Lessons learned in developing a flexible packet processor for high-speed links

Tomáš Hrubý, Willem de Bruijn and Herbert Bos
Department of Mathematics and Computer Science
Vrije Universiteit Amsterdam, The Netherlands
{thruby,wdb,herbertb}@few.vu.nl

Mihai Lucian Cristea and Li Xu
Leiden Institute of Advanced Computer Science
Universiteit Leiden, The Netherlands
{cristea,lxu}@liacs.nl

***Abstract*— There is a growing need for packet processing at high link rates. Commodity hardware and software are not able to cope with multi-gigabit speeds. Existing solutions for handling high rates tend to be ad-hoc and cannot easily be used for new applications. In this paper, we desibe our experiences in implementing a flexible packet processing framework that is capable of dealing with high link rates. We do so by listing and discussing the design principles that were used in the development of the fairly fast packet filter. The paper indicates which aspects we still consider important and which assumptions proved to be incorrect.**

INTRODUCTION

The need for packet processing at (multi-)gigabit speeds is growing, e.g., for monitoring, accounting, firewalls, intrusion detection systems (IDSs) and network address translators (NATs). The importance of such applications is on the rise and we expect that multiple applications will be present at the same node, for instance a monitoring platform at the edge of the network.

At the same time developers of such systems are plagued by many difficulties. Some argue that the increase in link speed exceeds Moore's law. Others counter this is only true in labs, while in practice there is little deployment beyond OC-192 or OC-768 (i.e., 10-40 Gbps). While the jury is out on how fast network speed is growing compared to computing speed, we do observe that commodity PCs are struggling even to keep up with rates of, say, a gigabit per second. This means that there is an urgent need for a platform that (a) can handle current link rates, and (b) scales to future link rates. There are many problems with current technology, with respect to both hardware and software. We briefly discuss some prevalent ones. We argue that existing off-the-shelf solutions are lacking as neither the hardware nor the software is geared for high speeds. For instance, well-known hardware problems concern the speed of memory and buses, while software problems concern excessive copying and context switching by the operating system (OS).

At least as serious is that existing solutions suffer from tunnel vision, i.e, each may solve one particular problem well, but they cannot be easily combined. For instance, one may use BPF kernel filters [1] because they are so widely supported, Windmill protocol filters [2] to take advantage of the compilation to native code, or nprobe [3] because of the efficient way in which traces are stored on disk, but it is difficult to see how one could *combine* these filters in a mix-and-match fashion and make it operate as a single unit. Similarly, hardware vendors such as Endace have developed efficient software for making optimal use of their advanced DAG network card [4]. Such software can and has been used to develop high-speed monitoring tools (e.g., OC3Mon [5]), but is not designed for use in heterogeneous environments with different types of cards, such as a PC equipped with both a DAG card, and a network processor board such as the Intel IXP [6] (and perhaps even a few plain old NICs with no intelligence whatsoever).

None of the existing systems tries to resolve the various problems in packet processing within a single, practical framework. We will now show how we addressed them, where there is room for improvement, and where our assumptions proved wrong. In this paper we explain the design principles that underly our system and that we believe to be relevant for high-speed packet processing in general. Some are well-known, others we discovered when developing our system. We do not intend to repeat the lessons in (earlier) similar papers such as [7]. Instead, we discuss additional design principles that we consider useful. Specifically, we show how to engineer for high throughput, how to exploit hardware heterogeneity and how to accommodate software diversity. At the same time, it was our goal to build a generic solution that can be used for any application. Moreover, we wanted to design a system that is *usable* both for advanced users and novices.

Unfortunately, engineering for speed conflicts with designing a generic solution, as it is almost always faster to provide a tailor-made solution for one specific problem. For instance, the authors of the virtual router project carefully crafted the code to match a specific hardware architecture (the IXP1200) and application (routing), resulting in very good performance, but limited flexibility [8]. Similarly, [9] focused solely on getting packet from fast links to disk. The Click project, on the other hand, provides a much more generic and user-friendly environment, but with significant overhead in the framework itself [10]. In this work, we are aiming for both flexibility and speed. We want to support fast links, multiple concurrent applications, and still allow users to develop complex applications in a trivial manner. Unlike Click, we do not limit ourselves to the kernel. Instead, we want to use resources closer to the wire also (e.g., network cards, routers). The identified methods form a framework for efficient, flexible

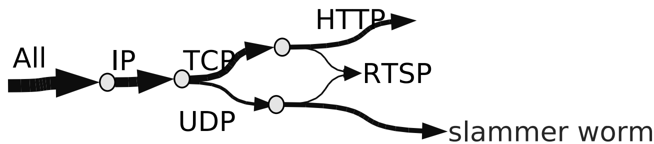

Fig. 1. Arbitrary packet selection with generalised 'flows'

packet processing that supports a diverse set of concurrently active applications. We have learned the lessons explained here while developing FFPF [11], a high-speed system for network monitoring. The first version of FFPF was released in May 2003, and it has seen several major overhauls since, to accommodate the lessons we learned. As many problems we encountered are not specific to monitoring it is our belief that anyone implementing a packet processor will face the same problems and may therefore benefit from the solutions we found. While we use FFPF to demonstrate the practicality of our approach, we focus solely on its support for generic packet processing. Details specific to monitoring have been discussed in depth before [11] and are therefore omitted. All code is available from `http://ffpf.sf.net`.

We will discuss the principles we found one by one. With each we start by giving a description of the problem at hand. The guideline is then explained by drawing from our experience in developing FFPF and by looking at alternative frameworks. Where applicable, we support our case with experimental results. We finish by summarising the points and looking at possible improvements (Section VIII).

I. Generalise flows

The task of a packet filter is to help applications select what they perceive as interesting data with minimal effort. It must simplify packet processing by abstracting away the implementational difficulties surrounding heterogeneity of hardware and software. A complete packet processing framework must be able to supply applications with all the data they need, whether the nature of these data be packets, statistics, or other. In other words, limiting the information that applications may receive is not a good solution. Frameworks that focus only on sending packets, such as the widely-used PCAP library, fail in this regard.

We propose the concept of a generalised 'flow' as the core abstraction for the packet processor. Unlike most existing uses of the term (e.g., TCP flow, Cisco NetFlow), a flow is defined as *any* subset of network packets that is of interest to the user. For instance, a user may request a single flow that comprises all TCP port 80 SYN packets together with all UDP packets containing the Slammer worm. Each flow streams through a number of processing functions (such as filters, counters and queues), that together will be called the *flowgraph*. An example is shown in Figure 1. If need be, nodes in the flowgraph may modify packets within a flow (e.g., for NATs or routers).

For additional (non-streaming) information about the traffic, such as statistics that are kept by the functions, orthogonal abstractions can be used. Any abstraction will do, as long as it can handle generic data structures. We propose as basic abstraction for such data an unstructured sequence of bytes. On top of this basic abstraction it is possible to implement more complex structures.

Note that the concept of a flow is fairly common in networking literature. However, it is mostly applied in a more restricted sense (e.g., TCP or IPFIX flows). Frameworks like x-kernel, Linux *netfilter/iptables* or Windmill permit packets to stream over chains of processing functions, but have limited support for obtaining additional (non-streaming) information [10], [12]. Other approaches like *click* do support this, but are ill-suited for streaming the flows to multiple userspace applications.

In FFPF, each function in the flowgraph has three buffers: a circular packet buffer ($PBuf$) where the flow's packets may be stored, a circular index buffer ($IBuf$) in which each valid entry contains a pointer to a packet in $PBuf$ as well as a 32-bit classification result, and (optionally) an unstructured sequence of bytes that serves as persistent state ($MBuf$). It may be used, for instance, to keep statistics, NAT tables, etc.

II. Support heterogeneous processing hierarchies

For convenience, most approaches build on abstraction layers that are too high (e.g., at the level of Linux `netfilter`). By doing so, they make it impossible to exploit features that are offered by advanced network cards. For example, DAG cards allow filtering on the NIC, the same is true for Juniper [4], [13] and open source OS-based routers. These features cannot easily be exploited by approaches that build on high-level abstractions.

Our second design principle addresses hardware problems in two ways: it aims to avoid sending all packets across the PCI bus to memory, and to support all programmable/configurable hardware of which more and more is found on the market. For this purpose, we need to process packets at any level that allows us to do so, e.g., userspace, kernel, on stream processors and programmable network cards, and even on remote devices such as Juniper routers. Note that such a description suggests a hierarchical tree-like hardware model with NICs and routers at the lowest levels and userspace application at the top[1].

This is illustrated in Figure 2 for a PC containing several NICs, a programmable DAG card, and an Intel IXP2400 network board. The dashed box at the left means that it may be connected to a programmable Juniper router. All boxes that are not shaded are part of our actual testbed. The PC is a slow Linux PIII at 1.1GHz with a 64/66 PCI bus equipped with several NICs and an IXP2400 evaluation board from Radisys (ENP-2611). The IXP2400 contains on chip a single XScale control processor running Linux and eight multi-threaded RISC processors, known as micro-engines (MEs) running no OS whatsoever. Not drawn is an IXP-based network processor (IXP2850) that sits in a separate box (much like the Juniper

[1] We treat kernel and userspace as different hardware levels as the hardware enforces different programming environments for the two.

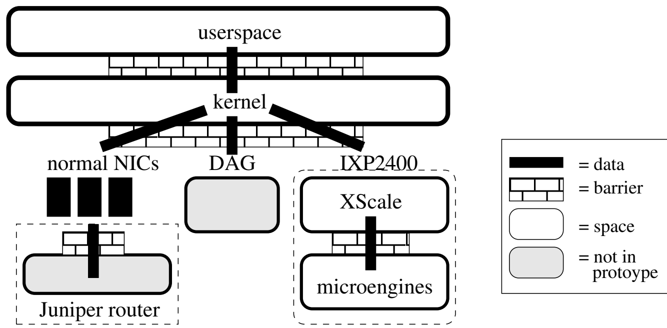

Fig. 2. Processing hierarchy with multiple 'flowspaces' and 'boundaries'

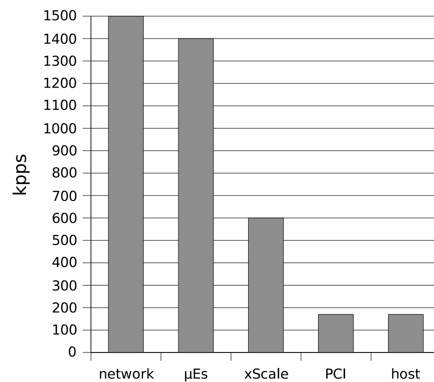

Fig. 3. Bottlenecks in the processing tree

router) and connects to one of the PCs gigabit NICs via a network link. The reason why we do not show it is that while we do have rudimentary support for the device, the implementation is far from mature. We assume the hierarchy is strict and heterogeneous.

The implication is that instead of doing everything in userland, we spread processing across many levels in a tree. As a result, the mapping of functions on the hierarchy becomes complex.

Mapping a flowgraph on top of a hierarchy of heterogeneous hardware and software is not a trivial exercise; it is dependent on function class availability at nodes in the hardware hierarchy, as well as on the ability to connect data-streams between nodes. To aid placement we model each node explicitly as a self-contained execution 'bubble', or *flowspace*. A specific implementation of a function class is bound to an individual flowspace. For instance, a Linux kernel BPF class may be tied to the host kernel flowspace. That is not to say that the same interface (e.g., BPF) cannot be implemented in multiple locations.

Obtaining a successful mapping involves finding a path through the flowspaces such that all functions can be instantiated and interconnected; this is a general resource allocation problem. To shield the user from micro-management tasks, the runtime system should take care of this complex task. The complexity is reduced by three simple rules discussed next.

A. Process at the lowest possible level

The assumption is that if a function is provided lower in the hierarchy, this is more efficient than the same function higher up. While it is possible to find counter examples, in our experience this rule works well in practice. Figure 3 shows the throughput of the processing levels in our test environment. We were able to process close to 1.5M packets per second on the MEs of the IXP. As we move higher up the tree, throughput quickly diminishes. This is fine as long as we make sure most processing is done at the lowest levels, and fewer and fewer packets travel up to the higher levels. For each function to be executed we therefore find the lowest point in the hierarchy that supports this function and instantiate it here, provided the following rule of thumb is not violated.

B. Ensure packets travel up (and down) at most once

A packet that has been sent across the PCI bus is never sent down again until an explicit 'transmit' occurs and then it never travels up again. Combined with the previous rule we suggest that the processing tree should work like a funnel, processing fewer and fewer packets as we move higher up the tree. This behaviour is in line with the assumption that lower-level hardware is more adept at processing packets at high-speed, but less flexible in the actions it supports. It is also the approach that was used in the Virtual Routers project [8].

In our implementation, each flowspace exports a standard interface with methods for querying function classes, controlling functions and accessing the data-streams. The runtime system can use this interface to search for implementations of a requested function class. Placement starts at the lowest levels of the processing tree (in line with rule II-A) and queries bubble up until they can be accommodated. In this way the algorithm iteratively walks through a requested flowgraph. Each time a function is instantiated the system connects the data-streams between the new function and those on which it is dependent, which have necessarily already been instantiated. Because incoming streams can only flow upwards (rule II-B), the instantiation search space becomes more limited as we process more of the request. The end-result will resemble a Directed Acyclical Graph that can span across all levels of the hardware tree, as shown in Figure 4.

C. Instantiate a function at the lowest common successor to its inputs

Rules II-A and II-B are not sufficient if packets may be received from multiple sources. This happens, for instance, if a user specifies a flow that contains packets arriving on either `/dev/ixp0` or `/dev/ixp1` that pass through BPF filter foo. It may be the case that the BPF function class is available both on the XScale of the two IXP boards, and in the host kernel. The question is: where should it be placed? Rules II-A and II-B simply state that it should be instantiated at the lowest level, for instance, the first IXP that is probed. However, this is incorrect, as the BPF filter should be applied to packets from both IXPs. We could instantiate foo on both

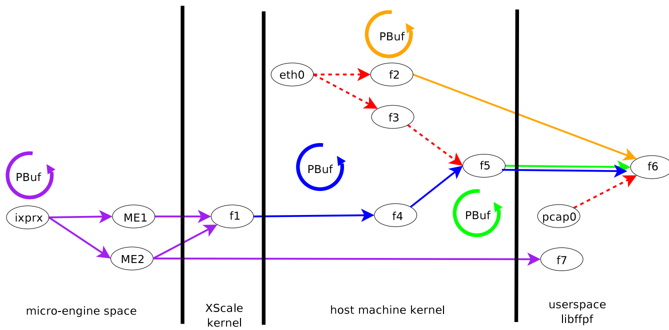

Fig. 4. A complex flowgraph on a heterogeneous processing tree

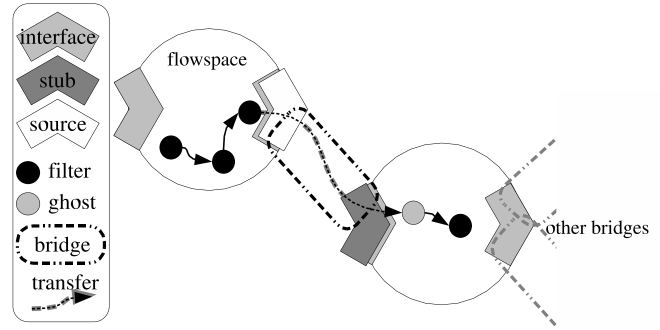

Fig. 5. Bridging barriers

IXPs, but this is awkward also, because we now create two separate streams rather than a single one. Imagine the BPF filter was a packet counter: in that case we would have two results rather than a single aggregate packet count. Instead, we use the following simple rule: if a flowgraph contains a function that receives packets from multiple nodes in lower-level flowspaces, instantiate it at the level in the processing hierarchy that is the lowest common successor to each of the nodes and that contains the function class.

FFPF implements this by routing the function instantiation along explicit paths. Each function of the flowgraph that is instantiated in the processing hierarchy has an identifier known as `flowid`. Part of the `flowid` is the *path* field, a 32 bit number that is given a value when we descend down the hierarchy to instantiate the function at the lowest flowspace that supports this function. The field specifies the path from the root (userspace on the host) to where the function is instantiated in the hierarchy. The 32 bit number is logically divided in 8 subfields of 4 bits, one subfield for each flowspace level in the hierarchy. The processing hierarchy is a tree so the subfields can be used to indicate which branch to take at each flowspace level to get to the flowspace in which the function is instantiated. A special value is reserved for 'invalid' which indicates that the function may not be instantiated at this level.

FFPF instantiates the flowgraph in a bottom-up fashion. When a function foo must be instantiated, it descends down the processing hierarchy until either of the following occurs (i) it reaches the lowest level, or (ii) all flowspaces beneath it are marked invalid in foo's path field. When it has reached this flowspace, it bubbles up again until it reaches a flowspace that supports the function class for foo. At each level in foo's path field we mark the subfields of the flowspaces below it as invalid. After the function is instantiated the path field is simply returned to FFPF, which uses it to initialise the path fields of all functions that depend on foo by copying the subfields that are marked invalid to all the path fields of dependent functions. This way the function will always be instantiated in the lowest flowspace that is successor to all its inputs. As each flowspace deals only with the flowspace immediately beneath it in the hierarchy, the mechanism is completely decentralised, which makes it is very simple to add new flowspaces and hierarchy levels.

III. Engineer for Barrier Diversity

As mentioned above, the processing hierarchy is generic and tree-like and may include routers, network cards and OS-enforced protection domains. In FFPF terminology, each node in this tree is programmable through an abstract interface: the 'flowspace' (userspace, kernelspace, etc.). Flowspaces are separated by barriers, such as network links, PCI buses, and kernel-userspace boundaries. Having a single model for crossing barriers makes the system more extensible. The model we have chosen is that of stubs that are used by a higher-level flowspace to interact with the layers beneath it. The stubs presents a 'ghost' image in a higher-level flowspace. This image transparently forwards requests to the real flowspace interface in the lower level. While the model is always the same, the implementation of the forwarding methods should vary depending on the nature of the barrier.

A. Recursively extend the control and datapaths

By exposing stub interfaces to their parent, flowspaces can transparently build bridges across barriers, such as the userspace/kernelspace divide or the PCI bus. Bridges are barrier-specific, therefore a flowspace can export multiple stubs, e.g., one for PCI and one for PCI Express, as shown in Figure 5. The stub interface's methods implement the low-level code necessary to move data across the bus and communicate with the real flowspace interface on the other side, freeing the framework (and the user) from these operational details. The FFPF library interface is unaware of the number of flowspaces in the system, as at each flowspace only the direct descendants are visible through their exported stubs. Requests, such as instantiation queries, are recursively reflected to the lowest level in the hierarchy. This mechanism removes the need for a central authority, and thereby reduces algorithmic complexity.

Unfortunately, boundaries differ in how they are best crossed. When packets are physically copied across high-latency boundaries, often the most efficient way to do so is to 'push' the packets to the remote side, as it avoids the round trip time (RTT) latency incurred by 'pull' models. However, some boundaries do not permit pushing. Moreover, sometimes pushing is less efficient than pulling, for instance when a packet is hardly ever touched at the host.

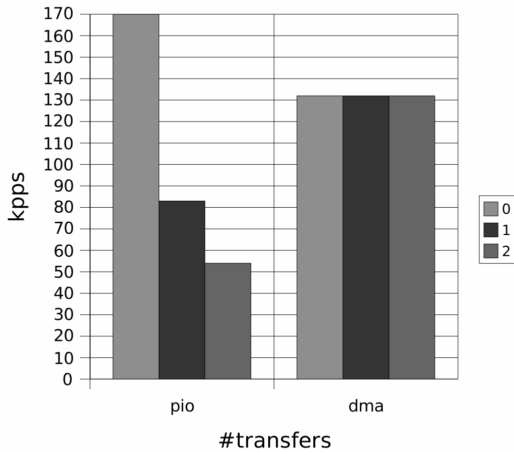

Fig. 6. Performance comparison of PCI transfer strategies

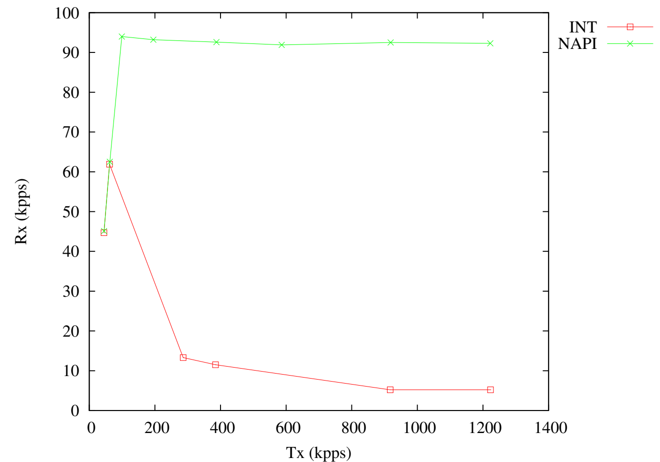

Fig. 7. Influence of polling on high-speed packet handling

Different applications may require different solutions. We have implemented three transfer strategies [14], each one tailored to a specific use-case. For clarity's sake we discuss these strategies for what proved to be the most important and complex barrier in our test environments: the PCI bus.

Zero-copy is a well-known performance enhancement to high-speed processing that transparently maps data from the network card up to userspace. It benefits those applications that only need to access parts of packets and only infrequently, as it only moves data that is actually requested. However, when packets are accessed more than once, or when the payload is inspected, it is often more efficient to copy the entire packet at once: *copy-once*. This method minimises request transfers over the bus and can optimise DMA to limit the copy-induced overhead. Finally, *copy-on-demand* takes the middle road, using zero-copy until a packet is explicitly marked as interesting by a function, after which it is copied at once. We had expected this method to give the best performance, as the majority of packets uses zero-copy for matching, while only accepted packets that are sent to applications have to be copied to local memory.

Figure 6 proves this assumption wrong. Plotted is the throughput in one particular traffic monitoring experiment we conducted with FFPF in our testbed for both programmed IO and DMA for 0, 1 and 2 requests per packet. It shows that throughput quickly diminishes when using PIO, but remains constant with DMA. What is not shown is that host CPU usage is also only at 50% for DMA, as opposed to 100% for PIO. In addition, sending requests at bus-width increments and using aligned addresses can increase performance to PCI's peak rate. Queueing requests to the DMA processor will especially increase sustained rate; this device can reorder requests to further maximise throughput, while at the same time offloading the CPU. Because memory remapping fails to make use of these performance enhancements it performs poorly. While [7] explains that DMA is not always the best choice and depends on the system, our experience is that in practice true zero-copy has very few real benefits for all but a very small minority of applications.

Details of zero-copy and other such transfer strategies are abstracted away, even made interchangeable, by moving the implementational code into the framework. The stub/source abstraction that we use for exporting flowspaces is also applied to functions in the flowgraph, as shown in Figure 5. When a barrier lies between two (or more) functions the one on the lower-level is mirrored higher up in the hierarchy. Whereas normal functions use a function call to forward packets up the tree, this 'ghost' image must use interrupts or polling to scan for packets. The datapath from the real function to its image traverses the bridge built between the flowspaces. Control-path requests are routed through the stub interface to setup a static forwarding route that is unaware of physical intricacies. Depending on access patterns, packets may be exported to the ghost through a zero-copy method by using memory remapping (e.g., through the mmap system call) or copied directly into its flowspace-local packet buffer. Either way, it must be waken up and notified when new data is available through a barrier-specific method.

B. *Interrupt at low rates, poll at high rates*

One of the lessons we learnt from experience corresponds to the observations in [15]: at high speeds the overhead of per packet interrupts is unacceptable and polling is more effective. On the other hand, for slow links, polling wastes cycles and interrupts are preferable. The number of interrupts at high speeds can be effectively reduced by polling after each interrupt. This is what is applied in the well-known NAPI patch to the Linux kernel.

Figure 7 compares throughput between two network cards. One has a NAPI-enabled driver, while the other does not. As the example shows, interrupt-only processing breaks down when the CPU becomes saturated. Throughput is not even sustained, it diminishes further as the rate increases. This behaviour stems from overhead incurred in the hardware-initiated interrupt handling routine. As the number of received

interrupts increases, more and more time is spent in this function, thereby shrinking the cycle budget for actual processing. By disabling interrupts during processing and switching to polling instead, the NAPI patch circumvents interrupt-induced starvation.

Switching between interrupting and polling depending on throughput is a heuristic that is equally valid in crossing other barriers than the PCI bus. Kernelspace/userspace crossings incur significant overhead because of context-switching. Waking up a process for every packet is similar to interrupt handling, but sleep-based polling will increase latency when the load is low. Busy-polling is not advisable on the host, as it eats cycles best used elsewhere. Therefore, mechanisms that switch between interrupting and polling automatically based on the load will perform best. That said, busy polling remains a viable option on special purpose stream hardware, such as the MEs found on IXP boards. In that case no other processes have to contend for cycles.In FFPF all accesses except those for the MEs are developed NAPI-style.

C. Engineer for multi-endianness

Most network-enabled hardware devices use big-endian encoding. Unfortunately, the popular x86 architecture has chosen little-endian for its native encoding. This conversion, together with high context-switching costs makes the x86 architecturally a suboptimal choice for high-speed packet processing. For various other reasons, cost not being the least, it remains the architecture of choice. These problems are less apparent in the core of the network, where hardware can be tailor-made and the forwarding path is relatively simple. It is on end-hosts, firewalls and border gateways that complex packet processing takes place. Here commodity hardware - and thus the x86 - prevails. Combining both layouts in the same abstract framework is therefore needed, but this introduces additional conversion issues.

The first heuristic to follow is that, because network packets are encoded as big-endian themselves, we want to minimise conversion to little endian. Especially in forwarding it is preferable not to send data up to the host, as that would incur a double conversion: another example of why processing in the lower echelons of the tree is advantageous.

When data can be shared between functions running on fast hardware and other on the (x86-based) local host we must safeguard correct handling from within the framework, as individual functions have no knowledge of their dependencies. Especially when packets are destined for the local host and we use zero-copy or copy-on-demand this problem will be apparent. Correct handling can be ensured by having buffers signal their endianness. Secure access methods can then be written that are able to convert data transparently. The drawback of this scheme is that additional control-flow is needed in the access functions, increasing memory access overhead. Unfortunately, this is currently the only viable option. What we can do to alleviate the stress on the system is choose on which side of the barrier to incur the majority of this overhead by toggling endianness of the buffers. Also, faster, non-schizophrenic buffers can be used in datapaths where no collisions can occur. Our buffer API masks the details of the underlying access methods from the calling functions and applications. Directly accessing the raw data through pointers is not allowed; buffer layout is intentionally unspecified. Not only does this abstraction solve endianness woes, it also helps us in other areas, e.g., to switch between multiple reader/writer collision handling methods.

IV. AVOID NEEDLESS COPYING AND CONTEXT SWITCHING

The keyword is 'needless'. As we have seen, 'zero-copy' can be less efficient than 'copy once'. If a packet is stored in a NIC buffer and accessed frequently by code on the host CPU, it is cheaper to copy the whole packet to main memory once, than to read and write across the PCI bus (zero-copy) multiple times. But in other scenarios it may be cheaper to leave packets on the card or never store them on the card in the first place.

The best known software problem is the excessive amount of copying and context switching in common OSs. From the NIC packets are copied to temporary kernel buffers and then copied again to their destinations in userspace. If a packet is needed by more than one application, it is copied multiple times. This is an increasingly important problem, as (concurrent) use of firewalls, NAT and intrusion detection grows, especially on commodity end-hosts.

In addition, while handling the packets, we are forced to context switch between kernel and applications constantly. The question is: why is this needed? The answer is that on the one hand packets *have* to be processed by the kernel, while on the other the kernel processing capabilities are too *limited*. For example, the snort IDS processes rules such as: generate an alert if the payload of a TCP port 80 packet contains the string xyz. On most systems, all the kernel can do is filter out TCP packets to port 80 and copy them to userspace. the rest is done in userland and requires a copy and a context switch. Neither are needed if we can do *everything* in the kernel. Alternatively, the card could copy the packets straight to userland so that the kernel never even sees the packets [4]. Doing so would be less efficient because it increases copy-induced overhead with each additional application, but it would at least remove a fair number of context-switches. It would not remove the context switches between applications though. Unfortunately, neither of the mentioned solution is possible in most current systems.

A. Minimise both 'horizontal' and 'vertical' copies

The problem is more complex than deciding whether a packet should be copied to higher levels in the hierarchy (vertical copies). We also want to avoid copying the same packet to multiple applications (horizontal copies).

One method of solving both problems is by sharing packet buffers between applications and lower level processing elements. Packets received from regular NICs are already shared throughout the Linux operating system kernel, and we engineered FFPF to reuse these handles as far as possible. For practical reasons we cannot use these buffers outside of the

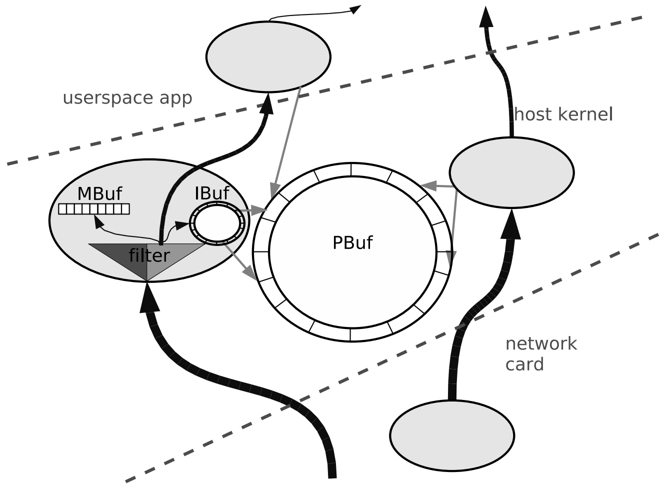

Fig. 8. Sharing packets within a flowgroup

kernel, and therefore had to create our own special purpose buffers.

In FFPF, we have introduced the notion of a *flowgroup*, a logical group of applications not unlike UNIX groups. While we define flowgroups in a more precise way later in this section, the main idea is that functions in flowgraphs may share the same $PBuf$, regardless of whether they reside in kernel or userspace. This is even true if the functions belong to different applications. The only restriction is that they need to be in the same flowgroup. See also Figure 8.

While the $PBuf$ is shared, each function has its own $IBuf$. Recall that each valid entry in $IBuf$ contains a pointer to a packet in $PBuf$ as well as a 32-bit classification result. The classification result can be used to pass additional information about the packet to interested applications. For instance, an IDS function may reserve a unique classification value for each attack it knows about. A shared packet buffer can dramatically reduce the copying-induced overhead referred to earlier, because we incur no more than one copy will take place per group regardless of groupsize. As with flowgraph overlapping discussed in Section VI, this advantage is most apparent when multiple applications are running concurrently.

On a regular host, sharing of packets using the $PBuf/IBuf$ scheme works well. What we did not anticipate, is that the additional structure imposes an additional burden when communicating over constrained resources. Especially PCI throughput suffers. The advantages obtained by using DMA for packet transfer may be partly lost, because we still update the indices through programmed IO. The lesson is that by designing for optimal flexibility, we are hurting performance. Current work is aimed at removing the index buffers from the lowest levels and implementing the bridge across the PCI using a simple packet queue that is well-suited for DMA. The cost will be somewhat reduced flexibility.

Address space remapping allows us to map kernelspace memory into the address space of each interested userspace process. This is used, for instance, to map $PBuf$, $IBuf$, and $MBuf$ to userspace. Similarly, we can remap kernelspace virtual memory to encompass IO device memory. Address space remapping is not generally applicable to distributed resources (e.g., routers), as we cannot improve much on the single copy that is currently incurred. Zero-copy over IP is seldom interesting, as the latency built up with multiple traversals over the network will quickly lead to unsustainable delays.

When sharing data between multiple applications, we need to address potential security problems. While most packet processing applications are currently run by the system administrator, there are plenty of uses for processing by regular end-users. Demultiplexing regular host-bound traffic and software-based firewalling are prime examples. For privacy reasons we cannot allow users to inspect arbitrary packets of their peers.

To solve this problem we group together requests in the flowgroups mentioned earlier. In principle, each member in a flowgroup has access to the same packets. Whether they are interested in all the packets they have access to is a different issue. Note that this mirrors fairly closely the UNIX access control groups. Indeed, we support grouping based on applications' group id (GID), but also implement UID, PID and one-for-all policies. In case multiple applications in different flowgroups run concurrently, we incur a performance penalty as this reduces potential overlap in processing paths and again duplicates copies.

It is desirable to have as few $PBuf$s per flowspace or even per flowgroup as possible. However, it is not always possible to avoid having more than one, as is illustrated in Figure 9. We see a flowgraph consisting of six functions $F1-F6$. If $F1$ and $F2$ are executed on IXPs and the copy policy is copy-on-demand, both will store packets in a different (local) $PBuf$. We indicate the $PBuf$ that is used by a function by listing the $PBuf$'s identifier in the table below it. So $F1$ uses $PBuf$ 2 and and $F2$ uses 4. A function $F3$ that is dependent on $F1$ and $F2$ will need to access packets from both $PBuf$s. A fourth function $F4$ perhaps works on packets that are received via FFPF's netfilter hook from an ordinary (non-programmable) NIC. It passes through $F4$ and $F5$ without being saved in any $PBuf$. Finally, both $F3$ and $F5$ feed into $F6$ which classifies the packets as interesting for the userspace application. This means that the packet must be saved. The existing $PBuf$s cannot be used for this purpose, so we have no choice other than to instantiate another $PBuf$ here. $F6$ now refers to three $PBuf$s. Note that all buffer allocation occurs at load time. At runtime there should be no more dynamic memory allocation on the datapath (see also Section VII-A)..

B. Minimize Cache Misses

Performance gains by sharing buffers is also offset by another issue. Contrary to our original expectations, introducing shared buffers *may* actual hurt performance in the single application scenario. Because we have to deal with asynchronous flowspaces a shared buffer must hold a relatively large backlog of packets. While we reduce context switch penalties, we may increase cache misses. When only dealing

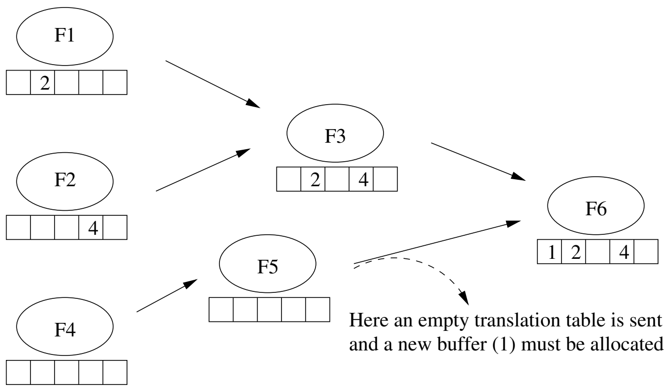

Fig. 9. Multiple packet buffers in a flowgroup

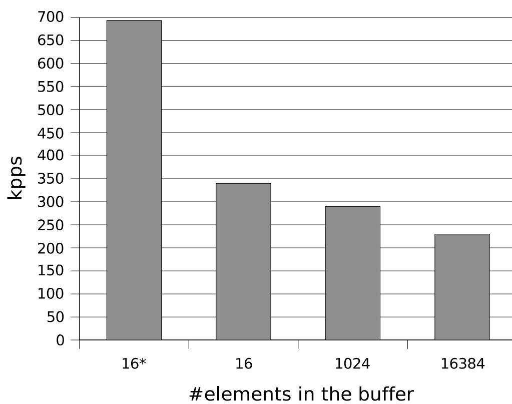

Fig. 10. Influence of cache misses on throughput

with a single application, or with a group of per-packet synchronised applications, the entire packet can be kept in very fast cache, reducing impact of memory reads. Running tests under multiple packet buffer sizes indeed showed that single process performance decreases immediately if we cross a hardware-specific threshold. During our tests the threshold lay between 10 and 100 slots (of 2000 bytes each) per buffer, but that value will fluctuate with the various level 1, 2, and 3 cache sizes available. As the impact of cache-misses was never taken into account, our reference architecture (FFPF) does not try to optimise cache access at the moment. Figure 10 shows the impact of buffer size on performance with a 16KB L1 data cache. As the buffers increase in size, throughput drops. Even the smallest buffer (`16`) is considerably slower when no buffer is inspected at all (`16*`). Choosing small buffer sizes will help, but this solution is difficult to exploit, as it quickly leads to buffer overflow and thus to dropped packets. Another common memory-reduction technique is to limit packet capture length to much smaller values than the maximum packet size. However, this clashes with the goal of supplying applications with all the information they need; again an example of the conflict between flexibility and speed.

C. Allow full processing and bypassing throughout the hierarchy

To minimise context switching, we need more processing in the kernel and lower layers. As mentioned earlier, what should be avoided is that each packet *must* be processed by the kernel when the processing that can be done is always insufficient.

FFPF allows users to spread 'full' processing across all layers. At the same time, it also allows flowspaces to be bypassed. For instance, if we do not provide any function classes in the kernel, the flows will stream directly from the card to userspace without touching the kernel at all.

V. Maintain language and platform neutrality

We already mentioned that it is hard to combine different packet processing platforms and languages. Take, for example, the de-facto standard in monitoring: PCAP with BPF. The API is popular, but the implementation is known to have serious drawbacks. For instance, it tends to incur sub-optimal performance due to the interpreted nature of the code. Also, it lacks of scope, because there is no support for backward jumps or persistent state. Consequently, useful features (e.g., pattern recognition) have been written on top of the framework in userspace, instead of in it.

As there is no one-size-fits-all solution to packet processing, the ability to mix and match is desirable. This means we need to support approaches that are stateless (e.g., [1], [16]–[20]) as well as those that are stateful (e.g., [21]–[23]). Stateful processing is attractive, as many applications are only interested in statistics (e.g., "what is the percentage of p2p traffic in my network?") rather than the packets themselves. If functions are able to generate the statistics themselves using persistent state, there is no need to send the packets to the applications at all.

The lack of persistent state is indicative of a more general shortcoming of many frameworks: by looking only at per packet processing they disregard other aspects of flows. For example, applications may be interested in flow statistics or may want to receive incidental callbacks. Not dealing with these issues within the framework limits its usefulness, as the PCAP example shows. The most integrated solution in this regard can be found in the Monitoring API (MAPI) defined in the EU SCAMPI project [23]. Whereas FFPF only exports simple memory arrays for static state-keeping, the MAPI bundles filter elements with functions that read out their datastructures. The packet counter, for instance, is bundled with a function returning an integer instead of a pointer to an unspecified buffer.

A. Allow functions to be combined

We have mentioned that mixing and matching of different solutions is desirable. In order to do so, we should be able to connect functions written in one language to functions written in a different language.

There are different ways in which functions can be combined. First, functions may be connected much like a UNIX pipe, where the output of one stage is the input of the next. In

FFPF, this was generalised to DAGs known as flowgraphs. We refer to this way of combining functions as streaming semantics.

In pipes, there is no shared memory between stages. Shared memory can be useful when one function depends on the output of another for its internal control flow. For this purpose, we have designed several new languages (collectively referred to as the FFPF Packet Languages or FPLs), that are able to exploit all advanced features of the FFPF framework. For instance, they implement combining two functions using function call semantics, i.e., it is possible to 'call' any function from the FPL code, even if this code is written in a different language. For instance we may call a BPF filter, but also hardware assisted functions, such as crypto units on the IXP2850 network processor. Under function call semantics, the callee is executed with the memory context ($PBuf, IBuf$, and $MBuf$) of the caller. In our opinion, both the streaming and the function call model have their place in a generic framework.

VI. Support complex processing graphs

In most current systems, packet processing functions such as filters are defined either as a single expression (e.g., a BPF expression), or at best as a list of sequentially applied functions (e.g., the 'UNIX pipe-like' way of expression network flows in the MAPI). It is unclear why this is, as we tend to think of protocols in terms of trees (e.g., "IP consists of TCP, UDP and other packets; TCP contains HTTP, SMTP and ssh, and UDP carries RTP and NFS traffic"). Even when we do not think about protocols directly, in packet selection it is common to think in terms of sets (e.g., "of all HTTP traffic I am interested in the `GET` request). Forcing users to collapse, say, a tree into a list or even a single expression is bad for modularity/reuse and for mapping on hardware. For example, a function to search for a pattern in a sequence of bytes is easier to reuse than a collapsed tree that, say, first filters out the UDP and TCP packets, and then applies pattern matching to TCP port 25 packets and another function to UDP packets. Moreover, it may be possible to map simple filters on the hardware, but not pattern matching. This is harder if filtering and pattern matching have collapsed in a single function. A final problem is that it is unclear how functions written in different languages can be collapsed.

Unlike Principle II, which deals with hardware configuration, this principle deals with the way users express their processing requirements. We have argued that tree collapse is harmful for modularity and mapping on hardware. Instead, users should express their processing in the most natural way. Expressing processing needs in a tree-like graph is not sufficient. For instance, a user may be interested in counting the total number of RTSP/TCP and RTSP/UDP packets. In this case, a single counter needs to be placed after the UDP and TCP branches have split. This example is shown in Figure 1

A different approach is taken by popular packet processors used in routing and forwarding. OpenBSD's `pf` and Linux's `Netfilter/IPTables` solutions are representatives of

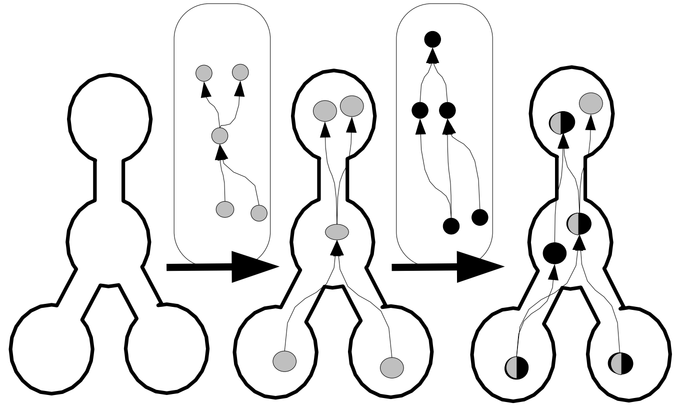

Fig. 11. Overlaying multiple requests onto a single processing graph

rule-based filtering frameworks. In such systems packets are sent to FIFO queues. Elements in the queues are matched consecutively against an ordered list of possibly complex rules until a match is found. Then, an action is performed that is linked to the rule. Regular actions are `drop`, `forward`, `accept` or `move` into a different queue. Specialized queues can be defined for packet mangling (e.g., for network address translation), connection tracking and other extensions. In this view, rules are similar to functions in our graph-based representation. Indeed, by sharing queues, complex processing graphs can be created in Netfilter. However, in practice few queues are defined. Firewalls and subnet routers have relatively simple targets: accept or drop. The rules to match against, on the other hand, can be extremely complex. It is also not uncommon for rulesets to grow into the thousands, each of which can lead to a potential cache-miss *for each packet*, severely limiting performance. Advanced rules can depend on connection state of a flow, on application protocol-specific options or even on traffic behaviour ('traffic shaping'). Moving all logic into individual rules potentially duplicates processing, similar to how multiple BPF filters operate. Also, worst-case classification time scales linearly with the ruleset. A separate issue is that naive implementations can lead to inefficient use of memory buffers. Such issues have been identified in Netfilter. The optimised `nf-Hipac` version manages to outperform the original, in part by reducing the memory-lookup cost per rule. Its method is at least partly based on rearranging the sequential ruleset into a decision-tree. This solution is similar to FFPF, but can outperform it, as it circumvents costly function calls. The trade-off made is that rule expressiveness is much more limited. The two approaches can therefore be seen as complementary. A Hipac filter could easily be accommodated within the FFPF framework, for instance.

A. *Express processing requests as DAGs*

More efficient use of table-based frameworks can be made by reducing overlap in classifying code. Multi-level processing can be seen as both an extension of table-based as of pipe-based solutions. Superseding both in flexibility, while poten-

tially costing no more than tables, is a Directed Acyclical Graph (DAG)-based representation. This treats rules as atomic selection steps. The difference with traditional table-based solutions is that steps can be part of deeply nested structures. The DAG representation maps nicely on top of the 'stream' semantics introduced before.

In such a framework users specify their requests as connections of individual processing steps. A BPF filter can select packets based on their header, while a pattern-matching element searches for worms, and a connection tracker helps with traffic shaping. At runtime the DAGs that encode user requests are overlayed on one another to form an active processing structure: the *flowgraph*, as shown in Figure 11. Each element in the flowgraph is a function that can drop packets from its input stream. Functions are not part of the FFPF framework; instead, they can be any function that adheres to a minimal API. The BPF virtual machine that is shipped with the Linux kernel has been wrapped in a thin FFPF envelope, for example. We call such functions *function classes*, which can be instantiated into *functions* at runtime by supplying them with a parameter (for BPF this could be "src localhost"). Through function overlaying the concept of a flowgraph can cut computational costs.

B. Overlay user requests to minimise work duplication

A selection request is incorporated into the flowgraph in four steps. First, the user creates a request DAG, details of which are handled in the next section. This request is parsed into a uniform representation and checked for syntactic errors in the userspace library. Second, when found correct, this request is sent down the processing hierarchy. Each node in the network is matched against the library of function classes in the flowspace, whereby we move upwards in the processing tree whenever a node cannot be matched with a function class.

At the end of this round each node in the request should be matched to a function class in such a way that the rule of continuous upward flow is adhered to. If no such route exist the entire request is cancelled. Otherwise, we say that the request is *populated*: it has found a successful match with the available resources. In the third phase the processing tree is again walked, but now datastructures are created: the function classes are instantiated into live functions and the interconnections between functions are setup, including bridges over barriers and memory buffers. At this point we also allocate the memory for the $PBufs$. The rule we use to decide where the memory should be allocated (the current flowspace, or a higher-level flowspace) is that the memory is allocated in a higher-level flowspace if there are no functions at the current flowspace that depend on this function. In that case, we automatically push the packets to the next level.

We do not setup structures earlier, because a populate request has a high chance of failure, and setting up and tearing down these structures is costly. When the structures have been set up the flowgraph is ready for processing, but put into a sleep state. An activation call starts the main processing loops.

When the flowgraph is shared between applications, instantiation requests try to map function class requests onto already running functions. In FFPF we reuse functions only when they are identical.

Some functions (e.g., filters) distinguish between human-readable forms of their expressions and faster, interpreted versions. BPF and FPL3 are examples. To facilitate these functions an additional translator interface exists within the framework. By supplying a BPF-human to BPF-bytecode translator, human readable BPF input will be translated to virtual machine instructions transparently, similar to how it is handled by PCAP. More interesting is the FPL3 example. As FPL3 code can be compiled to native code, the location of instantiation decides whether to create a kernelspace module, userspace library or ME object file. When a request is populated the translator is automatically invoked. In its turn it forwards requests to an FPL3 compiler optimised for the given flowspace. Finally, if non-host resources are used (e.g., the IXP board), an external management application is notified which takes care of loading the code into the environment. For the IXP the manager is a separate application running on the network card. This tool keeps track of available MEs, handles function class mapping onto them and controls the on-board memory layout.

VII. Separate control and data planes

Instantiating a complex processing hierarchy on distributed hardware is complex and may require several visits to each level in the hierarchy. Since this is in the control plane such overhead is acceptable, as long as the runtime overhead is minimised. Moving complex actions to a separate control-plane is a well-established rule in high-speed stream processing, and needs no further explanation. A practical rule that can be derived from this principle in our case is the following.

A. Provide all memory allocation in the control plane

Nowhere in the FFPF datapath do we use dynamic memory allocation in the datapath. Memory structures are either static, allocated at initialisation (which is control-path) or taken from pre-allocated shared pools.

Additionally, pointer lookups must be minimised. Unfortunately it proved impossible to have pre-calculated pointers in all structures, as some structures change often. The buffer lookup table within a flowspace is an example of such a necessary evil.

VIII. Support different levels of abstraction

This final principle is well-known in the study of human computer interaction, but often lacking in systems software. To improve the usability of a system, the design should cater to different levels of expertise and different ways of working.

A successful example in the field of packet processing is pcap/BPF [1]. Using tools like `tcpdump` users are able to express simple filters using a high-level expression language, application writers can use `libpcap` as a convenient library, while experts may even write BPF filters directly. We propose

to follow a similar structure, adding just one level: a mode of operation where users simply combine predefined functions.

Novice users may employ FFPF's graphical user interface (GUI) to select and connect a set of packet processing functions from a library of function classes to form a DAG. The library contains function classes from different packet processing approaches, e.g., BPF filters, pattern matching algorithms and statistics gatherers. The DAG determines the application's flow. If applicable, functions in the request may be parameterised from within the GUI. For instance, users may specify the BPF filter to use in the flowgraph by typing the pcap expression in a pop-up window associated with the BPF function. In other words, a user first selects a function class and provides all the information that is needed to instantiate the function. Similarly, users may specify that the results of a function (either the packets themselves or the persistent state) should be visible in the user application, by ticking an 'export' box in the GUI. When the flowgraph is ready, users select the application they want to connect to its output.

The GUI transforms a DAG in its graphical form to an expression in the FFPF input language. The input language is a simple specification language for attributed DAGs that allows one to connect in an arbitrary fashion a set of functions. It contains two language constructs ('>' operates much like UNIX redirection, and '|' is used for branching) and a way of grouping elements (by means of square brackets '[]'). Rather than through the GUI, the language can also be used directly from the command line. As a trivial example, the following expression sends all data from devices `eth0` and `eth1` to a BPF filter, where all TCP port 80 traffic that passes the filter is sent both to function `pktcount` and to function `bytecount`.

```
[[(dev,expr=eth0)|(dev,expr=eth1)] >
   (BPF,expr="tcp and dest 80")] > [(pktcount)|(bytecount)]
```

The example shows that functions may be written in different languages. As explained in [11], we currently support a growing set of languages, including BPF, C, and a few homegrown packet languages. One of the languages is known as the FFPF packet language 3 [24] (FPL3) and was designed to exploit all the advanced features of FFPF. It is efficient, supports persistent state and compiles to optimised native code. Current compiler targets include Linux userspace, Linux kernel and Intel IXP1200 and IXP2400 network processors. Moreover, it is possible to 'call' any other FFPF function from within the expression, implementing function calls semantics.

Like `pcap`, the FFPF API can be used as a library for building new applications. Although the libraries support different levels of expertise, the basic usage is very simple: programmers *create* a flow handle, *populate* their flow by providing an expression in the input language, *instantiate* the flowgraph, and then *activate* it. FFPF is responsible for mapping the various functions on the processing hierarchy. Additionally, an extensive *Tcl/Tk* library for FFPF is provided for developing graphical applications.

On top of this native API, we have implemented several FFPF 'personalities'. First, we have implemented the well-known `libpcap` so that we are able to run the bulk of the legacy applications. Second, we have implemented the much more advanced MAPI, which provides fast and stateful packet processing on behalf of monitoring applications, and allows multiple functions to be executed in sequence on an FFPF-like flow. However, the processing is limited to a linear list, which provides less flexibility than a DAG. More importantly, the MAPI is at a lower level of abstraction than the FFPF API, as developers have to hard-code their requests as multiple API calls.

Expert users may extend the function classes that are available in FFPF. For instance, we have added a class that allows users to employ in FFPF any function in the MAPI library without modification. As the function API in FFPF is so simple, it is not difficult to add other approaches as well (e.g., different filter approaches, or a specific firewall).

OTHER RELATED WORK

Others have tried to extend existing solutions. MPF, for example, enhances the BPF virtual machine with new instructions for demultiplexing to multiple applications and merges filters that have the same prefix [19]. This approach is generalised by PathFinder which represents different filters as predicates of which common prefixes are removed [17]. PathFinder is interesting in that it is amenable to implementation in hardware. DPF extends the PathFinder model by introducing dynamic code generation [18]. BPF+ [16] 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. These approaches target filter optimisation especially in the presence of many filters.

That low-level processing is practically within reach has been shown before. FPL3 relies on `gcc`'s optimisation techniques and on external hard-coded functions for expensive operations. Like FPL3, and DPF, the Windmill protocol filters also target high-performance by compiling filters in native code [20]. And like MPF, Windmill explicitly supports multiple applications with overlapping filters. However, compared to FPL3, 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 support 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.

Support for high-speed traffic capture is provided by Oc3mon [5]. Like the work conducted at Sprint [9], Oc3mon supports DAG cards to cater to multi-gigabit speeds [4]. Unacceptable for many situations, both approaches have made the *a priori* decision not to capture the entire packet at high speeds. Nprobe [3] is a protocol monitor that made a similar choice.

TABLE I

SUMMARY OF THE DESIGN PRINCIPLES

1) Generalise flows
2) Support heterogeneous processing hierarchies
3) Engineer for barrier diversity
4) Avoid needless copying and context switching
5) Maintain language and platform neutrality
6) Support complex processing graphs
7) Separate control and data planes
8) Support different levels of abstraction

TABLE II

FEATURE COMPARISON OF SELECTED FRAMEWORKS

principles	1	2	3	4	5	6	7	8
LSF/BPF	-	-	-	-	-	-	+	+
pf	-	-	-	o	-	+	+	-
Netfilter	o	-	-	+	+	+	+	-
MAPI	+	+	+	+	++	+	+	-
Virtual Router	o	+	-	++	-	+	+	-
FFPF	+	+	++	++	++	++	+	+

Gigascope is a stream database for network analysis that supports an SQL-like stream query language that is compiled and distributed over a processing hierarchy which may include the NIC itself [25]. The focus is on data management and there is no support for backward compatibility, persistent storage or handling of dynamic ports.

Most extensive in its design is the SCAMPI architecture, which pushes processing to the lowest levels when possible [23]. SCAMPI borrows heavily from the way packets are handled by Endace DAG cards [4]. 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 through standard `pcap`, whereby packets are first pushed to userspace. It relies on BPF for its expressiveness and allows only a non-branching (linear) list of functions to be applied to a stream. The user is shielded from implementational details through a clean abstraction called the Monitoring API, or MAPI.

CONCLUSIONS AND FUTURE WORK

Having discussed in detail the design principles that we used to build a flexible high-speed packet processor, we briefly summarise them again in Table I. This list is not cast in stone, but in our opinion it represents a useful set of guidelines for development of packet processors. In Table II we show to what extent some practical or innovative packet processing frameworks are in line with these guidelines.

We will continue to refine these lessons by extending our framework. Two directions we are taking are using FFPF for the general kernel networking path and expanding its processing hierarchy to encompass distributed resources.

REFERENCES

[1] 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., Jan. 1993.

[2] G. R. Malan and F. Jahanian, "An extensible probe architecture for network protocol performance measurement," in *Computer Communication Review, ACM SIGCOMM, volume 28, number 4, ISSN 0146-4833,http://www.acm.org/sigcomm/sigcomm98/tp/abs_18.html*, Vancouver, Canada, Oct. 1998.

[3] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt, "Architecture of a network monitor. in proc. of PAM'03," 2003. [Online]. Available: citeseer.ist.psu.edu/moore03architecture.html

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

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

[6] RadiSys, "Enp-2611 product data sheet."

[7] P. Druschel, L. L. Peterson, and B. S. Davie, "Experiences with a high-speed network adaptor: A software perspective," in *SIGCOMM*, 1994, pp. 2–13. [Online]. Available: citeseer.ist.psu.edu/druschel94experience.html

[8] T. Spalink, S. Karlin, L. L. Peterson, and Y. Gottlieb, "Building a robust software-based router using network processors," in *Symposium on Operating Systems Principles*, 2001, pp. 216–229. [Online]. Available: citeseer.ist.psu.edu/spalink01building.html

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

[10] B. Chen and R. Morris, "Flexible control of parallelism in a multiprocessor pc router," in *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001, pp. 333–346.

[11] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: Fairly Fast Packet Filters," in *Proceedings of OSDI'04*, San Francisco, CA, December 2004.

[12] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Trans. Softw. Eng.*, vol. 17, no. 1, pp. 64–76, 1991.

[13] J. Networks., "Internet processor ii asic: Rate-limiting and traffic-policing features," http://www.juniper.net/solutions/literature/white_papers/200005.pdf, 2000.

[14] T. Nguyen, W. de Bruijn, M. Cristea, and H. Bos, "Scalable network monitors for high-speed links: a bottom-up approach," in *IPOM'04*, Beijing, China, 2004.

[15] J. M. Smith and C. B. S. Traw, "Giving applications access to gb/s networking," *IEEE Network*, vol. 7, no. 4, pp. 44–52, 1993. [Online]. Available: citeseer.ist.psu.edu/smith93giving.html

[16] A. Begel, S. McCanne, and S. L. Graham, "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture," in *SIGCOMM*, 1999, pp. 123–134. [Online]. Available: citeseer.nj.nec.com/begel99bpf.html

[17] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "Pathfinder: A pattern-based packet classifier," in *Operating Systems Design and Implementation*, 1994, pp. 115–123. [Online]. Available: citeseer.nj.nec.com/bailey94pathfinder.html

[18] D. R. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *SIGCOMM'96*, 1996, pp. 53–59. [Online]. Available: citeseer.nj.nec.com/engler96dpf.html

[19] M. Yuhara, B. Bershad, C. Maeda, and J. E. B. Moss, "Efficient packet demultiplexing for multiple endpoints and large m essages," in *USENIX Winter*, 1994, pp. 153–165. [Online]. Available: citeseer.nj.nec.com/yuhara94efficient.html

[20] G. R. 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, Oct. 1998.

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

[22] J. van der Merwe, R. Caceres, Y. Chu, and C. Sreenan, "Mmdump - a tool for monitoring internet multimedia traffic," *ACM Computer Communication Review*, vol. 30, no. 4, October 2000.

[23] 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.

[24] M.-L. Cristea, W. de Bruijn, and H. Bos, "Fpl-3: towards language support for distributed packet processing," in *Proceedings of IFIP Networking 2005 (accepted for publication)*, 2005.

[25] C. Cranor, T. Johnson, O. Spatschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*. ACM Press, 2003, pp. 647–651.