FFPF: Fairly Fast Packet Filters

Herbert Bos[†], Willem de Bruijn[†], Mihai Cristea[∗], Trung Nguyen[∗], Georgios Portokalidis[∗]

[†]*Vrije Universiteit Amsterdam, The Netherlands*
{herbertb, wdb}@few.vu.nl
[∗]*Universiteit Leiden, The Netherlands*
{cristea,tnguyen,gportoka}@liacs.nl

Abstract

FFPF is a network monitoring framework designed for three things: speed (handling high link rates), scalability (ability to handle multiple applications) and flexibility. Multiple applications that need to access overlapping sets of packets may share their packet buffers, thus avoiding a packet copy to each individual application that needs it. In addition, context switching and copies across the kernel boundary are minimised by handling most processing in the kernel or on the network card and by memory mapping all buffers to userspace, respectively. For these reasons, FFPF has superior performance compared to existing approaches such as BSD packet filters, and especially shines when multiple monitoring applications execute simultaneously. Flexibility is achieved by allowing expressions written in different languages to be connected to form complex processing graphs (not unlike UNIX processes can be connected to create complex behaviour using pipes). Moreover, FFPF explicitly supports extensibility by allowing new functionality to be loaded at runtime. By also implementing the popular `pcap` packet capture library on FFPF, we have ensured backward compatibility with many existing tools, while at the same time giving the applications a signficant performance boost.

1 Introduction

Most network monitoring tools in use today were designed for low-speed networks under the assumption that computing speed compares favourably to network speed. In such environments, the costs of copying packets to user space prior to processing them are acceptable. In today's networks, this assumption is no longer true. The number of cycles available to process a packet before the next one arrives (the cycle budget) is minimal. The situation is even worse if multiple monitoring applications are active simultaneously, which is increasingly common as monitors are used for traffic engineering, SLA monitoring, intrusion detection, steering schedulers in GRID computing, etc. Moreover, the processing requirements are increasing. Consider the following monitoring applications:

1. An intrusion detection system (IDS) checks the payload of every packet for worm signatures [31].

2. An application based on the 'Coralreef' suite keeps statistics for the ten most active flows [21].

3. A tool is interested in monitoring flows for which the port numbers are not known *a priori*. Such flows are found, for example, in peer-to-peer and H.323 multimedia flows where the control channels use well-known port numbers, while the data transfer takes place on dynamically assigned ports [32].

4. Multiple monitoring applications (e.g. `snort`, `tcpdump`, etc.) access identical or overlapping sets of packets.

In high-speed networks, none of these applications are catered to in the kernel in a satisfactory manner by existing solutions such as BPF, the BSD Packet Filter [25], and its Linux cousin, the Linux Socket Filter (LSF). In our view, they require a rethinking of the way packets are handled in the operating system.

In this paper, we discuss the implementation of the fairly fast packet filter (FFPF). FFPF introduces a novel packet processing architecture that provides a solution for filtering and classification at high speeds. FFPF has three ambitious goals: speed (high rates), scalability (in number of applications) and flexibility. Speed and scalability are achieved by performing complex processing either in the kernel or on a network processor, and by minimising copying and context switches. Flexibility is considered equally important, and for this reason, FFPF is explicitly extensible with native code and allows complex behaviour to be constructed from simple components in various ways.

On the one hand, FFPF is designed as an alternative to kernel packet filters such as CSPF [26], BPF [25], mmdump [32], and xPF [19]. All of these approaches rely on copying many packets to userspace for complex processing (such as scanning the packets for intrusion attempts). In contrast, FFPF permits processing at lower levels and may require as few as zero copies (depending on the configuration) while minimising context switches. On the other hand, the FFPF framework allows one to add support for any of the above approaches.

FFPF is not meant to compete with monitoring suites like Coralreef that operate at a higher level and provide libraries, applications and drivers to analyse data [21]. Also, unlike MPF [34], Pathfinder [3], DPF [17] and BPF+ [5], the goal of this research is not to optimise filter expressions. Indeed, the FFPF framework itself is language *neutral* and currently supports five different filter languages. One of these languages is BPF, and an implementation of `libpcap`[1] exists, which ensures not only that FFPF is backward compatible with many popular tools (e.g., `tcpdump`, `ntop`, `snort`, etc. [31]), but also that these tools get a significant performance boost (see Section 5). Better still, FFPF allows users to mix and match packet functions written in different languages.

To take full advantage of all features offered by FFPF, we implemented two languages from scratch: FPL-1 (FFPF Packet Language 1) and its successor, FPL-2. The main difference between the two is that FPL-1 runs in an interpreter, while FPL-2 code is compiled to fully optimised native code.

The aim of FFPF is to provide a complete, fast, and safe packet handling architecture that caters to all monitoring applications in existence today and provides extensibility for future applications. Since its first release in May 2003 we have constantly improved the code and gained a fair amount of experience in monitoring. We now feel that the architecture has stabilised and the ideas are applicable to systems other than FFPF as well. FFPF is available from `ffpf.sourceforge.net`. Some contributions of this paper are summarised below.

1. We generalise the concept of a 'flow' to a stream of packets that matches arbitrary user criteria.

2. Context switching and packet copying are reduced (up to 'zero copy').

3. We introduce the concept of a 'flow group', a group of applications that *share* a common packet buffer.

4. Complex processing is possible in the kernel or NIC (reducing the number of packets that must be sent up to userspace), while Unix-style filter 'pipes' allow for building complex flow graphs.

5. Persistent storage for flow-specific state (e.g., counters) is added, allowing filters to generate statistics, handle flows with dynamic ports, etc.

To our knowledge, few solutions exist that support *any* of these features and none that provide *all* in a single, intuitive, architecture. In this paper, we present the FFPF architecture and its implementation in the Linux kernel. The remainder of this paper is organised as follows. In Section 2, a high-level overview of the FFPF architecture is presented. In Section 3, implementation details are discussed. A separate section, Section 4 is devoted to the implementation of FFPF on the IXP1200. FFPF is evaluated in Section 5. Related work is discussed throughout the text and summarised in Section 6. Conclusions and future work are presented in Section 7.

2   FFPF high-level overview

The FFPF framework can be used in userspace, the kernel, the IXP1200 network processor, or a combination of the above. As network processors are not yet widely used, and (pure) userspace FFPF does not offer many speed advantages, the kernel version is currently the most popular. For this reason, we use FFPF-kernel to explain the architecture, and describe the userspace and network processor versions later. The main components are illustrated in Figure (1.a).

A key concept in FFPF is the notion of a *flow* which is different from what is traditionally thought of as a flow (e.g., a 'TCP flow'). It may be thought of as a generalized socket: a flow is 'created' and 'closed' by an application and delivers a stream of packets, where the packets match arbitrary user criteria (e.g., "all UDP and TCP packets sent to port 554", or "all UDP packets containing the CodeRed worm plus all TCP SYN packets"). The flow may also provide other application-specific information (e.g., traffic statistics).
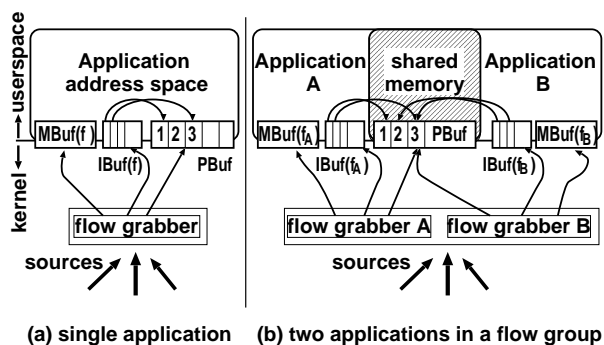

(a) single application   (b) two applications in a flow group

Figure 1: The FFPF architecture

A flow is captured by a *flow grabber*. For now, con-

[1] http://www.tcpdump.org/

sider a flow grabber to be a filter that passes just the information (packets, statistics) in which the user is interested. Packets arrive in the system via one or more packet sources. Examples of packet sources include: (a) a network driver that interacts with a dumb NIC, (b) a smart NIC that interacts with FFPF directly, or (c) a higher-layer abstraction in the operating system that hides device-specific issues. A flow grabber receives the packets and if they correspond to its flow, stores them in a circular packet buffer known as $PBuf$. In addition, it places a pointer to this packet in a second circular buffer, known as the index buffer, or $IBuf$. Applications use the pointers in $IBuf$ to find packets in $PBuf$.

The reason for using *two* buffers for capturing a flow is that while $IBuf$ is specific to a flow, $PBuf$ is *shared*. If the application opens two flows, there will be just one $PBuf$ and two $IBuf$s. If the flows are 'overlapping' (i.e., some packets in flow$_a$ are also in flow$_b$), only one copy of each packet will be in $PBuf$. However, if a packet is in both flows, a pointer to it is placed in both $IBuf$s. In other words, we do not copy packets to individual flows. Moreover, the buffers are memory mapped, so we do not copy between kernel and userspace either. We show later how $PBuf$ can also be shared by multiple applications (as sketched in Figure (1.b)). Using memory mapping to avoid copying is a known technique, also used in monitoring solutions like DAG and SCAMPI [10, 30]. Edwards et al. also give userspace applications direct control over packet buffers, but provide an explicit API to access the buffers rather than memory mapping [15].

Thus far, we have assumed that a flow grabber is equivalent to a filter. In reality, a flow grabber can be a complex graph of interconnected filters, where a filter is defined as an element that takes a stream of packets as input and returns a (possibly empty) subset of this stream as output. In addition, a filter may provide arbitrary information about the traffic, e.g., statistics, intrusion alerts, etc. For this purpose, every filter has an associated $MBuf$ (also memory mapped), which is a buffer that is used to produce results for applications, or to keep persistent state. It can also be used by the application to pass configuration parameters to the filter. For instance, in case of a 'blacklist filter' the application may store the addresses of the blacklist in $MBuf$. Note that the ability to perform more complex processing than just filtering, helps to reduce context switches, e.g., because applications that are interested in periodic statistics only and not in the packets themselves need not be scheduled for packet processing.

In later sections, we show that FFPF is language neutral, so that, for instance, BSD packet filters can be combined with filters written in other languages. In fact, the filters in a flow grabber are simple instantiations of filter *classes*, one of which may be the class of BPF filters. In addition to existing languages like BPF, we support two new languages (see Section 3.3) that are explicitly designed to exploit all features offered by FFPF. Among other things, they provide extensibility of the FFPF framework by their ability to call 'external functions' (provided these functions were previously registered with FFPF). External functions commonly contain highly optimised native or even hardware implementations of operations that are too expensive to execute in a 'safe' language (e.g., pattern matching, generating MD5 message digests).

We have covered most aspects of FFPF that are relevant if a single monitoring application is active. It is now time to consider what happens if multiple applications are present. For this purpose, we introduce a new concept, called the *flow group*. A flow group is a set of applications with the same *access rights* to packets, i.e., if one application is allowed to read a packet, all others in the same group may also access it. Flow groups are again used to minimise packet copying. Applications in the same group *share* a common $PBuf$. $PBuf$ contains all packets for which one or more applications in the group have expressed interest. This is illustrated in Figure (1.b). If more than one group express interest in the packet, it is copied once per group, unlike existing approaches (such as BPF/LSF) which copy the packet to each application separately. This makes FFPF cheaper than other solutions when supporting multiple applications. In the current implementation, the flow group is determined by group id. In the future, we plan to provide applications with more explicit control over flow groups.

We see that FFPF demultiplexes packets to their respective flows *early*, i.e., well before they are processed by the kernel protocol stack. This is a tried technique that is also used in projects like LRP [14]. Unlike LRP, however, we do not place the packets themselves on application-specific queues, but only the corresponding pointers. Thus, it is possible to avoid copying both for demultiplexing purposes and for crossing the protection domain boundaries.

2.1 Receiving packets in a flow

An application may be interested in multiple flows. Flows are captured from a raw input stream in four steps. Firstly, a flow handle is created with the `flow_create()` operation. Creating a flow handle sets up a user-space data structure which is used as an identifier in all future operations on the flow, but does not result in any packets being captured. Secondly, the flow handle structure is *populated* using the `flow_populate()` operation by specifying for instance the graph of connected filters, callback functions and other parameters to be associated with the flow. The

result is a *flow definition* in user space consisting of a graph of filters that will capture the flow, associated callbacks, etc. Thirdly, the flow definition is used as blue print to *instantiate* a *'flow grabber'* which is done by calling the `flow_instantiate()` operation. Only at instantiation time are the filters that capture the flow instantiated and connected, provided the flow definition passes the *authorisation control* check (Section 3.4). Fourthly, an instantiated flow grabber by itself still does not capture packets; the flow grabber first needs to be *activated*. Conversely, an activated flow can be *paused* (and subsequently re-activated). Flow activation and pausing is performed using the `flow_activate()` and `flow_pause()` operations. Finally, a flow can be *closed* (`flow_close()`). When a flow is closed (or the corresponding application crashes), all flow state is destroyed. In the remainder of this paper, we will use the term 'flow' to refer both to the flow grabber (the code in the kernel that captures the flow), and to the packets captured by the flow grabber (the real 'flow'), except where the distinction is important.

Instantiation is a separate step, because the flow specification is sent *in its entirety* to authorisation control, so that we can enforce that a packet function f (e.g., payload scanning) be allowed if and only if another function g (e.g., a filter passing only traffic from a specific subnet) is applied before (or after) f. Flow activation is also a separate step, as it gives administrators more accurate control over the start time (flow activation is more lightweight than flow instantiation).

2.2 Filter expressions

FFPF is language neutral, which means that different languages may be mixed. As mentioned earlier, we currently support five languages: BPF, FPL-1, FPL-2, C, and OKE-Cyclone. Support for C is limited to root users. The nature of the other languages will be discussed in more detail in Section 3. Presently, we only sketch how multiple languages are supported by the framework.

Figure (2.a) shows an example with two simplified flow definitions, for flows A and B, respectively. The grabber for flow A scans web traffic for the occurrence of a worm signature and saves the IP source and destination addresses of all infected packets. In case the signature was not encountered before, the packet is also handed to the application. Flow grabber B counts the number of fragments in web traffic. The first fragment of each fragmented packet is passed to the application.

There are a few things that we should notice. First, one of these applications is fairly complex, performing a full payload scan, while the other shows how state is kept regardless of whether a packet itself is sent to userspace. It is difficult to receive these flows efficiently using existing packet filtering frameworks, because they either don't allow complex processing in the kernel, or do not keep persistent state, or both. Second, both flows may end up grabbing the same packets. Third, the processing in both flows is partly overlapping: they both work on HTTP packets, which means that they first check whether the packets are TCP/IP with destination port 80 (first block in Figure 2). Fourth, as fragmentation is rare and few packets contain the CodeRed worm, in the common case there is no need for the monitoring application to get involved at all.

Figure (2.a) shows how these two flows can be accommodated. A common BPF filter selecting HTTP/TCP/IP packets is shared by both flows. They are connected to the flow-specific parts of the data paths. As shown in the figure, the data paths are made up of small components written in different languages. The constituent filters are connected in a fashion similar to UNIX pipes. Moreover, a pipe may be 'split' (i.e., sent to multiple other pipes, as shown in the figure) and multiple pipes may even be 'joined'. Again, in UNIX fashion, the framework allows applications to create complex filter structures using simple components. A difference with UNIX pipes, however, is the method of connection: FFPF automatically recognises overlapping requests and merges the respective filters, thereby also taking care of all component interconnects.

Each filter has its own $IBuf$, and $MBuf$, and, once connected to a packet source, may be used as a 'flow grabber' in its own right (just like a stage in a UNIX pipe is itself an application). Filters may read the $MBuf$ of other filters in their flow group (although we have not yet implemented synchronisation primitives to prevent races). In case the same $MBuf$ needs to be written by multiple filters, the solution is to use function-like *filter calls* supported by FPL-1 and FPL-2, rather than pipe-like *filter concatenation* discussed so far. For filter call semantics, a filter is called *explicitly* as an external function by a statement in an FPL expression, rather than implicitly in a concatenated pipe. An explicit call will execute the target filter expression with the calling filter's $IBuf$ and $MBuf$. An example is shown in Figure (2.b), where a first filter call creates a hash table with counters for each TCP flow, while a second filter call scans the hash table for the top-10 most active flows. Both access the same memory area.

2.3 Construction of filter graphs by users

FFPF comes with a few constructs to build complex graphs out of individual filters. While the constructs can be used by means of a library, they are also supported by a simple command-line tool called `ffpf-flow`. For example, pronouncing the construct '`->`' as 'connects to'
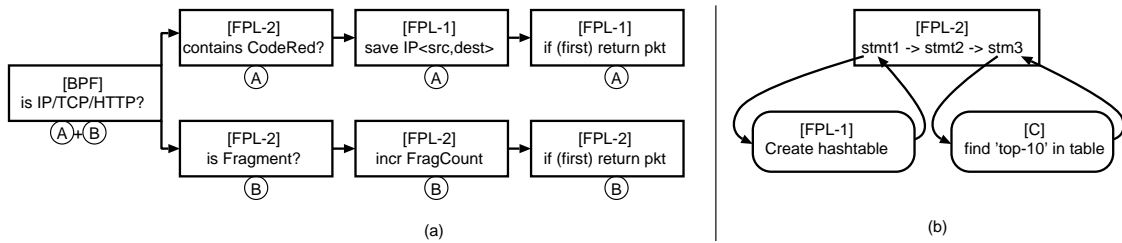
Figure 2: (a) combining different languages in two flows (A and B), (b) calling external functions from a single flow

and '|' as 'in parallel with', the command below captures two different flows:

```
./ffpf-flow \
  "(device,eth0) | (device,eth1) -> (sampler,2,4) -> \
      (FPL-2,"...") | (BPF,"...") -> (bytecount,,8)"
  "(device, eth0) -> (sampler,2,4) -> (BPF,"...") \
      -> (packetcount,,8)"
```

The top flow specification indicates that the grabber should capture packets from devices `eth0` and `eth1`, and pass them to a sampler that captures one in two packets and requires four bytes of $MBuf$. Next, sampled packets are sent both to an FPL-2 filter and to a BPF filter. These filters execute user-specified filter expressions (indicated by '...'), and in this example require no $MBuf$. All packets that pass these filters are sent to a bytecount 'filter' which stores the byte count statistic in in $MBuf$ in an eight byte counter. The counter can be read directly from userspace, while the packets themselves are not passed to the monitoring application. The second flow has a prefic of two 'filters' in common with the first expression (devices are treated as filters in FFPF), but now the packets are forwarded to a different BPF filter, and from there to a packet counter.

As a by-product, FFPF generates a graphical representation of the entire filter-graph. A graph for the two flows above is shown in Figure 3. For illustration purposes, the graph shows few details. We just show (a) the configuration of the filter graph as instantiated by the users (the ovals at the top of the figure), (b) the filter instantiations to which each of the component filters corresponds (circles), and (c) the filter classes upon which each of the instantiations is based (squares). Note that there is only one instantiation of the sampler, even though it is used in two different flows. On the other hand, there are two instantiations of the BPF filter class. The reason is that the filter expressions in the two flows are different.

The ability to load and interconnect high-speed packet handlers in the kernel was also explored by Wallach et al., with an eye on integrating layer processing and reducing copying [33]. Similarly, Click allows programmers to load packet processing functions consisting of a configuration of simple elements that push (pull) data to (from) each other [28]. The same model was used in
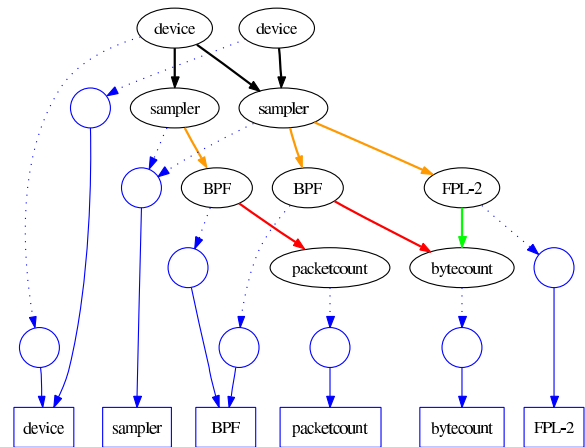

Figure 3: Auto-generated diagram of filter graph

the Corral, but with support for third parties that may add and remove elements at runtime [9]. The filter concatenation and support for a hierarchy that includes IXP1200s resembles paths in the Scout project [4]. Scout was not designed for monitoring *per se* and, hence, does not directly provide some of FFPF's features such as new languages or flow groups. Mono-lingual kernel-programming projects that also do not support these features include FLAME [1] and our own Open Kernel Environment [7]) which provide high speed processing by loading native code (compiled Cyclone) in the kernel.

2.4 Processing

A key aspect to performance is that most processing takes place at the lowest possible level, e.g. in the kernel or network processor. For example, in the FFPF implementation on IXP1200 network processors, packet processing and buffer management are handled entirely by the IXP.

As shown in Figure 4, FFPF spans all three levels of the processing hierarchy: userspace, kernel, and network interface. Filters can be loaded at any of these levels.

The figure shows that filters from lower levels (e.g. the network card) may be connected to filters at higher levels. For instance, first-pass filtering may take place at the IXP1200, followed by more expensive processing at the host. A similar approach is found for instance in paths in the Scout OS [4]. Where in the processing hierarchy filters should be loaded depends on the availability of filter classes in each space and the trade-off in efficiency vs. stability at each level. Users need not concern themselves with this task as the deployment decision is made automatically by the FFPF toolkit.

The shaded areas in the figure indicate APIs that we developed on top of the native FFPF interface. The `libpcap` implementation guarantees backward compatibility with a host of applications. As shown in Section 5, running legacy `libpcap` applications on FFPF rather than on the existing Linux framework also leads to a significant performance boost. The MAPI is a very powerful monitoring API developed within the SCAMPI project [30]. Since FFPF's functionality exceeds that of both `pcap` and `MAPI`, the implementation of these interfaces involved just a few hours work. The FFPF toolkit supports automatic allocation of filters to the most optimal place in the processing hierarchy. Moreover, when a new flow grabber is instantiated, the toolkit automatically tries to merge it with already existing 'filter graphs', so that every common prefix is executed only once.
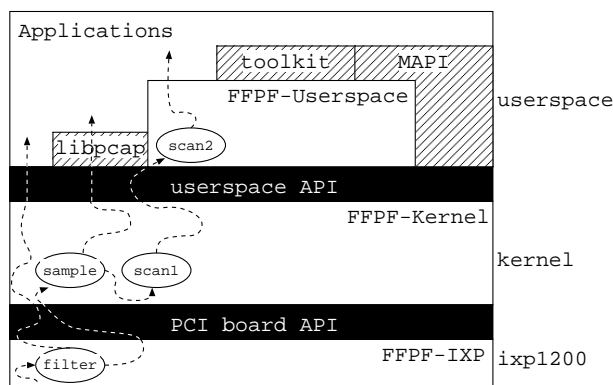

Figure 4: FFPF software structure (with some sample flows)

3 Implementation

3.1 The Buffers

Both $PBuf$ and $IBuf$ are circular buffers of N fixed size slots, with N a configurable constant. $PBuf$ slots are large enough to hold maximum-size packets, while the slots in the index buffers hold two 32 bit values: an index in $PBuf$ and the packet's 'classification result'

(the value returned by the filter). A packet is considered 'interesting' if the filter returns a non-zero result.

Applications read packets of a flow by indexing $PBuf$ of filter f with the values in $IBuf$. By default, network packets are stored in $PBuf$ from the link layer up. As applications access the index buffers, classification results are immediately available, normally within the same cache line. Although the indices *point* only to the packets in which they are interested, applications are able to see packets received by all others in the same flow group (but not those received by other groups).

3.1.1 Buffer management

Circular buffers in FFPF have two indices to indicate the current read and write positions. These are known as R and W, respectively. Whenever W catches up with R, the buffer is full. The way in which the system handles this case is defined by the buffer management system (BMS). The modular design of FFPF allows different BMSs to be used. The administrator chooses the BMS at startup time. The optimal choice depends on the mix of applications that will be executed and their relative importance. Currently, two BMSs have been defined. The first is known as 'slow reader preference' and is commonly used in existing systems. The second is known as 'fast reader preference' and is a novel way of ensuring that fast readers are not slowed down by tardy applications.

Slow reader preference. In SRP, as long as the buffer is full, all new packets are dropped. Both R and W are mapped read-only to an application's address space and updated in kernel or network card. The packet grabber in the kernel/card writes data in a group's $PBuf$ and updates W until the buffer is full, i.e., until W catches up with the *slowest reader* in the group. Thus, the slowest reader in a group may block all other readers in that group. The R value of the slowest reader will be denoted by R^*. An application explicitly updates its own R by way of system call after it has processed a number of packets and, if needed, the kernel then also updates R^*. One of the keys to speed is that R need not be incremented by one for every packet that is processed. Instead, an application may process a thousand packets and then increment R by a thousand in one go. Doing so saves many kernel boundary crossings. A similar mechanism is used for DAG cards [10].

As an example, consider the implementation on IXP1200 network processors, where packet processing and buffer management is handled entirely by the IXP. The IXP receives a packet, places it in $PBuf$, updates W, receives the next packet, and so on. Meanwhile, the filters are executed in independent processing engines on

the network processor and determine whether a reference to the packet should be placed in the filters' index buffers. Assuming that the administrator chose to use 'zero-copy' packet handling (more about the various options in Section 4.3), applications access packets immediately, as the buffers are memory mapped through to userspace. While applications process the packets, the kernel is not used at all. Only after an application has processed n packets of a flow and decides to advance its R explicitly, the kernel is activated. On the reception of a request to advance an application's R, the kernel also calculates the new value of R^* and passes it to the packet receiving code on the IXP. In the extreme case, where a single application is active, the IXP code and application work fully independently and the number of interrupts and context switches is minimal. The way FFPF's SRP captures packets in a circular buffer and memory maps them to user space is similar to Luca Deri's `PF_RING` [13], although `PF_RING` copies packets to each application individually.

Fast reader preference. FRP is a departure from the 'traditional' way of dealing with buffer overflow. In FRP mode, FFPF keeps writing packets, regardless of the status of the readers, and it is the reader's responsibility to keep up. An application that fails to keep up may have older but still unread data overwritten by new packets. In this case, R is of no concern to FFPF and used only by the application. The idea is that applications check *after* they processed a set of packets, whether or not these packets were overwritten in the meantime (and hence whether the application should consider them lost after all). For this purpose, FFPF keeps a memory mapped *wrap counter* that is incremented each time W 'wraps' to zero. Using the counter, applications can check themselves whether the current value of W is greater than their value of R and thus whether the packets they just accessed were valid.

Suppose an application is about to access a set of 100 packets when the values of R, W and wrap counter are 50, 400, and 10, respectively. When the application has finished processing, it again checks these values and now finds that W is 450, while the wrap counter is 11. In other words, the writing process has wrapped and overwritten all packets that were just processed. The application will count these packets as dropped. Note that as a result the drop rate in a group may vary from application to application. It should be mentioned that FRP is not necessarily more efficient in terms of the total processing that is required for buffer management. Rather, it distributes this computation to the applications themselves, removing the dependencies between readers that exist in a centralised solution.

3.1.2 Filter-specific memory array

The third buffer in Figure 1 is the filter's memory array $MBuf$. It is used by both the filters in the kernel and the userspace application. User applications have read and write access to the memory arrays of their filters, so the arrays can be used to exchange data between the application and a filter expression. The $MBuf$ area is *persistent*, i.e., its contents remain valid across multiple invocations of a filter. It is argued in [19] that the absence of persistent state is one of the major drawbacks of BPF. While [19] describes how BPF can be extended to also allow for persistent memory (and explicit switching between persistent and non-persistent memory is needed), this paper describes an approach in which it is part of the design from the outset.

A simple use case is a filter f which treats the entire memory array as a hash table that is used to count the number of packets received on all TCP/IP flows. The corresponding filter first checks whether a packet is TCP/IP. If so, it calculates a hash of the `<ipsrc,ipdest,srcport,dstport>` tuple and increments the counter stored at that location in the memory array. The result is that without intervention by the user application, the memory array contains the packet counts of all TCP/IP flows seen by the system (assuming the hash table is large enough). The implementation of this example is trivial if the language is capable of using persistent state. An example of such code in FPL-2 is shown in Figure 6 and will be discussed in Section 3.3.1.

3.2 The flows

Flows are captured by stringing together filters as explained in Section 2.2. The packets received in a flow can be read by the application in different ways. The simplest way is to read continuously from the buffer whenever a packet is available, e.g., using the `filter_getnext_pkt()` operation. Doing so, however, keeps the application polling constantly. From a CPU usage and context switching point of view, packets may be read more efficiently by blocking, e.g., until a certain number of packets has been received. FFPF offers two flavours of blocking: (a) `wait_for_n_pkts(n)`, a blocking call that only returns after n packets are received, and (b) installing a `filter_callback()` which is non-blocking itself and results in a callback of a registered callback function whenever n packets are received. At callback registration time, users specify how long the callback should remain active. Of course, even with `filter_getnext_pkt()` an application may block explicitly, e.g. by calling `sleep(10)` to process every 10 seconds all packets that were received in that period.

3.3 FFPF Packet Languages

While FFPF is language neutral, some languages are better suited to exploit the strengths of FFPF than others. BPF, for instance, can not by itself take advantage of FFPF's persistent state, extensibility, etc. For this reason, we developed two new languages for filter expressions, known as FPL-1 and FPL-2 (FFPF packet languages 1 and 2). They were designed to exploit all of FFPF's features. The main distinctions are that FPL-1 is a fairly slow interpreted stack language, while FPL-2 is fully optimised native code (based on registers), and that FPL-1 code can be self-modifying, while FPL-2 code is fixed. Also, the syntax in FPL-2 is much improved.

Given that FPL-1 has been around for a year now, why did we develop FPL-2? The reason is that although FPL-1 bytecode is fairly efficient, running it in an interpreter hurts performance. Moreover, as observed by McCanne and Van Jacobson: for modern processor architectures, stack-based languages are less efficient than register-based approaches [25]. For this reason, we developed a language that (1) compiles to fully optimised object code, and (2) is based on registers and memory, and (3) has a more readable syntax.

Apart from self-modification, there is little *functional* difference between FPL-1 and FPL-2. For this reason, we discuss 'FPL' as a general concept, using FPL-2 language constructs for illustration. A detailed explanation of FPL-1 and FPL-2 can be found in [8] and [12].

3.3.1 FPL

The FPL language is summarised in Figure 5. It supports all common integer types (signed and unsigned bits, nibbles, octets, words and double words) and allows expressions to get hold of any field in the packet header or payload in a friendly manner. Moreover, offsets in packets can be variable, i.e., determined by an expression. For convenience, an extensible set of macros allows use of shorthand for packet fields, e.g., instead of asking for bytes nine and ten to obtain the IP header's protocol field, a user may abbreviate to 'IP_PROTO'. We briefly explain constructs that are not intuitively clear.

FOR. The FOR loop construct is limited to loops with a pre-determined number of iterations. The break instruction, allows one to exit the loop 'early'. In this case (and also when the loop finishes), execution continues at the instruction following the ROF construct.

Registers and memory. FPL is able to access the filter's $MBuf$ by means of the assignment operator. For instance, one may assign the content of a memory location to a register, perform a set of calculations, and then assign the value of the register back

operator-type	operator
Arithmetic	`+, -, /, *, %, --, ++`
Assignment	`=, *=, /=, %=, +=, -=`
	`<<=, >>=, &=, ^=, |=`
Logical/Relational	`==, !=, >, <, >=, <=,`
	`&&, ||, !`
Bitwise	`&, |, ^, <<, >>`

statement-type	operator
if/then/else	IF (expr) THEN stmt1 ELSE stmt2 FI
for()	FOR (initialise; test; update)
	stmts; BREAK; stmts; ROF
external function	EXTERN(filter, input, output)
hash()	INT HASH(start_byte,len,tablesize)
return a value	RETURN (val)

Data type	syntax
Register $n^{\dagger}$	R[n]
Memory location n	MEM[n]
Packets access:	
- byte $f(n)$	PKT.B[$f(n)$]
- word $f(n)$	PKT.W[$f(n)$]
- bit m in byte n	PKT.B[n].U1[m]
- byte m in word n	PKT.W[n].U8[m]
etc.	(many options, including macros)

Figure 5: FPL-2 language constructs ($^{\dagger}m$ and n arbitrary variables)

to memory. All accesses to $MBuf$ are checked for bounds violations. An example of $MBuf$ usage in FPL-2 is shown in Figure 6. The code implements the filter f mentioned in Section 3.1.2 that keeps track of how many packets were received on each TCP connection (assuming for simplicity that the hash is unique for each live TCP flow).

```
// count number of packets in every flow,
// by keeping counters in hash table
// (assume hash is unique for each flow)
IF (PKT.IP_PROTO == PROTO_TCP)
THEN
    // register = hash over TCP flow fields
    R[0] = Hash(14,12,256);
    // increment the pkt counter at this position
    MEM[ R[0] ]++;
FI
```

Figure 6: Example of FPL-2 code: count TCP flow activity

External functions. An important feature of FPL is extensibility and the concept of an 'external function' is key to extensibility, flexibility and speed. External functions are an explicit mechanism to introduce extended functionality to FFPF and add to flexibility by implementing the 'filter call' semantics shown in Figure (2.b). While they look like filters, the functions may implement anything that is considered useful (e.g., checksum calculation, pat-

tern matching). They can be written in any of the supported languages, but it is anticipated that they will often be used to call optimised native code performing computationally expensive operations.

In FPL, an external function is called using the EXTERN construct, where the parameters indicate the filter to call, the offset in $MBuf$ where the filter can find its input data (if any), and the offset at which it should write its output, respectively. For instance, EXTERN(foo,x,y) will call external function foo, which will read its input from memory at offset x, and produce output, if any, at offset y. Note that FFPF does not prevent users from supplying bogus arguments. Protection comes from authorisation control discussed in Section 3.4 and from the compiler. The compiler checks the use of external functions in a filter. An external function's definition prescribes the size of the parameters, so whenever a user's filter tries to let the external function read its input from an offset that would make it stray beyond the bounds of the memory array, an error is generated. This is one of the advantages of having a 'trusted' compiler (see also Section 3.3.3. In addition, authorisation control can be used to grant users access only to a set of registered functions.

A small library of external filter functions has been implemented (including implementation of popular pattern matching algorithms, such as Aho-Corasick and Boyer-Moore). The implementation will be evaluated in Section 5. External functions in FPL can also be used to 'script together' filters from different approaches (e.g., BPF+ [5], DPF [17], PathFinder [3], etc.), much like a shell script in UNIX.

3.3.2 Monitoring application with dynamic ports

Many existing packet filters are not well suited for handling applications with dynamic ports. Such applications use control channels with well-known port numbers, while data transfer takes place over ports that are negotiated dynamically. Examples are found in peer-to-peer networks and multimedia streams that employ control protocols like RTSP, SIP and H.323 to negotiate port numbers for data transfer protocols such as RTP.

These flows are complex to monitor and the problem was considered important enough to develop a special-purpose tool (mmdump, not unlike tcpdump) to tackle it [32]. Like xPF [19], mmdump adds statefulness to the pcap/BPF architecture and in addition allows filters to be self-modifying. A filter may capture and inspect all control packets and if they contain the port number to be used for data, modify itself to also capture these packets.

While the same behaviour is supported in FFPF which allows an external function to extend the FPL-1 expression from which it was called (subject to authorization constraints), this may not be best way of handling the problem: self-modifying code is difficult to trace and debug. Moreover, there is a simpler way to monitor dynamic ports. For example, given that RTSP packets are sent on port 554, the filter in Figure 7 filters out all such packets and passes them to an external function GetDynTCPDPortFromRTSP. When called, the function scans all RTSP session packets for the occurrence of 'Transport', 'client_port' and 'server_port' to find the port numbers that will be used for data transfer (e.g., audio and video). These ports are stored in $MBuf$ (lines 4-5). If the packet is not RTSP, we check if the destination port of the packets is in the array of port numbers and if so, return the value TRUE (lines 7-9), so that the packet is sent to userspace. In other words, *only* data packets of streams that are set up using RTSP are sent to userland. Note that the example is for illustration purposes only. It is a simplified version of what real applications would use. For instance, we only deal with transfers that use TCP (also for the data) and extract just a single destination port (while the traffic is likely to be bi-directional).

```
1. // R[0] initially 0 stores no. of dynports found
2. IF (PKT.IP_PROTO==PROTO_TCP) THEN
3.   IF (PKT.TCP_DPORT==554) THEN
4.     MEM[R[0]]=EXTERN("GetDynTCPDPortFromRTSP",0,0);
5.     R[0]++;
6.   ELSE
7.     FOR (R[1]=0; R[1] < R[0]; R[1]++)
8.       IF (PKT.TCP_DPORT == M[ R[1] ] ) THEN
9.         RETURN TRUE;
10.      FI
11.    ROF
11.  FI
12. FI
12. RETURN FALSE;
```

Figure 7: Monitoring dynamic flows

3.3.3 Compile-time checks

The two FPL compilers are able to generate 'resource safe' code, i.e., it is possible to check *at compile time* how many resources can be consumed by an expression, how many loop iterations may be incurred, etc. Neither FPL language supports pointers and interaction with the rest of the kernel is limited to the explicitly registered external functions. Also, while it is not possible to determine the resource consumption of external functions *statically*, we are able to check (and control) *which* functions may be called from a filter. As a result, a simple authorisation check rejects filter expressions that do not agree with the local safety policy and no runtime checks for resource consumption are necessary. This is

explained in the next section. At runtime the code only checks for array bound violations, divide by zero, etc. By configuring the size of all buffers and slots as a power-of-2, bounds checking involves no more than a bitwise AND.

In an approach modelled after the OKE (see Section 3.5), the (trusted) FPL-2 compiler takes the filter expression, checks whether it is safe and if so, compiles it to a Linux kernel module which is subsequently compiled by `gcc`. It also generates a *compilation record*, which proves that this module was generated by the local (trusted) FPL-2 compiler. The proof contains the MD5 of the object code and is signed by the compiler (Figure 8). We check at load time whether the code was generated by a trusted compiler and whether the MD5 matches the code [8].
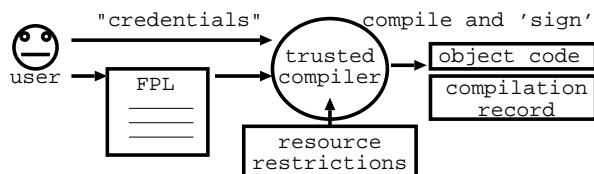
Figure 8: User compiles kernel module

3.4 Authorisation

It is important to note that the use of FFPF is not restricted to `root` users. Our view coincides with what was originally advocated in the `packetfilter` approach in Ultrix: limiting access to tools like `tcpdump` to a specific user (as found in many existing systems) is a design decision, not an axiom. Moreover, we think it is flawed. In FFPF, ordinary users may receive (in the form of credentials) explicit permission to monitor the network (possibly with restrictions, e.g., only packets with specific addresses)..

For all control operations, e.g. when flow grabbers are instantiated or filters connected (Figure 9), users present authorisation information. The information required depends on language, user id and group id. When an FPL-2 filter is instantiated, users provide both object code and compilation record. The authorisation module checks whether the code is indeed FPL-2 code generated by the local compiler. If so (and all other authorisation checks are also passed), FFPF instantiates it. All authorisation information is normally stored in a single directory indicated by an environment variable. As a result, the checks are transparent to the user.

Authorisation control is implemented as a stand-alone daemon called at instantiation time. The daemon compares flow definitions both with the users' credentials and with the host's security policy and returns a verdict ('ac-
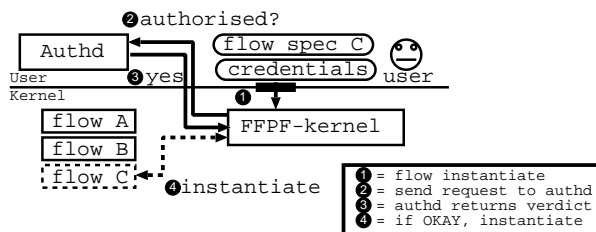
Figure 9: User loads module in the kernel

cept' or 'reject'). Credentials and trust management in FFPF are implemented in KeyNote [6]. The daemon provides fine-grained control that allows for complex policies. For instance, although we don't use most of them in the current distribution for simplicity reasons, it is possible to specify such policies as: (a) a call to external function `strsearch` is permitted for packets arriving on `eth0` *only* if it is preceded by a sampling function, (b) all calls to an external function `drop` *must* be followed by a return statement, (c) if no call is made to an external sampling function, the callback that is requested should wait for at least 1000 packets (e.g., to limit the number of callbacks), and (d) filter x may only be connected to filter y, etc. These policies can only be checked if the entire flow definition is available. The examples show that authorisation control guards against 'unsafe' flow grabbers, but can also be used to guard against 'silly' mistakes.

Authorisation control is optional. For instance, if the only party using FFPF is the system administrator, authorisation control may be left out to simplify management. A slightly modified version of the FFPF authorisation control daemon is also used in the SCAMPI network monitoring project [30].

3.5 Third-party external functions

The final two languages supported by FFPF are C and OKE-Cyclone. They are not intended to be used for packet processing by normal users on a day-to-day basis (although this is not precluded), but rather for implementing fast filters or external functions that can be called from FPL-1 or FPL-2. Writing kernel modules in C is too complex for most users, and writing code in the OKE is even more complex than that. We expect only power users to exploit the 'native code extensibility' features. Even so, once written and declared safe, the code and credentials needed to install such code can be given to third-parties.

External functions written in C and compiled as kernel modules can only be loaded by the system administrator. However, in the Open Kernel Environment [7][2] it was shown how third-party users can load fully opti-

[2]Available from `www.liacs.nl/~herbertb/projects/oke/`

mised native code in the Linux kernel, without compromising safety in any way. OKE support was added to FFPF, so that even non-root users are allowed to load fast native functions in the kernel and register them with FFPF. Subsequently, these functions can be called by or connected to filter expressions just like ordinary external functions.

The FPL-2 way of injecting code was directly modelled after the OKE, so compilation and instantiation are as sketched in Figures 8 and 9, except that the language used in the OKE is *OKE-Cyclone* , a 'crash-free' version of C [20]. Unlike FPL-2, this is a language that supports pointer memory allocation and full interaction with the kernel. Accordingly, to be able to generate 'resource safe' code, the compiler must check and instrument the user code much more strictly. Depending on the credentials provided by the user the OKE compiler restricts the user code in terms of access to resources, e.g. CPU, heap, and stack usage, access to APIs, access to sensitive fields in packets, accesses to kernel heap, etc.

Using the OKE, users no longer depend on root users to load the desired functionality as native code. The cost of full resource control in the OKE is roughly 10% compared to plain C. Like authorisation control, the OKE is optional. We refer to [7] for a discussion of related work in safe kernel programming.

3.6 FFPF packet sources

Packets enter the FFPF framework via a call to an FFPF function called `hook_handle_packet()` which takes a packet as argument. As this is the only interface between the code responsible for packet capture and the FFPF packet handling module, it is easy to add new packet sources. Currently, three sources are implemented.

The first source, known as `netfilter`, captures packets from a netfilter hook. Netfilter is an efficient abstraction for packet processing in Linux kernels (from version 2.4 onward). The second source, known as `raw`, also works with older kernels. The third packet source, known as `ixp`, differs from the other two in that the IXP1200 device is assumed to be dedicated to monitoring in the FFPF framework[3]. As this packet source is a substantial project in and of itself, we will summarise its main characteristics in a separate section.

---

[3]If the IXP is used as a 'normal' NIC (e.g., as described in [23]), FFPF's standard packet sources work without modification.

4 The IXP1200 packet source

4.1 The IXP1200 processor

The Intel IXP1200 runs at a clockrate of 232 MHz and is mounted on a Radisys ENP2506 board together with 8 MB of SRAM and 256 MB of SDRAM. The board contains two Gigabit network ports ①. Packet reception and packet transmission over these ports is handled by the code on the IXP1200 processor ②. The Radisys board is connected to a Linux PC via a PCI bus ③. The IXP itself consists of a StrongARM host processor running embedded Linux and six independent RISC processors, known as microengines. Each microengine has its own instruction store and register sets. On each of the microengines, registers are partitioned between 4 hardware contexts or 'threads' that have their own program counters and allow for zero-cycle context switches.
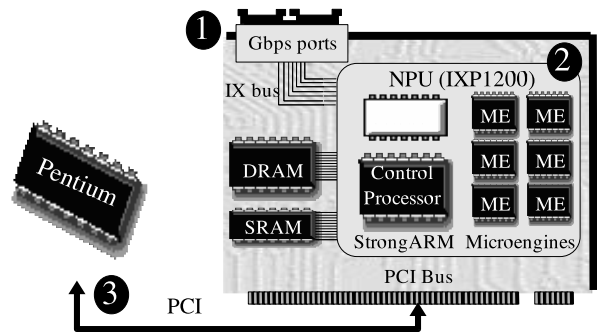

Figure 10: The IXP1200 NPU

4.2 FFPF on the IXP1200

The $PBuf$ in the IXP implementation resides in SDRAM on the network card. FFPF maps the packet buffer as well as the other FFPF buffers into the host address space to support zero-copy functionality. A separate control structure, consisting of packet descriptors on a hardware-supported LIFO is kept in SRAM.

4.2.1 The microengines

A single microengine per Gigabit port is responsible for receiving and buffering packets in $PBuf$. All remaining microengines execute application-specific filter expressions. For this purpose, we implemented an FPL-2 compiler that generates Intel's microengine-C. On each of the microengines a skeleton *main loop* with a slot for user code is provided by the FFPF framework. Users with the right credentials may 'plug in' FPL-2 expressions in this slot. When such a flow grabber is instantiated, the complete program is loaded on the microengine.

An FFPF IXP1200 filter is bound to a microengine's filter. As a consequence, the IXP1200 can support a maximum of five filters. As the IXP1200 is considered 'obsolete' and no longer supported by Intel, and newer versions of the IXP support more microengines at higher clockrates, both the number of filters that can be supported and their speeds may be expected to increase.

A filter uses all four threads to process the packets one by one. If the filter determines that the packet is interesting, the microengine places an index for the packet in the filter's $IBuf$. Otherwise, a special function pkt_drop() marks the packet as finished by setting a flag in the SRAM packet descriptor entry. In addition, it checks if all other filters are also finished with the packet. If so, the packet descriptor will be reclaimed.

4.2.2 StrongARM core components

The StrongARM components are responsible for control tasks, including initialization, loading and control of microengines, memory mapping of SDRAM to the host, initialization of different memory buffers, etc. It is also responsible for signalling across the PCI bus. For instance, in a 'slow-readers preference' implementation, the StrongARM receives the 'advance read pointer' messages from the host.

4.3 To copy or not to copy

While many research projects aim for zero-copy implementations, we argue that this is not always optimal. For this reason, we developed three implementations: (1) zero copy, (2) *always* copy packets to the host processor, and (3) only copy packets that have been marked as interesting by a host-side flow. Which version will be used by FFPF can be decided by the administrator at runtime.

The problem with zero-copy is that packet accesses from the host inevitably become slow, and if the average number of accesses per packet exceeds a certain threshold, the performance decreases. For instance, the zero-copy implementation just described works well, if most of the packet accesses are performed by the microengines and few or none by the host applications. Once the host application also needs to access the packet extensively, most reads have to cross the PCI bus. While some benefit may be expected from prefetching (reducing the overhead to less than a round-trip time), the penalty is still severe. If on the other hand, we had chosen the inverse zero-copy solution, whereby packets were immediately written to host memory and not stored in on-board SDRAM (ignoring potential bus bandwidth problems), host accesses would be optimised at the expense of the code on the microengines. We conclude that in situations where both the host and the IXP have an average number of accesses per packet that is substantial compared to a single copy across the bus, copying the interesting packets once is always better than a zero-copy solution.

5 Experimental analysis

The FFPF architecture is arguably more complex than many of its competitors. A possible consequence of increasing expressiveness may be a decrease in performance of simple tasks. To verify FFPF's applicability in the general case we have directly compared it with the widely used Linux socket filter (LSF), by running identical queries through (a) libpcap with Linux' LSF backend, and (b) libpcap based on an FFPF implementation. We realise that for various aspects of filtering faster solutions may exist, but since the number of different approaches is huge and none would be 'obvious' candidates for comparison, we limit ourselves to the most well-known competitor and compare under equivalent configurations (using the same BPF interpreter, buffer settings, etc.).

To show their relative efficiency we compare the two packet filters' CPU utilization (system load) using OProfile[4]. Since packet filtering takes place at various stages in kernel and userspace, a global measure such as system load can convey overall processing costs better than individual cyclecounters. Results of subprocesses are discussed using clockcycle counts later on. Both platforms have been tested with the same front-end, tcpdump (3.8.3). Use of the BPF interpreter was minimized as much as possible: only a return statement was executed. All tests were conducted on a 1.2 GHz Intel P3 workstation with a 64/66 PCI bus running Linux 2.6.2 with the new network API (NAPI), using FFPF with FRP and circular buffers of 1000 slots.

5.1 Packet sniffing performance

Figure 11 shows the overhead incurred by running the packet filters at increasing bitrates for 1500 byte packets (600Mbps is the maximum rate we are able to generate reliably from a single source). While not shown in the figure, we verified that packet size plays no role in this experiment, only the packet rate. In general, we can see that FFPF makes more efficient use of the system than LSF, as the amount of FFPF idle time for high rates may exceed that of LSF by a factor of two, depending on the capture length. Unlike LSF, FFPF performance, while always better than LSF for high rates, depends strongly on the maximum packet capture length. Varying the number of slots in the packet buffer has a similar effect on performance, which leads us to conclude that it is probably caused by memory access and cache behaviour. As LSF lacks a circular buffer, cache misses will be rare.
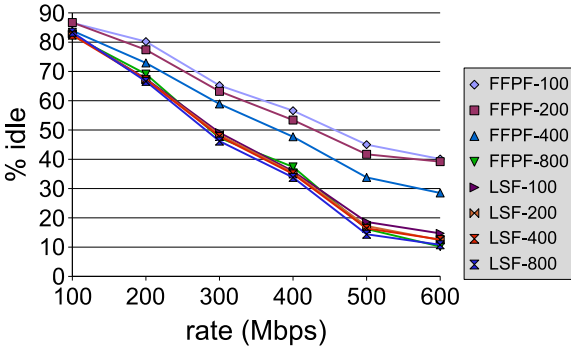
---

[4]http://oprofile.sourceforge.net

Figure 11: System idle time for FFPF and LSF as a function of the bandwidth for different capture lengths

Larger caches, or tweaking of buffer size helps to alleviate the dependency. However, this was not done in these experiments. The drop rate for FFPF in all of these configurations is negligible, while for LSF at 600 Mbps the drop rate is 2-3%, depending on the capture size.

The use of shared buffers in FFPF reduces copying and context switching, especially if the number of applications increases. It is our hypothesis that network monitoring will be increasingly important and that multiple different applications will want to filter overlapping traffic (e.g., for intrusion detection, traffic engineering, a sysadmin interested in an overview of activity of protocols, etc.).

Figure 12 shows, for high bitrate, how the two frameworks scale when starting an increasing number of tcpdump applications with overlapping flows. Since LSF duplicates much of the work for each application, it quickly saturates. We should point out that for reasons unknown to us, OProfile never reports 0% idle time (the minimum is always 2-3%). Even with just two simultaneous applications LSF reaches maximum system load and consequently starts dropping packets. With 6 client applications LSF drops between 64% (LSF-100) and 75% (LSF-800) of all incoming packets. FFPF, on the other hand, drops 10% (FFPF-100) to 15% (FFPF-800).

Interestingly, as the CPU load never reaches 100% for FFPF, the drop cannot be attributed to starvation. Rather it is caused by buffer overflow: by keeping the number of $PBuf$ slots constant throughout the experiments (1000 slots), there were not enough slots to support six parallel client applications. Increasing the number of slots will decrease the droprate due to buffer overflow, but will increase overall system load due to cache and line misses. However, even without tweaking FFPF clearly outperforms LSF. Its more *gradual* performance degradation can be expected as little work is duplicated. Filtering is handled in the kernel and duplicate tasks are merged.

The remaining performance penalty is therefore related only to userspace data output and the remaining context switches.
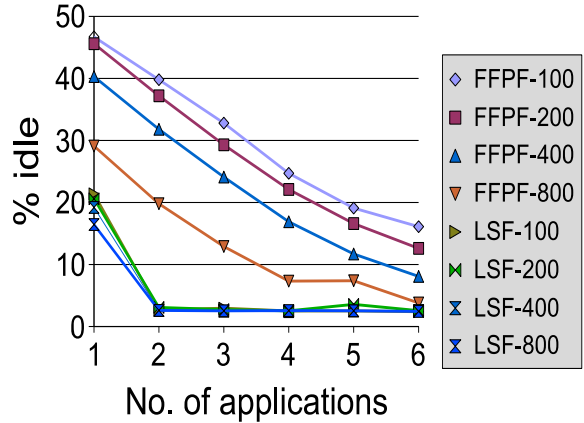

Figure 12: Idle time as function of the no. of concurrent applications for various capture lengths at 600Mbps

5.2 Analysis of operational costs

We have shown that FFPF increases packet filtering efficiency even for relatively simple tasks. The previous tests fail to show, however, where the performance gains originate and how the system would operate with more complex filters. Table 1 breaks down the overhead in several subtasks.

Rows $1 - 4$ deal with general overhead, namely the calling of a filter , the total overhead per filter in the flowgraph (with filters that return immediately after being called to show only framework overhead), the saving of an element in an index buffer and the saving of a 1500B packet to $PBuf$. The decrease in cost by a factor 50 for saving a reference in $IBuf$ over saving a full packet shows that in the presence of overlapping flows, FFPF's flowgroups can truly increase efficiency. This, combined with memory mapping of buffers, is perhaps the most important factor to the gradual degradation of performance when running multiple applications.

Rows $5 - 8$ show resource consumption for a number of often executed filters, namely the Aho-Corasick pattern matching algorithm used in snort [31], and a simple tcpdump filter[5] executed in FPL2 code and BPF respectively. Rows 7 and 8 show that FPL2 is four times as efficient as BPF, even for such a trivial filter. While not shown, cost savings grow with expression complexity (as expected). Unfortunately, the performance of really elaborate filters, such as those shown in Figures 6

---
[5]"ip src 192.168.1.3 and ip proto \udp and dst port 54321"

	task	cycles
1	calling a filter	71
2	single filter stage in flowgrabber	171
3	saving index in $IBuf$	154
4	storing packet in $PBuf$	7479
5	waking up user process	624
6	snort's Aho-Corasick algorithm (match)	1000
7	same but without match	9900
8	FPL-2 filter	185
9	BPF filter	740

Table 1: Breakdown of various types of overhead in cycles

and 7, cannot be compared, as such complex filters cannot be expressed in BPF.

Pattern matching can also be seen to be costly. We show the case where an application (e.g., snort) is only interested in packets that contain a signature. Especially when a signature is *not* found after scanning the entire packet processing costs are high (the result shown is for 1500 byte packets). By executing this function in the kernel, FFPF eliminates a journey to userspace for *every* packet, avoiding unnecessary packet copies, context switches and signalling. Note that even compared to the high overhead of pattern matching, the overhead of storing packets is significant.

The complete cost of context switching is hard to measure due largely to the asynchronous nature of userspace/kernel communication. One measure that is quantifiable is the cost to wake up a user process, row 5 in Table 1. At 600 cycles (4 times the overhead of a filter stage), this is a significant cost. To minimize this overhead users can reduce communication by batching packets. Waking up a client process only once every N packets reduces this type of overhead by $N-1$. In FFPF, N is configured by the size of the circular buffers and can be thousands of packets.

Furthermore, comparing filtering (row $5 - 8$) and framework (rows $1 - 4$) overhead shows that costs due to FFPF's complexity contributes only a moderate amount to overall processing. Finally, we discuss in a related publication that the IXP implementation is able to sustain full Gigabit rates for the same simple filter that was used for Figure 1, while a few hundred Mbps can still be sustained for complex filters that check every byte in the packet [29]. As the FPL-2 code on the IXP is used as pre-filtering stage, we are able to support line rates without being hampered by bottlenecks such as the PCI bus and host memory latency, which is not true for most existing approaches. We conclude that FFPF can be used as an efficient solution for both simple (e.g. BPF) and more complex (sampling, pattern matching) tasks.

6 Related work

Much of the related work was discussed in the text. In this section, we discuss projects that, although related, could not easily be linked with any *specific* aspect of FFPF.

MPF enhances the BPF virtual machine with new instructions for demultiplexing to multiple applications and merges filters that have the same prefix [34]. This approach is generalised by PathFinder which represents different filters as predicates of which common prefixes are removed [3]. PathFinder is interesting in that it is amenable to implementation in hardware. DPF extends the PathFinder model by introducing dynamic code generation [17]. BPF+ [5] shows how an intermediate static single assignment representation of BPF can be optimised, and how just-in-time-compilation can be used to produce efficient native filtering code. All of these approaches target filter optimisation especially in the presence of many filters, and as a result are not supported directly in FFPF (although it is simple to add them as external functions). With FPL-2, FFPF relies on gcc's optimisation techniques and on external functions for expensive operations.

Like FPL-2, and DPF, the Windmill protocol filters also target high-performance by compiling filters in native code [24]. And like MPF, Windmill explicitly supports multiple applications with overlapping filters. However, compared to FPL-2, Windmill filters are fairly simple conjunctions of header field predicates. MPF extends the BPF instruction set to exploit the fact that most filters concern the same protocol, so that common filter tests can be collapsed. It seems that the suppport is at the level of assembly instructions which makes it fairly hard to use. Moreover, for each of these approaches packets are still *copied* to individual processes and require a context switch to perform processing other than filtering. As FFPF is extensible and language neutral, each of these approaches can be added to FFPF if needed.

Operating systems like Exokernel, and Nemesis [16, 22] allow users to add code to the operating system and implement single address spaces to minimise copying. While FFPF no doubt can be efficiently implemented on these systems, one of its strengths is that it minimises copying on a very popular OS that does not have a single address space.

Support for high-speed traffic capture is provided by OCxMon [2]. Like the work conducted at Sprint [18], OCxMon supports DAG cards to cater to multi-gigabit speeds [10]. Unlike FFPF, both approaches have made an *a priori* decision not to capture the entire packet at high speeds.

Nprobe is aimed at monitoring multiple protocols [27] and is therefore, like Windmill, geared towards applying protocol stacks. Also, Nprobe focuses on disk bandwidth

limitations and for this reason captures as few bytes of the packets as possible. FFPF has no *a priori* notion of protocol stacks and supports full packet processing.

Gigascope is a stream database for network analysis that resembles FFPF in that it supports an SQL-like stream query language that is compiled and distributed over a processing hierarchy which may include the NIC itself [11]. The focus is on data management and there is no support for backward compatibility, peristent storage or handling dynamic ports.

Most related to FFPF is the SCAMPI architecture which also pushes processing to the lowest levels [30]. SCAMPI borrows heavily from the way packets are handled by DAG cards [10]. It assumes the hardware can write packets immediately in the applications' address spaces and implements access to the packet buffers through a userspace daemon. Common NICs are supported by standard `pcap` whereby packets are first pushed to userspace. Moreover, SCAMPI does not support user-provided external functions, supports a single BMS and relies on traditional filtering languages (BPF). Finally, SCAMPI allows only a non-branching (linear) list of functions to be applied to a stream.

7 Conclusions and future work

In this paper, we discussed the architecture and implementation of the fairly fast packet filter. FFPF provides a complete monitoring platform that caters to most applications. It was shown to be both more flexible and more efficient than existing approaches. Speed is gained by minimising both packet copying and context switching, pushing processing to the lowest levels, and executing computationally expensive functions as native code. It was demonstrated that FFPF outperforms Linux socket filters even for traditional applications that make no use of FFPF's more advanced features. The concepts of flows and flow groups, the concatenation of expressions, the buffering mechanism that favours fast flows and the minimisation of copying are generic mechanisms that may serve as the basis for fast packet processing in any OS.

In a future version of FFPF, we will explore the notion of flow groups further. Specifically, readers that are 'too slow' will be automatically placed in a separate flow group, lest they hinder fast applications. Also, we will shortly release a version of the architecture for use in application domains other than monitoring. For instance, in addition to packet reception, this version will be able to block, edit and (re)transmit packets, allowing for uses such as firewalling, network address translation and routing. Most of the required functionality is implemented, but currently not enabled in FFPF. Finally, we are in the process of developing a distributed version of FFPF.

References

[1] Kostas G. Anagnostakis, S. Ioannidis, S. Miltchev, and Michael B. Greenwald. Open packet monitoring on flame: Safety, performance and applications. In *Proc. of IWAN'02*, Zuerich, Switzerland, December 2002.

[2] J. Apisdorf, k claffy, K. Thompson, and R. Wilder. Oc3mon: Flexible, affordable, high performance statistics collection. In *1996 USENIX LISA X Conference*, pages 97–112,, Chicago, IL, September 1996.

[3] Mary L. Bailey, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkar. Pathfinder: A pattern-based packet classifier. In *Operating Systems Design and Implementation*, pages 115–123, 1994.

[4] A. Bavier, T. Voigt, Wawrzoniak M, L. Peterson, and P. Gunningberg. Silk: Scout paths in the linux kernel, tr 2002-009. Technical report, Dept. of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.

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

[6] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system version 2. *Network Working Group, RFC 2704*, September 1999.

[7] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proc. of OPENARCH'02*, New York, USA, June 2002.

[8] Herbert Bos and Georgios Portokalidis. Packet monitoring at high speed with FFPF. Technical Report TR-2004-01, LIACS, Leiden University, Leiden Netherlands, January 2004.

[9] Herbert Bos and Bart Samwel. The OKE Corral: Code organisation and reconfiguration at runtime using active linking. In *Proceedings of IWAN'2002*, Zuerich, Sw., December 2002.

[10] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, Hamilton, New Zealand, April 2000.

[11] Chuck Cranor, Theodore Johnson, Oliver Spatschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM*

SIGMOD international conference on on Management of data, pages 647–651. ACM Press, 2003.

[12] Mihai Cristea and Herbert Bos. A compiler for packet filters. In *Proc. of ASCI*, Netherlands, June 2004.

[13] Luca Deri. Improving passive packet capture:beyond device polling. `http://luca.ntop.org/`, 2004.

[14] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): a network subsystem architecture for s erver systems. In *Proceedings of the second USENIX symposium on Operating systems de sign and implementation*, pages 261–275, 1996.

[15] Aled Edwards, Greg Watson, John Lumley, David Banks, Costas Calamvokis, and C. Dalton. User-space protocols deliver high performance to applications on a low-cost gb/s lan. *SIGCOMM Comput. Commun. Rev.*, 24(4):14–23, 1994.

[16] D. R. Engler, M.F. Kaashoek, and J.W. O'Toole Jr. The exokernel approach to extensibility (panel statement). In *Proc. of OSDI'94*, page 198, Monterey, Ca., November 1994.

[17] Dawson R. Engler and M. Frans Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM'96*, pages 53–59, 1996.

[18] G. Iannaccone, C. Diot, I. Graham, and N. McKeown. Monitoring very high speed links. In *ACM SIGCOMM Internet Measurement Workshop 2001*, September 2001.

[19] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: packet filtering for low-cost network monitoring. In *Proc. of HPSR'02*, pages 121–126, May 2002.

[20] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX 2002 Annual Technical Conference*, June 2002.

[21] Ken Keys, David Moore, Ryan Koga, Edouard Lagache, Michael Tesch, and k claffy. The architecture of Coral-Reef: an Internet traffic monit oring software suite. In *PAM2001*. CAIDA, RIPE NCC, April 2001.

[22] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *JSAC*, 14(7), September 1996.

[23] Kenneth Mackenzie, Weidong Shi, Austen McDonald, and Ivan Ganev. An intel IXP1200-based network interface. `http://www.ixaedu.com/resources/`, 2003.

[24] G. Robert Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Computer Communication Review, ACM SIGCOMM, volume 28, number 4*, Vancouver, Canada, October 1998.

[25] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. 1993 Winter USENIX conference*, San Diego, Ca., January 1993.

[26] J.C. Mogul, R.F. Rashid, and M.J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. of SOSP'87*, pages 39–51, Austin, Tx., November 1987. ACM.

[27] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt. Architecture of a network monitor. in proc. of PAM'03, 2003.

[28] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[29] Trung Nguyen, Willem de Bruijn, Mihai Cristea, and Herbert Bos. Scalable network monitors for high-speed links: a bottom-up approach. In *IPOM'04*, Beijing, China, 2004.

[30] M. Polychronakis, E. Markatos, K. Anagnostakis, and A. Oslebo. Design of an application programming interface for ip network monitoring. In *Proc. of NOMS'02*, Seoul, Korea, April 2004.

[31] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Adminstration Conference*, 1999.

[32] J. van der Merwe, R. Caceres, Y. Chu, and C. Sreenan. Mmdump - a tool for monitoring internet multimedia traffic. *ACM Computer Communication Review*, 30(4), October 2000.

[33] D. Wallach, D.Engler, and M.F. Kaashoek. ASHs: Application-specific handlers for high-performance messaging. In *Proc. of SIGCOMM'96*, pages 40–52. ACM Press, 1996.

[34] M. Yuhara, B.N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large m essages. In *USENIX Winter*, pages 153–165, 1994.