implementation collapses all filter programs for TCP sessions into a single internal program.

When a packet arrives and the filter mechanism processes the collapsed filter, it executes the common part (A and B) just like a conventional program. If the program rejects the packet in the common part, the whole filter group rejects the packet. Therefore, MPF does not repeatedly execute the common protocol dispatch code in vain.

MPF match sequence	Equivalent BPF match sequence

`ret_match_imm #3, #ALL`
`key #key0`
`key #key1`
`key #key2`
	`ld M[0]` ; A = M[0]
	`jeq #key0, k1, fail` ; if (A == key0) goto k1; else goto fail;
	`k1: ld M[1]` ; A = M[1]
	`jeq #key1, k2, fail` ; if (A == key1) goto k2; else goto fail;
	`k2: ld M[2]` ; A = M[2]
	`jeq #key2, ok, fail` ; if (A == key2) goto ok; else goto fail;
	`ok: ret #ALL` ; return the whole packet
	`fail: ret #0` ; abort this filter

Table 1: *The* `ret_match_imm` *instruction from MPF and its equivalent sequence from BPF. The first argument of* `ret_match_imm` *indicates the number of data items to be compared. The subsequent key instructions provide immediate data. These immediate values are compared with the values in the scratch memory: M[0], M[1], M[2], respectively. If the corresponding values are equal, then the filter returns with success. The second argument of the* `ret_match_imm` *instruction specifies the number of bytes of the packet sent to the recipient (ALL indicates the entire packet). If any pair of the corresponding values is not equal, the filter terminates with failure, and the packet is not sent to the recipient for this filter.*

If the program does not reject the packet in the common part, the filter module executes the non-common part, namely the `ret_match_imm` instruction. The filter mechanism calculates the hash value from the data in the scratch memory (M[0] .. M[2] in the example), and searches for the data in its hash table. If the search is successful, then the packet is sent to the corresponding receive port. If the hash search fails, then the collapsed group of filter programs rejects the packet (but other filter programs might still apply).

2.1 Limitations

While our strategy limits the number of common sequences that a packet filter may have to one, and restricts the code following the common sequence to a single associative match instruction, it nevertheless admits a quite powerful and easy-to-implement optimization. A more general alternative would extend the packet filter implementation to identify and cache the result of arbitrary common sequences during filter processing. Our requirements (fast endpoint demultiplexing), combined with the two-tiered dispatch common to most protocols, convinced us to choose a solution that was simple to implement and right most of the time, rather than one that was more complicated to implement and right about as often.

3 Dispatching fragmented messages

Fragmentation occurs when a lower-level protocol layer cannot transfer the entire packet of a higher-level protocol. For example, consider the case of a protocol stack consisting of Ethernet, IP, and UDP. The UDP port number is required for demultiplexing, and is embedded at the beginning of an IP message. Since UDP messages can be larger than the maximum Ethernet message (4K bytes or larger for NFS packets over UDP, but only about 1500 bytes for Ethernet), the IP segment containing the UDP message must be fragmented. Only the first fragment will contain the UDP header (which includes the UDP port number), but each IP fragment will contain a fragment bit set in its header (except the last fragment), a unique message id, offset and length information, and finally the data. The unique message id, offset, and length are used to reassemble the incoming message.

Demultiplexing fragments is difficult for several reasons: only the first fragment contains the transport header (which provides the information needed for determining the target address space), fragments may arrive out of order, and some fragments may not arrive at all. We wanted to support simple and efficient demultiplexing of fragments using the packet filter. We were not concerned with performing actual reassembly at the packet filter layer, as we expected that service to be provided by the higher-level protocols.

To deal with fragmentation, we added the notion of per-filter static memory that allows packet filters to bridge between dispatch information present in the first fragment of a message (for example, the UDP port), and that present in subsequent fragments (for example, the unique message id). Specifically, we record the higher level session dispatch information and associate it with a lower level message id, allowing us to dispatch fragments to the correct address space. This information persists for a finite time, after which it is automatically removed.

Because fragments don't always arrive in order, we also allow a filter to postpone processing of a packet in the case where the first fragment does not arrive first, meaning that no dispatch information is available. These

fragments are postponed, and processed only after other packets have arrived (hopefully, the dispatch information has become available). Postponed packets are dropped if it appears the dispatch information will not become available, or if we run out of space.

We added four new instructions to the packet filter instruction set to handle fragmentation: `register_data`, `ret_match_data`, `postpone`, and `jmp_match_imm`. The filter module expects the first three instructions in the common part and `jmp_match_imm` in the non-common part. These instructions are described in Table 2, and used in Appendix B, which shows the flow of a TCP/IP filter program (psuedo code) that handles fragmented packets.

Instruction	Description
`register_data #N, #T`	This `register_data` instruction stores M[0]...M[N-1] of the scratch memory into the filter's static memory. The filter mechanism automatically removes the static memory data after T milliseconds. A filter program uses this instruction to store the information for the dispatch of fragmented packets. A `jmp_match_imm` instruction must be executed and must have found a match before this instruction is executed. This restriction helps the filter mechanism to record the data in an associative manner.
`ret_match_data #N, #R`	The `ret_match_data` instruction compares M[0]..M[N-1] of the scratch memory values with the static memory values of this filter if they exist. If the values are the same, R bytes of the packet are sent to the recipient of this filter. If not (or if the static memory values do not exist), execution continues with the next instruction. A filter program can use this instruction to dispatch fragmented packets.
`jmp_match_imm #N, Lt, Lf`	The `jmp_match_imm` instruction is similar to the `ret_match_imm` instruction in that N immediate data values following the instruction are compared with M[0] .. M[N-1] of the scratch memory. This instruction conditionally jumps forward depending on the result of the comparison. If the data match, control transfers to `Lt`, otherwise it goes to `Lf`.
`postpone #T`	This instruction postpones processing of the current packet, deferring it to some later time. The specific processing time is dependent on the implementation, but, the implementation must guarantee that the packet is processed again if the situation changes such that the packet can be dispatched. If a postponed packet is chosen for processing, it may be postponed again. The filter mechanism guarantees to discard the packet after T milliseconds from original arrival, but it may discard the packet earlier because of storage limitations.

Table 2: *New instructions to support handling of fragmented packets.*

The `register_data` and `ret_match_data` instructions store and retrieve the fragmentation information. When a packet filter executes the `register_data` instruction, the data in its scratch memory are used as keys associated with its receive port in a second (filter-specific) hash table. The `ret_match_data` instruction uses this hash table to provide fast lookup on the fragment information. Each entry in the second hash table has its own expiration time specified by the filter program as a timeout value.

The `jmp_match_imm` instruction is a branching version of the `ret_match_imm` instruction described in the previous section. If the match fails, the program branches to the false-case label. If the match succeeds, the program branches to the true-case label. As a side effect of succeeding, the receive port associated with the key data following the `jmp_match_imm` instruction becomes "associated" with the currently running packet filter. If the filter then executes a normal `return` instruction, the associated receive port is recalled and used as the recipient. In this way, we avoid explicitly manipulating kernel descriptors (really, IPC ports) within the packet filter, while still being able to collapse filters that handle fragmentation.

The `postpone` instruction addresses the situation where a later fragment arrives before the first fragment. We

assume that such out of order arrival is rare, so the postponement mechanism is quite simple. A filter module immediately gives up processing a postponed packet and adds the packet to a pending queue. Pending packets are reprocessed immediately after each new packet is filtered. Of course, the filter program may postpone the packet again. However, the packet's expiration time is set when it is first postponed, and the packet will be dropped after that time, or if the number of postponed packets becomes too large.

The fragmentation support described in this section imposes essentially no overhead on filtering of non-fragmented packets. Note that `jmp_match_imm`, unlike `ret_match_imm`, allows work to be done following the match on recipient. In particular, it allows the filter to record fragment matching data upon encountering the first fragment.

4 Performance

We evaluated the performance of the new MPF compared with CSPF and BPF. Since the motivation of MPF is per-task (user space) protocol processing, we used TCP/IP dispatch processing as our benchmark. The filter program we used is shown in Figure 3, Appendix A, and Appendix C.

To compare the latency of MPF with the previous packet filter implementations, we took the following measurements.

- Filtering time as a function of the number of sessions with a uniform hash distribution.
- Filtering time as a function of the number of sessions when all sessions hash to the same hash bucket.

In addition, since MPF collapses similar filters into a single filter program at installation time, we performed experiments to quantify the additional overhead incurred during the installation process. To determine the filter installation overhead for MPF, we measured the following.

- Filter installation time for a new session of the same protocol.
- Filter installation time for a different protocol.

The first experiment measures the time required to install a filter for a new session of an existing protocol such as TCP/IP. The second experiment measures the time to install a completely unrelated packet filter.

4.1 Packet Filter Latency

We measured packet filter latency by running the filter module at user level so that we could easily control the input packets. All measurements were on a DECstation 5000/200 (25 MHz MIPS R3000) using Mach 3.0 (MK82) and Unix server version UX41. The filter module and all relevant functions use exactly the same code as the kernel, except that the processor priority manipulations (`spl` calls) are omitted. However, these manipulations do not affect the filter's performance. Measurements were taken using a 25 MHz free-running counter mapped into the user address space. Since the counter is read with a single load instruction, the timestamp overhead is minimal.

[THE FINAL PAPER WILL ALSO CONTAIN IN-KERNEL MEASUREMENTS RUNNING UNDER A REAL PROTOCOL STACK.]

Packet filter performance is strongly affected by the cache; the time to filter a packet with a cold cache can be four times as long as the time with a warm cache. Since the cache's "warmth" depends on how frequently network packets arrive, packet filter latency will have high variance in a real system. To determine cache effects on performance, we ran our benchmarks with cold and warm caches. Cold-cache measurements were obtained by flushing the caches before each packet. This represents a worst-case (and reproducible) measurement. Warm-cache measurements were obtained by running the benchmark without flushing the caches. Actual performance will vary between these extremes, with the operating point depending on the frequency of packet arrival, and the nature of other system activity.

To help interpret the results, Table 3 shows the time required for the Ethernet device driver to service an incoming packet and for the kernel to deliver an incoming packet to the destination address space. These operations occur before and after the packet filter runs, respectively. Packet filtering (demultiplexing) should take much less time than these operations, which are dominated by data movement.

Latency vs. Number of sessions

The first experiment measures the time required to filter a TCP/IP packet, as a function of the number of active TCP/IP sessions. Each packet is unfragmented and contains 10 bytes of data. Note that the time to filter a packet is constant for all data sizes since the filters examine only the packet headers. Only TCP/IP protocol filters were

Operation	Packet Size	
	64	1514
Time to read an incoming packet from the Ethernet device.	0.1	0.5
Time to move an incoming packet from the kernel to the destination address space.	0.1	0.2

Table 3: The time (in milliseconds) required for the device driver to service incoming packets and for the kernel to deliver incoming packets to their destination address spaces. These times were measured on a DECstation 5000/200.

installed (i.e. there was only one collapsed filter for MPF) and the source IP addresses and source and destination ports were drawn from consecutive ranges.

Table 4 and Figure 4 show the results. The latency of both BPF and CSPF grows linearly as the number of sessions increases because they must run a filter program for each session. The latency for MPF, on the other hand, is insensitive to the number of sessions since only one filter program is executed to demultiplex packets for all sessions. With only ten TCP sessions, MPF shows performance that is 7.6 times better than CSPF and 4.3 times better than BPF (warm cache case). When we consider the total latency necessary to demultiplex a small packet in the kernel, MPF is 50% faster than CSPF and 33% faster than BPF (warm cache case). The advantage of MPF becomes greater as the number of sessions increases.

	# of sessions	1	2	6	10	20	40	60	80	100
warm cache	MPF [ms]	0.035	0.035	0.035	0.035	0.035	0.036	0.036	0.036	0.037
	BPF [ms]	0.031	0.042	0.090	0.149	0.330	0.559	0.830	1.188	1.481
	CSPF [ms]	0.053	0.081	0.169	0.266	0.511	0.998	1.538	2.103	2.613
	BPF/MPF	0.9	1.2	2.6	4.3	9.5	15.7	23.4	32.8	40.1
	CSPF/MPF	1.5	2.3	4.9	7.6	14.7	28.1	43.3	58.1	70.7
cold cache	MPF [ms]	0.153	0.154	0.154	0.155	0.155	0.157	0.159	0.160	0.163
	BPF [ms]	0.127	0.144	0.212	0.280	0.436	0.787	1.112	1.429	1.768
	CSPF [ms]	0.132	0.160	0.268	0.382	0.658	1.205	1.753	2.293	2.861
	BPF/MPF	0.8	0.9	1.4	1.8	2.8	5.0	7.0	8.9	10.8
	CSPF/MPF	0.9	1.0	1.7	2.5	4.2	7.7	11.0	14.3	17.5

Table 4: Filter times (in milliseconds) as a function of the number of active TCP/IP sessions with a uniform hash distribution.

Effect of Hash Collisions

The previous experiment measures the best case latency for MPF, where there are no hash collisions. The second experiment measures the worst case latency, where all sessions hash to the same hash bucket and the matching TCP session is the last one in the bucket. Table 5 and Figure 5 show the results.

# of sessions	1	2	6	10	20	40	60	80	100
warm cache, MPF [ms]	0.035	0.035	0.038	0.042	0.051	0.061	0.106	0.150	0.173
cold cache, MPF [ms]	0.153	0.157	0.164	0.172	0.191	0.224	0.270	0.306	0.347

Table 5: Filter times (in milliseconds) as a function of the number of active TCP/IP sessions with worst-case hash collisions.

While the processing time of MPF with one hash bucket grows linearly with the number of sessions, it grows much more slowly than BPF and CSPF (measured in the previous experiment) because MPF still executes fewer instructions that BPF because of the collapsed filters.

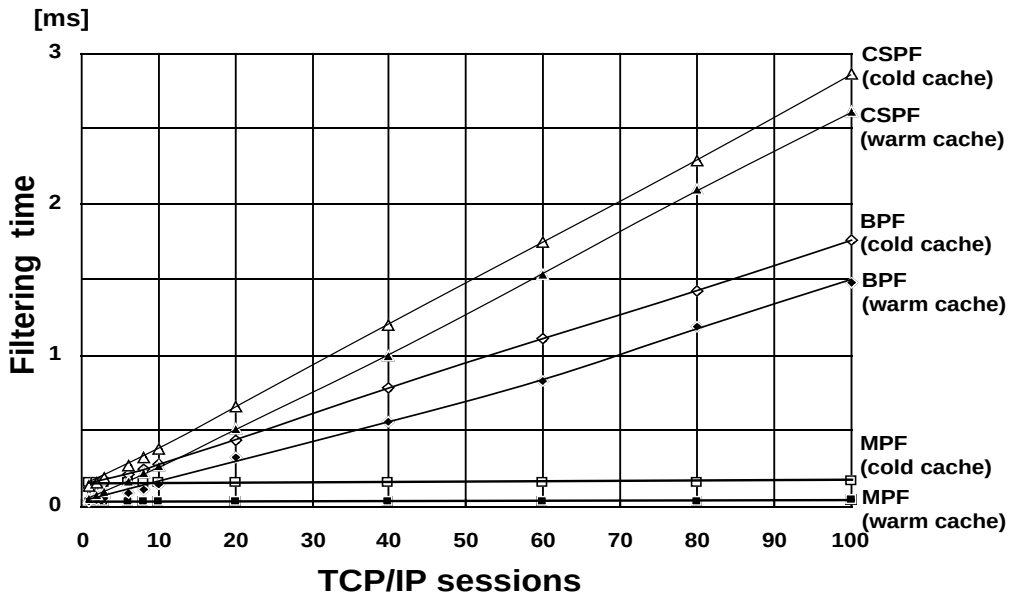

Figure 4: *Filtering time of the three filter mechanisms*

4.2 Filter Installation

The final two experiments measure the filter installation overhead for MPF.

Installing new sessions for the same protocol

The first experiment measures the time required to install a new session of an existing protocol, such as TCP. We expect that this will be the most common case in real systems since existing operating systems tend to use a single protocol family such as the ARPA Internet protocols for the DECNET protocol suite. We installed filter programs for the same protocol (TCP/IP), but different sessions, one by one. The results are shown in Figure 6.

When the number of installed programs is small, BPF and CSPF filter installation is about 60% faster than MPF, because MPF must compare the new filter program against each installed program. As the number of programs increases, the performance difference diminishes; with 100 sessions, the installation time is about the same for the three filter mechanisms.

The extra installation overhead is not excessive for a protocol such as TCP/IP because connection establishment requires one and a half network roundtrips.

Installing unrelated filter programs

The second experiment measures the time to install a new filter program that cannot be collapsed with any existing filters. This case occurs whenever a filter is installed that does not correspond to a protocol with a large number of active sessions. The filter programs installed were the same as in the previous experiment except that the protocol number was unique for each program. As a result, MPF tries to match each new filter with each existing filter, fails, and then prepares a separate hash table for the new filter.

Figure 7 shows the time to install a new filter program for each implementation. The installation time for MPF grows linearly with the number of filters while the installation time for BPF and CSPF remains constant. In practice, we expect that the number of protocols (and hence the number of uncollapsed filter programs) will be small.

5 Conclusions

MPF is a new packet filter mechanism that can efficiently dispatch small and large packets even in the existence of many sessions, making it suitable for per-task protocol processing. We have introduced a new match instruction which the packet filter mechanism can use as a hint to put the same protocol's programs together. This collapsing removes repetitive execution of the same code and also provides efficient associative lookup. MPF also supports

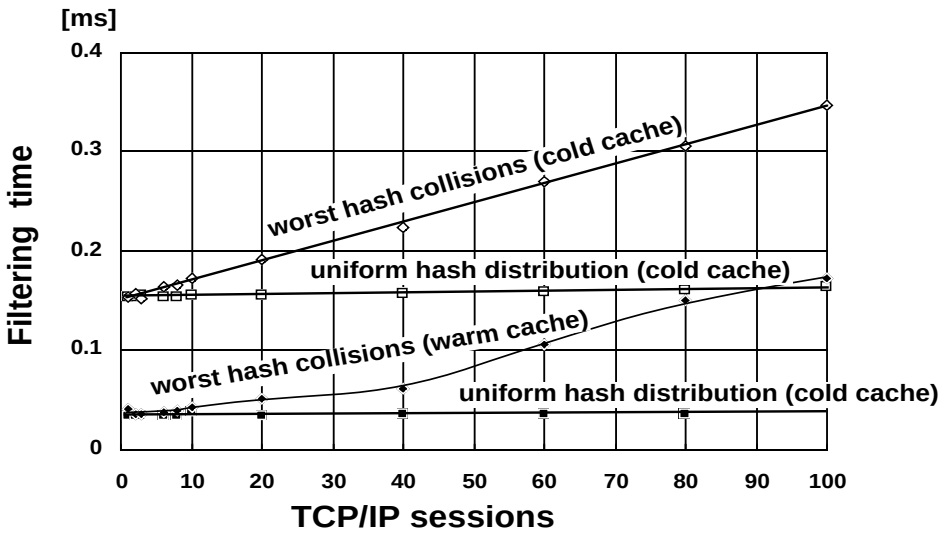

Figure 5: Effects of hash collisions. The curve from the previous experiment is shown for comparison.

new instructions to dispatch fragmented packets. Our implementation of MPF shows that it is 7.6 times faster for TCP/IP packet filtering than CSPF, and 4.3 times faster than BPF with only ten registered sessions. The total demultiplexing time in the kernel for TCP/IP improves by 50% over CSPF and 33% over BPF. Moreover, MPF's relative advantage increases with the number of sessions. The source code for MPF can be obtained through anonymous ftp as part of CMU's Mach 3.0 distribution at *cs.cmu.edu*.

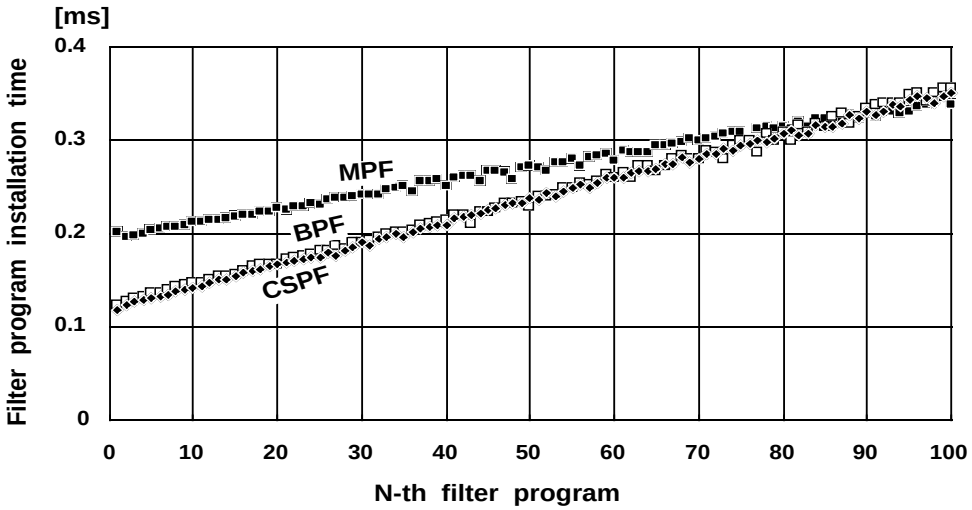

Figure 6: *Filter program installation time, for the same protocol*

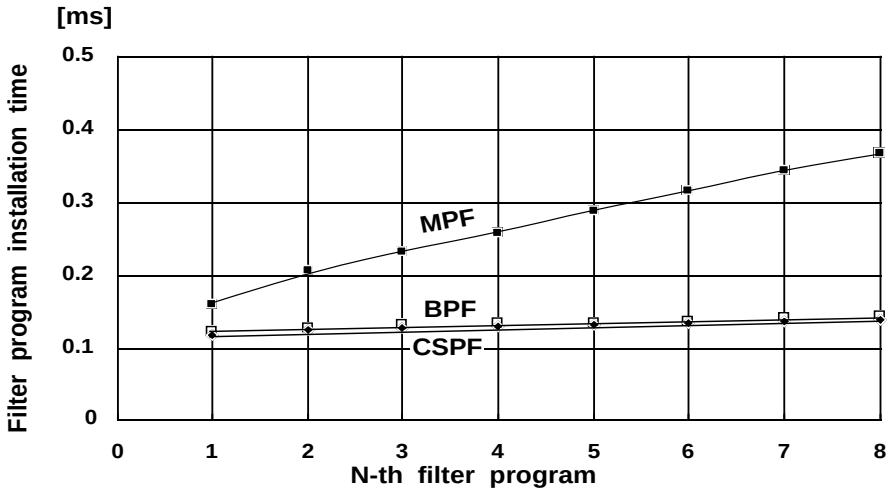

Figure 7: Filter program installation time, for different protocols.

References

[Accetta et al. 86] Accetta, M.J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M.W., "Mach: A New Kernel Foundation for UNIX Development", Proc. of the Summer 1986 USENIX Conference, pp.93-113, July 1986.

[Black et al. 92] Black, D. L., Golub, D. B., Julin, D. P., Rashid, R.F., Draves, R. P., Dean, R. W., Forin, A., Barrera, J., Tokuda, H., Malan, G., Bohman, D., "Microkernel Operating System Architecture and Mach", Proc. of the Microkernels and Other Kernel Architectures Workshop, pp. 11-30, April 1992.

[Maeda and Bershad 92] Maeda, C., and Bershad, B.N., "Network Performance for Microkernels", Proc. of the Third Workshop on Workstation Operating Systems, April 1992.

[Maeda and Bershad 93] Maeda, C., and Bershad, B.N., "Protocol Service Decomposition for High-Performance Networking", To appear in the Proc. of the 14th ACM Symposium on Operating Systems Principles, 1993.

[McCanne and Jacobson 93] McCanne, S., Jacobson, V., "The BSD Packet Filter: A New Architecture for User-level Packet Capture", Proc. of the Winter 1993 USENIX Conference, pp.259-269, January 1993.

[Mogul et al. 87] Mogul, J., Rashid, R., and Accetta, M., "The Packet Filter: An efficient Mechanism for User-level Network Code", Proc. of the 11th ACM Symposium on Operating Systems Principles, pp.39-51, 1987.

[Reynolds and Heller 91] Reynolds, F. and Heller, J., "Kernel Support for Network Protocol Servers", Proc. of the USENIX Mach Symposium, pp.149-162, November 1991.

[RFC791] Postel, J. B., "Internet Protocol", Request For Comments 791, September 1981.

[Sun 90] Sun Microsystems Inc., NIT(4P), SunOS4.1.1 Reference Manual, 1990.

Appendix

A BPF example program

```
        /*
         * P[i]: packet data at byte offset i.
         * M[i]: i-th word of the scratch memory.
         * Word = 4 Bytes, Half Word = 2 Bytes, Byte = 1 Byte.
         *
         * dst_addr: IP address of this host
         *           (destination IP address of this session)
         *
         * src_addr: source IP address of this session
         * src_port: source TCP port number of this session
         * dst_port: destination TCP port number of this session
         */

        begin                           ; BPF identifier
        ldh     P[OFF_ETHERTYPE]        ; A = ethertype
        jeq     #ETHERTYPE_IP, L1, Fail ; If not IP, fail.
L1:
        ld      P[OFF_DST_IP]           ; A = dst IP address
        jeq     #dst_addr, L2, Fail     ; If not from dst_addr, fail.
L2:
        ld      P[OFF_SRC_IP]           ; A = src IP address
        jeq     #src_addr, L3, Fail     ; If not from src_addr, fail.
L3:
        ldb     P[OFF_PROTO]            ; A = protocol
        jeq     #IPPROTO_TCP, L4, Fail  ; If not TCP, fail.
L4:
        ldh     P[OFF_FRAG]             ; A = Flags|Frag_offset
        and     #!Dont_Frag_Bit         ; Clear Don't Fragment bit
        jeq     #0, L5, Fail            ; If fragmented, fail
L5:
        ldx     4 * (P[OFF_IHL] & 0xf)  ; X = offset to TCP header

        ld      P[x + OFF_SRC_PORT]     ; A = src TCP port
        jeq     #src_port, L6, Fail     ; If not from src_port, fail.
L6:
        ld      P[x + OFF_DST_PORT]     ; A = dst TCP port
        jeq     #dst_port, Suc, Fail    ; If not to dst_port, fail.
Suc:
        ret #ALL                        ; Accept the whole packet.
Fail:
        ret #0                          ; Reject the packet.
```

B An example filter program that processes fragmented packets

```
/*
 * P[<x>]: Data <x> in the packet.
 * M[i]: i-th word of the scratch memory.
 * Presented as higher level code for clarity
 */

if (P[proto] == IP) {
    if (P[dest_addr] != dest_addr)      /* Not to me */
        return FAILURE;
    if (P[IP_proto] != TCP)
        return FAILURE;

    if (P[fragment_offset] == 0) {
        /* non-fragmented packet or the 1st fragment */

        M[0] = P[src_addr];
        M[1] = P[src_port];
        M[2] = P[dst_port];

        if (P[more_fragments]) {
            /* This packet is the 1st fragment */
```

```
            /*
             * Check if this 1st fragment is for this filter.
             */
            if (JMP_MATCH_IMM(3, #src_addr, #src_port, #dst_port)) {
                /*
                 * Yes. This is for this filter.
                 * Register the fragmentation information to dispatch
                 * remaining fragments.
                 */
                M[1] = P[fragment_ID];
                REGISTER_DATA (2, time_out);

                /* Return the whole packet */
                return ALL;
            } else {
                /* No. This is NOT for this filter. */
                return FAILURE;
            }
        } else {
            /*
             * This packet is a non-fragmented packet.
             * If it is for this filter, return the whole packet.
             * Otherwise, fail.
             */
            RET_MATCH_IMM (3, ,#src_addr, #src_port, #dst_port, ALL);
        }
    } else {
        /* 2nd or later fragmented packet */

        M[0] = P[src_addr];
        M[1] = P[fragment_ID];
        RET_MATCH_DATA (2, ALL);    /* If matched, return the whole packet */
        POSTPONE (time_out);    /* Otherwise postpone the filtering */
    }
} else {
    return FAILURE;
}
```

C CSPF example program

```
/*
 * Word = 2 Bytes
 * Note: CSPF is a stack-based word (2 Bytes) machine and there is no
 *       direct way to access a byte or a 4-byte data.
 */

pushword P[OFF_ETHERTYPE]       ; push ethertype
pushlit | cand                  ; if not IP, fail.
#ETHERTYPE_IP

pushword P[OFF_DST_IP_HI]       ; push higher half of dst IP addr.
pushlit | cand                  ; if not dst_addr_hi, fail.
#dst_addr_hi
pushword P[OFF_DST_IP_LO]       ; push lower half of dst IP addr.
pushlit | cand                  ; if not dst_addr_lo
#dst_addr_lo

pushword P[OFF_TTL_PROTO]       ; push word of TTL and PROTO
pushlit | and
#PROTO_MASK                     ; mask off TTL byte
pushlit | cand                  ; if not TCP, fail.
#IPPROTO_TCP

pushword P[OFF_Frag]            ; push Frag_flags|Frag_offset
pushlit | and                   ; Clear Don't Fragment bit
#!Dont_Frag_Bit
pushzero | cand                 ; If fragmented, fail.

pushword P[OFF_SRC_IP_HI]       ; push higher half of src IP addr.
pushlit | cand                  ; if not src_addr_hi, fail.
#src_addr_hi
pushword P[OFF_SRC_IP_LO]       ; push lower half of src IP addr.
pushlit | cand                  ; if not src_addr_lo
#src_addr_lo
```

```
pushword P[OFF_VER_IHL]         ; push word with IP Header Length
pushlit | and                   ; extract IHL
#IHL_MASK
pushlit | lsh                   ; convert it to word offset
#1                              ; (little endian machine)

pushlit | add
#OFF_IP                         ; skip a header from Ethernet.
pushstk + 0                     ; duplicate stack top

pushind                         ; push src TCP port
pushlit | cand                  ; if not to src_port, fail.
#src_port

pushlit | add                   ; get word offset to dst TCP port.
#OFF_DST_PORT
pushind                         ; push dst TCP port
pushlit | cand                  ; if not to dst_port, fail.
#dst_port
                                ; otherwise, accept the packet.
```