xPF: Packet Filtering for Low-Cost Network Monitoring

S. Ioannidis⋆ K. G. Anagnostakis⋆ J. Ioannidis† A. D. Keromytis‡

⋆CIS Department, University of Pennsylvania
{sotiris,anagnost}@dsl.cis.upenn.edu

†AT&T Labs – Research ‡CS Department, Columbia University
ji@research.att.com angelos@cs.columbia.edu

Abstract

The ever-increasing complexity in network infrastructures is making critical the demand for network monitoring tools. While the majority of network operators rely on low-cost open-source tools based on commodity hardware and operating systems, the increasing link speeds and complexity of network monitoring applications have revealed inefficiencies in the existing software organization, which may prohibit the use of such tools in high-speed networks. Although several new architectures have been proposed to address these problems, they require significant effort in re-engineering the existing body of applications.

In this paper we present an alternative approach that addresses the primary sources of inefficiency without significantly altering the software structure. Specifically, we enhance the computational model of the Berkeley Packet Filter (BPF) to move much of the processing associated with monitoring into the kernel, thereby removing the overhead associated with context switching between kernel and applications. The resulting packet filter, called xPF, allows new tools to be more efficiently implemented and existing tools to be easily optimized for high-speed networks. We present the design and implementation of xPF as well as several example applications that demonstrate the efficiency of our approach.

1 Introduction

Network monitoring is becoming an increasingly important function in a modern IP-based network infrastructure. This is mostly due to the stateless nature of the IP service, which requires appropriate control loops to be implemented by observing and responding to network behavior. For the majority of networks, such functions are implemented using commodity components: PC workstations or servers running free operating systems and open-source monitoring tools [2, 10, 12, 15, 22, 1, 21, 5]. Most of these tools rely on the Berkeley Packet Filter (BPF) facility [18], which allows them to capture packets from the network interface.

However, as the speed of network links continues to increase, the use of commodity components and BPF is becoming inefficient [9, 20, 23]. Additionally, the increasing complexity and sophistication of network monitoring tools further adds to the processing burden of a monitoring system. Finally, BPF performance degrades rapidly as the number of concurrent applications increases [5]. There have been several studies on optimizing the packet filter [8], efficient network interface access [20], as well as system architectures designed specifically for network monitoring [6, 17, 15, 4]. Most network monitoring applications, however, do not need elaborate packet filters (and therefore cannot take advantage of the aforementioned optimizations) to demultiplex packet streams into different processes; instead, a single process receives all the packets and does its own packet handling. On the other hand, monitoring-specific system architectures provide a rich and complex set of functions, but require significant effort for re-implementing the existing body of tools. Finally, these architectures are not widely available, and are thus less likely to be mature enough to be adopted in the near future.

In this study we attempt to provide efficient network monitoring capabilities with minimal changes to existing infrastructure. This is achieved by using BPF itself as the engine for *executing monitoring applications*, rather than a mechanism for demultiplexing packets to applications in user-space. To make this possible, we enhance BPF by enriching the computing model it provides. Specifically, we introduce persistent memory support and remove the restriction of forward-only branches. In this way, monitoring applications execute inside the packet filter and only need to communicate with user-space code in order to perform functions such as storing or communicating data to the user or higher-level applications. The resulting system achieves high performance and low system overheads without altering the general system structure. At the same time, the simplicity, portability and compatibility with existing tools and the appealing maturity and stability of existing infrastructure are retained.

2 Design

The principal goal of our design is to reduce the context switching overheads associated with the BPF approach by moving as much of the packet processing as possible into the filter machine. In essence, we use BPF as a virtual machine to execute monitoring applications inside the kernel, rather than as a tool for demultiplexing packets to applications. There are two primary restrictions in the current design of BPF that do not allow this. The first one is that filters cannot maintain state across invocations (remember that filter code is invoked with every packet arrival). This makes it hard to write any BPF "programs" that maintain state, unless such state is communicated to user-space with every filter invocation. The second restriction is the lack of backward jump support, which significantly limits the development of more sophisticated filters. For example, looking up a value in a table, or any program that uses loops cannot be implemented in BPF. We have extended the BPF filter machine to remove these two restrictions; we provide persistent memory for each filter, mechanisms for memory management, and support for safely allowing backward jumps using lightweight run-time checks.

Adding memory The original BPF filter architecture involves a very simple filter interpreter that executes assembly-like instructions; there is an accumulator, an index register, an implicit program counter, some scratch memory, and, of course, the actual filter program. The scratch memory in the original BPF code is small (usually 64 bytes), as it is only meant to be used during a single filter execution. We have added the ability to have memory that is maintained between subsequent calls to the BPF filter code, so that state can be kept.

This persistent memory is accessed using the age-old concept of *bank switching*. Two of the unused opcodes in the original BPF interpreter are redefined to switch the active memory bank to either the original scratch memory area, or the persistent memory; their mnemonics are BPF_BSS and BPF_BSP (for "Bank Switch to Scratch" and "Bank Switch to Persistent," respectively). This allows the same instructions to be used for accessing the memory, with all the addressing modes provided by the BPF code. These, however, are not enough; we add indexed load and store instructions; thus, when the addressing mode BPF_NDX is used, the memory location pointed to by the contents of the X register, offset by the immediate operand, are recalled into or stored from the accumulator. Naturally, before the memory is accessed, the effective address is checked to verify that it lies within the bounds of what was allocated, or within the bounds of the scratch memory as in the original code. Since packet filter code can be written that effectively discards the copy of the packet it received, multiple invocations of the filter code can happen and computations performed based on the contents of each packet without having to pay the penalty of a context switch to user-mode. Since there is no difference in the execution speed between scratch memory accesses and persistent memory accesses, and the scratch memory is not initialized to anything upon filter invocation anyway, there is no need to keep switching between the two; if a filter wants to use persistent memory, it simply bank-switches to it at the beginning of its execution; some part may be used as scratch, and some other part as permanent storage. The code that uses the scratch memory is retained for compatibility with older programs that do not use the persistent memory feature, so that some scratch memory will always be present without having to explicitly allocate any.

Persistent memory is allocated for each open instance of the BPF device (effectively, once per file descriptor since BPF file descriptors are rarely *dup()*-ed). This memory is managed with a set of *ioctl()* system calls. The first BIOCMALLOC call allocates a block of memory and returns a *handle*, a small integer that is used by subsequent calls to refer to the allocated block of memory. Two calls, BIOCMREAD and BIOCMWRITE are used to read from and write to the block of memory specified by a handle which is also passed as an argument. Finally, the BIOCMSWITCH call sets the active block of memory that is to be used by the packet filter. There are three options for this call: the newly switched-to block can be left untouched (to keep whatever contents it had since the last time it was activated), set to zeroes, or get a copy of the contents of the currently-active block. Note that only one such block can be accessed by the BPF code at any time; the BPF code's bank-switching selects between the internal scratch and the currently-active persistent block. The user-level switching selects between alternate persistent blocks. This is useful because the switching happens atomically and definitely not while a filter is executing; thus we can have a filter executing and populating statistics or timestamps in one block while a user level program is manipulating another, effectively having the equivalent of double-buffering.

The persistent storage can be used for a number of purposes. The most obvious is to use it for keeping statistics counters, timestamps, *etc.*, which can be periodically read from a user-level program. Another use is to use part of it as a look-up table, since the table can be pre-computed by the user-level program and loaded into the filter. Look-up tables can also be used to compute complicated functions.

Eliminating branch restrictions Bounding execution time in BPF is ensured by eliminating backward jumps. This has the advantage of providing us with an upper bound for the execution time: linear to the length of the program. However, such a programming model is hard to program in and rather limiting. Another possible approach is to execute each installed filter as a kernel thread and context

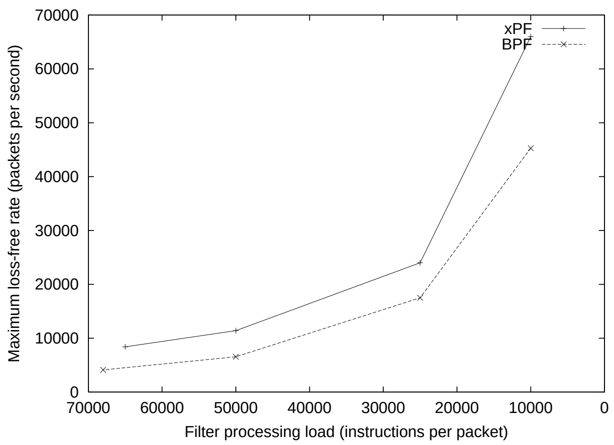

Figure 1: **Performance of BPF *vs.* xPF**

switch between threads when they exceed their allocated time slot. This approach is too heavy weight for our purposes, which is round-robin execution of monitoring functions on incoming packets.

Notice that the filter is effectively an interpreted virtual machine, with an outer loop doing an "instruction fetch" at the current "program counter" and interpreting it. Thus, our solution is to simply have a counter of how many BPF "instructions" have been executed since the filter started running, and take an "exception" when the number is exceeded. The exception handler jumps to a predefined location in the filter code, where a cleanup operation can be performed before terminating the filter; during the execution of the exception handler, only forward jumps are allowed, so that *its* execution time is finite.

3 Experiments

A prototype of xPF has been implemented as part of the OpenBSD [3] operating system. We demonstrate the performance benefits of xPF and describe three sample filter programs for NetFlow-like accounting, trajectory sampling, and round-trip time estimation respectively. These modules were originally implemented on the FLAME architecture [4] and were re-implemented in xPF.

3.1 Measurements

For outlining the performance and scalability of xPF, we have implemented a simple benchmark application that simply counts packets received by the filter. In standard BPF, every packet needs to be passed to the application to increment the counter. Using xPF, the counter is incremented inside the filter, eliminating the need for context switching for each packet. In addition to incrementing the counter, we instruct the filter to consume an arbitrary number of processing cycles, in order to infer the system monitoring capacity, as we describe next.

Our test platform consists of two 1GHz Pentium III PCs with 256MB SDRAM, and 1 Gbit/s Ethernet interfaces (NetGear GA620 32-bit PCI based with Alteon chipset). One PC is used as the traffic generator and the other one runs xPF, the filter described above and instrumentation tools. We measure the *maximum loss-free rate* of traffic that the system can receive and process for different processing loads per packet. The results are presented in Figure 1, showing the improvement of xPF which becomes larger as the traffic rate increases, demonstrating the scalability of our approach.

3.2 Applications

NetFlow-like accounting Implementing an accounting package such as NetraMet [10] for providing per-flow accounting similar to NetFlow [11] using BPF requires every packet to be copied to user-space. The user-space part would then take care of looking up the flow associated with the packet, and incrementing the appropriate counters. It is possible to perform all computation inside the kernel, by instrumenting xPF with the appropriate functionality, for maintaining the flow database, looking up packets and incrementing counters. In certain intervals, the packet filter would need to call the user-space function to collect the current data and pass them on to the user for further processing or storage.

Trajectory sampling Trajectory sampling, developed by Duffield and Grossglauser[13], is a technique for coordinated sampling of traffic across multiple measurement points, effectively providing information on the spatial flow of traffic through a network. The key idea is to sample packets based on a hash function over the invariant packet content (*e.g.,* excluding fields such as the TTL value that change from hop to hop) so that the same packet will be sampled on all measured links. Network operators can use this technique to measure traffic load, traffic mix, one-way delay and delay variation between ingress and egress points, yielding important information for traffic engineering and other network management functions. Although the technique is simple to implement, we are not aware of any monitoring system or router implementing it at this time.

We have implemented trajectory sampling as an xPF filter that works as follows. First, we compute a hash function $h(x) = \phi(x) mod A$ on the invariant part $\phi(x)$ of the packet. If $h(x) > B$, where $B < A$ controls the sampling rate, the packet is not processed further. If $h(x) < B$ we compute a second hash function $g(x)$ on the packet header that, with high probability, uniquely identifies a flow with a label (*e.g.,* TCP sequence numbers are ignored at this stage, since they change over the lifetime of the TCP connection). If this is a new flow, we create an entry into a hash table, storing flow information (such as IP address, protocol, port numbers *etc.*). Additionally, we store a timestamp along with $h(x)$ into a separate data structure. If the flow already exists, we do not need to store all the information on the flow, so we just log the packet. For the purpose of this study we did not implement a mechanism to transfer logs from the kernel to a user-level module or management system; at the end of the experiment the logs are stored in a file for analysis.

Passive RTT estimator We have implemented a simple application for measuring the round-trip delays observed over a network link by TCP connections. Round-trip delays are an important metric for understanding end-to-end performance, mostly due to its central role in TCP congestion control[16]. Additionally, measuring the round-trip times observed by users over a specific ISP provides a reasonable indication of the quality of the service provider's infrastructure, as well as its connectivity to the rest of the Internet. Finally, observations on the evolution of round-trip delays over time can be used to detect network anomalies on shorter time scales, or to determine the improvement (or deterioration) of service quality over longer periods of time. For example, an operator can use this tool to detect service degradation or routing failures in an upstream provider, and take appropriate measures (*e.g.,* redirecting traffic to a backup provider), or simply have answers for user questions.

The implementation of this application is fairly simple and efficient. We watch for TCP SYN packets passing through the monitored link, indicating a new connection request, and then watch for the matching TCP ACK packet in the same direction. The difference in time between these two packets is a reasonable approximation of the round-trip time between the two ends of the connection. For every SYN packet received, we store a timestamp into a hashtable. As the first ACK after a SYN usually has a sequence number which is the SYN packet's sequence number plus one, this number is used as the key for hashing. Thus, in addition to watching for SYN packets, the application only needs to look into the hash table for every ACK received. The hash table can be appropriately sized depending on the number of flows and the required level of accuracy.

4 Related Work

The concept of a packet filter was first proposed by Mogul *et al.* more than ten years ago in [19]. There has been extensive research in the area since then, trying to refine the filtering model. The most widely used packet filter is BPF [18] which uses a register-model instruction set, unlike the stack machine model used by [19]. Each filter is run on every incoming packet which imposes high overhead on user-level programs using it. The Mach Packet Filter (MPF) [24] enhances the BPF model by demultiplexing filter specifications to recognize when two filters use similar patterns. If MPF detects similarities, it merges the predicates forming a single predicate. PathFinder [7] improved on MPF with a re-designed filtering engine that was better matched to the pattern-matching transformation. Another approach, DPF [14], employs dynamic-code generation to exploit run-time knowledge to achieve even better performance. Finally, BPF+ [8] allows for packet filters to be expressed in a high level language and be compiled down to native code using just-in-time compilation. It utilizes a verifier to guarantee the safety properties of the resulting code.

There is a fundamental difference in the goals of our enhanced BPF and in the scope of the above efforts. The enhancements to BPF proposed in this paper provide capabilities that are specific to the application domain of network monitoring, where the BPF machine is used more to *compute* rather than *filter*. Hence, although some specific optimizations such as the just-in-time compiler of [8] could easily be incorporated to make network monitoring applications even more efficient, the above efforts are mostly orthogonal to our enhanced BPF design.

5 Summary and concluding remarks

We have shown that efficient network monitoring is possible, to some extent, without introducing complex new infrastructure. This is accomplished using simple extensions to the Berkeley Packet Filter (BPF) that allow the filter mechanism to perform monitoring functions. In this way, monitoring functions can safely and efficiently execute in the system kernel, hereby eliminating the system overheads associated with context switching. xPF can be used to support existing and new applications while allowing commodity systems to satisfy the growing demand for network monitoring at increasing network speeds.

Acknowledgements

This work was supported by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795, and DARPA under Contracts F39502-99-1-0512-MOD P0001 and F30602-01-2-0537. We would also like to thank the anonymous reviewers for their constructive review, and Jonathan Smith for his valuable comments and suggestions throughout the course of this work.

References

[1] EtherReal. http://www.etherreal.org/.

[2] Tcpdump/Libpcap Web page. http://www.tcpdump.org/.

[3] The OpenBSD Operating System. http://www.openbsd.org/.

[4] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proceedings of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2002.

[5] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, and J. M. Smith. Practical network applications on a leightweight active management environment. In *Proceedings of the 3rd IFIP International Working Conference on Active Networks (IWAN)*, October 2001.

[6] J. Apisdorf, k claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, Affordable, High Performance Statistics Collection. In *Proceedings of the 1996 LISA X Conference*, October 1996.

[7] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 115–123, November 1994.

[8] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proceedings of SIGCOMM*, pages 123–134, August 1999.

[9] B. Bershad et al. Extensibility, safety and performance in the SPIN operating system. In *Proc. 15-th Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[10] N. Brownlee. Traffic Flow Measurement: Experiences with NeTraMet. RFC2123, March 1997.

[11] Cisco Corporation. NetFlow services and applications. http://www.cisco.com/.

[12] L. Deri and S. Suin. Effective Traffic Measurement using ntop. *IEEE Communications Magazine*, 38(5):138–145, May 2000.

[13] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proceedings of SIGCOMM*, pages 271–282. August 2000.

[14] D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of SIGCOMM*, pages 53–59, August 1996.

[15] K. Keys, D. Moore, R. Koga, E. Lagache, M. Tesch, and k claffy. The Architecture of CoralReef: An Internet Traffic Monitoring Software Suite. In *PAM 2001 Workshop*, April 2001.

[16] T. V. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5(3):336–350, June 1997.

[17] G. R. Malan and F. Jahanian. An Extensible Probe Architecture for Network Protocol Performance Measurement. In *Proceedings of SIGCOMM*, September 1998.

[18] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–270, January 1993.

[19] J. Mogul, R. Rashid, and M. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[20] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[21] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 1999.

[22] M. Roesch. Snort - Leightweight Intrusion Detection for Networks. In *Proceedings of the 1999 LISA Conference*, November 1999.

[23] Steve Muir and Jonathan Smith. AsyMOS - An Asymetric Multiprocessor Operating System. In *Proceedings of the 1998 IEEE 1st Conference on Open Architectures and Network Programming (OPENARCH'98)*, April 1998.

[24] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the 1994 Winter Usenix Conference*, pages 153–165, January 1994.