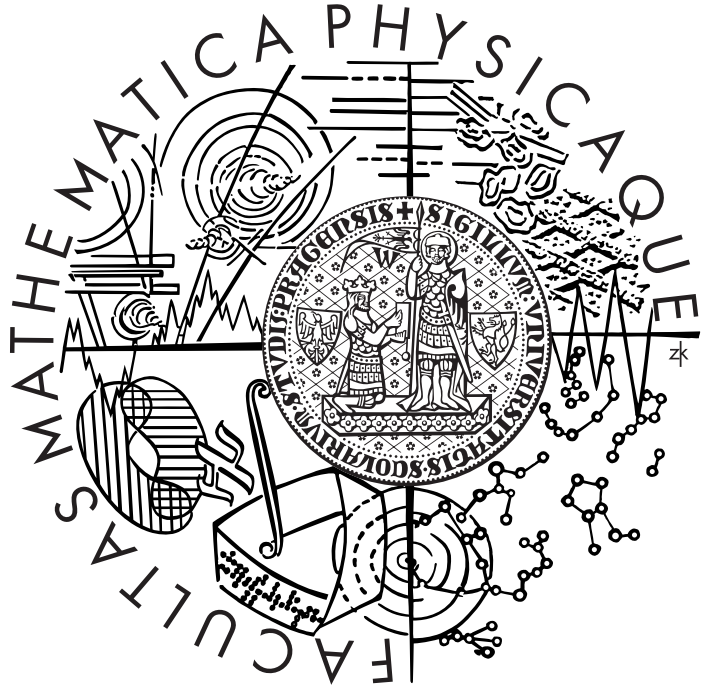
Charles University, Prague
Faculty of Mathematics and Physics

Master thesis



Tomáš Hrubý

Integrated Network Traffic Processing Framework

Department of Software Engineering
Study Program : Computer Science

Supervisor Vrije Universiteit Amsterdam : **Dr. ir. Herbert Bos**
Supervisor Charles University Prague : **Doc. Ing. Petr Tuma, Dr.**

Hereby I declare that I wrote this thesis by myself and listed all used sources. I agree with making this thesis publicly available.

Prague, December 19, 2006 Tomáš Hrubý

Acknowledgments

I would like to thank Herbert Bos, my supervisor at Vrije Universiteit Amsterdam, for his important comments and support. I thank Willem de Bruijn for his work on the FFPF project and his helpful remarks to my extension as well as Lennert Buytenhek for his priceless advice with IXP programming. Also, I would like to give special thanks to Petr Tuma for coordination with Charles University.

Abstract

Knowledge of what data are carried by network links is crucial to be able to prevent attacks and to improve quality of services. Therefore it is important to develop network monitoring tools which can operate on speeds of new gigabit networks. This thesis discusses general principles of designing a highly flexible framework which is divided into several levels. These spread across various hardware and software environments. This allows us to keep up with a gigabit speed. We show details on an extension of the FFPF framework to run on top of an IXP based PCI board. In addition, we present an implementation of Ruler, a language for packet pattern matching and data anonymization, implemented for highspeed traffic monitoring using IXP network processor. This work also presents performance evaluation, discussion of bottle-necks, general problems and compares with other related projects.

Contents

Chapter 1

Network monitoring and packet filtering

As the Internet, the world wide network, grows we experience a huge increase in the amount of transfered data. Also the kind of traffic changes significantly. The Internet connection became a natural part of our live. The effect is that there is only a small fraction of traffic on the Internet where it is easy to say what kind of data it carries. With fast growing speeds of network connections, people started to use internet to exchange large multimedia files, in many cases illegally. There is no simple way how to ban this. Nowadays P2P[1] networks allows us to share not only music but also compressed video. As there are no central servers and the protocols are proprietary, it is difficult to detect such traffic. Where in the past, most of the connections could have been classified by used ports (e.g., 80 for HTTP, 22 for SSH, ...), the P2P connections changes them dynamically. moreover gigabit networks are available and a gigabit network card is integrated on most newer motherboards. That enables sharing entire CDs and DVDs among users.

Another example of current dangers are spreading worms and viruses which attack computers all around the world. Preventing this in an early stage needs monitoring and filtering such data. Another challenge for monitoring is to find places of congestion which allows administrators to improve the quality of their services. Another case where the traffic filtering can help is a transparent redirecting of request to multiple machines with replicas or to dedicated servers (e.g., streaming servers) which results in splitting the load to many machines and thereby shortening response times and increasing quality.

Tools used for these purposes on 100Mbps networks are no longer suitable for use on a gigabit network. The amount of data is so tremendous that only re-implementing such utilities is still not enough. But not everybody can afford special hardware which is too expensive. Therefore we are pushed to use relatively inexpensive hardware together with current operating systems, to point at the bottlenecks and find solutions which can improve the overall performance.

Our project, FFPF (Fairly Fast Packet Filters), focuses on packet filtering and data delivery to monitoring applications as well as exploiting special network cards with network processors and using a commodity hardware together with a popular open source operating system.

[1] peer-to-peer

This work presents two different ways how to use network processors to gain the necessary performance. Both ways represents extrems of what such a hardware allows us to implement. As the rest of the work shows, each approach has its pros and cons and is suitable for a different range of task. The first one represents a higly flexible and general solution that enables using FFPF by unskilled administrators. The hardware is exposed as a set of execution units. It is more suitable in scenarious where a monitor consists of many simple tasks that are interconnected to for the final filter.

The latter approach integrates an IXP chip as a special purpose device, which is able to execute heavy-weight tasks. It may require skilled administration, however, the resulting performance is higher. Since the task is not split in many independent units and interconnections are hidden by the implementation, we do not pay such a high prize as by using slower general methods to pass data between parts of the framework. This work shows that even this extrem allows solutions that implments a single task which may be configured in many different ways as need be.

This thesis is structured as follows. In Chapter 2 we discuss the basic ideas used in the FFPF project and we give a higher level overview of its architecture. The implementation details of the extended FFPF are explained in Chapter 3, together with problems we were facing and how these were solved. Chapter 4 presents our performance evaluation of the extended FFPF. Chapter 5 describes how integrating an IXP chip as a special purpose device can increase performance. Reference application is described in Chapter 6 and its performance is presented in Chapter 7. Chapter 8 compares the FFPF framework to other related projects and Chapter 9 compares both ways of using an IXP and concludes.

Chapter 2

The FFPF monitoring framework

This chapter describes the internal architecture of FFPF, its building blocks and how they interact. First of all we give a brief overview of FFPF with description of how it works, later we explain important and interesting parts in more detail.

2.1 Intel IXP2xxx network processor

The Intel IXP network processors is a family of processors with a RISC[1] core and a set of RISC micro engines (MEs) that are specialized for packet processing. As mentioned above there was already an implementation for IXP1200 which has a *StrongARM* compliant core and 6 micro engines. The IXP2xxx [1] which we focused on in this work has the Intel *XScale* [2] core (ARMv5 compliant) Depending on the model, it may have 8 (e.g., on the IXP2400) or 16 (on the IXP28xx) micro-engines. In Intel terminology, these micro-engines are called micro-engines version 2. We will refer to their using ME as a shorthand. The XScale core has a clock-rate of 600MHz and is running an embedded Linux kernel. Every ME has its own instruction store and a local memory. There are various other kinds of memory available, which are shared among MEs and XScale. Each type of memory has different latency and abilities. By using them judiciously, they allow for an efficient implementation of filters running on MEs.

There are many reasons why FFPF can benefit from running its filters on MEs, for instance :

- High level of parallelism - independent filters can run in parallel on a different ME once packets are available
- High level of parallelism inside an ME - ME has hardware contexts with zero-cycle context switching and asynchronous memory access. That means if one thread is waiting for a signal (usually from memory to indicate that data are loaded into registers), the context is immediately switched and another ready-to-run thread does its work. Each context has its own set of registers and a program counter. It assumes that a filter is designed in a multi-threaded manner.

[1] Reduced Instruction Set Computing

- Various kinds of memory ranging from fast but small memory which is local for each ME, to larger and slower ones. Each memory also provides different features like atomic operations or hardware support for queues or rings.
- Pipelining - MEs are connected into a chain and neighboring engines can share registers. This enables them to pass data without going off-chip to memory. This feature was introduced in the IXP2xxx series. It was not available for the IXP1200. Programmers can write filters using more MEs and benefit from this. We allow filters to use blocks of MEs.
- Hardware support for expensive actions like hashing, CRC computation encryption and decryption.
- The XScale core or processor in the host machine will never touch packets which are dropped by ME filters. Therefore fewer packets are sent over the slow PCI bus.

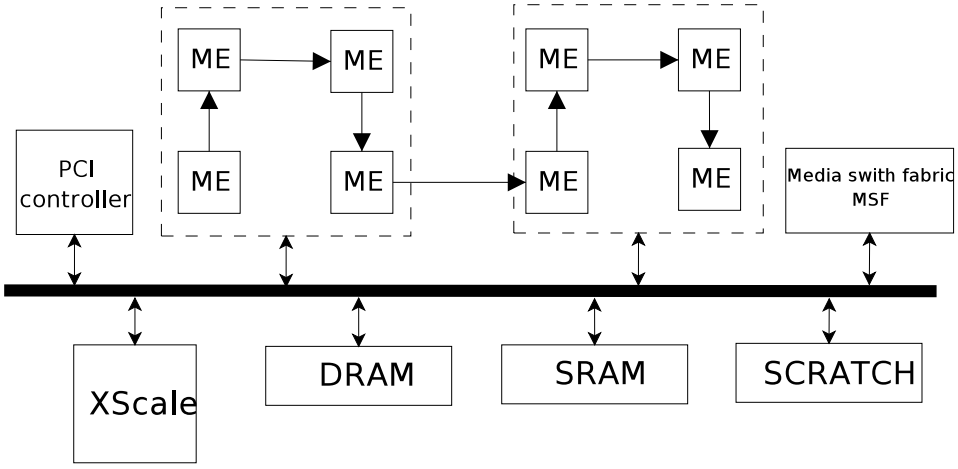


Figure 2.1: Scheme of the IXP2400

Since the XScale core is running Linux kernel, every filter originally designed for kernel space on the host can be reused on the XScale. This offloads the processor in the host machine and improves overall performance.

The IXP2400 NPU[2] is shown in Fig. 2.1. It presents two blocks of micro-engines, the XScale, various kinds of memory, the media switch fabric (MSF) and the PCI controller.

2.2 What is FFPF

FFPF as described in [3] is a framework for packet processing, network monitoring and traffic filtering at high speeds (gigabits per second) using the *Linux* operating system. It is new in its approach and has several features which make this tool well extensible and flexible.

[2]Network Process Unit

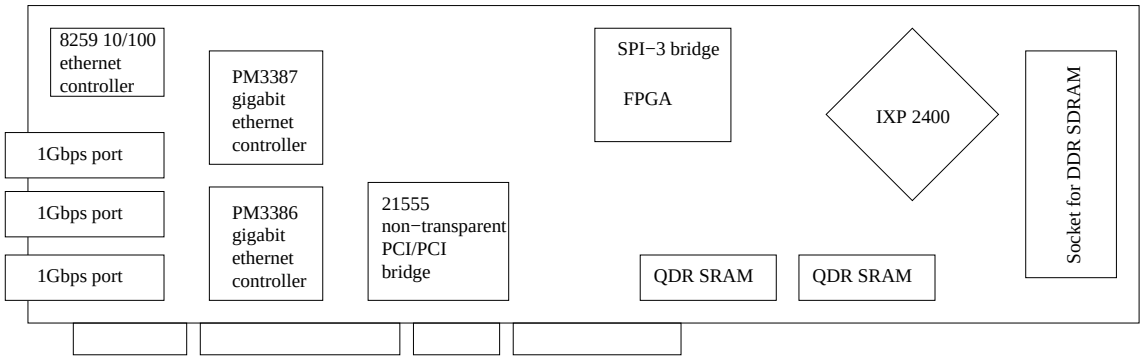


Figure 2.2: Scheme of the ENP2611 PCI board

Filtering is not based on a set of rules as in filters like *iptables* [4] or on a script as filters like BPF [5]. Instead, it uses the idea of independent filters connected as a filtering graph. Each node in the graph can be a different kind of a filter and the user can at any time implement his own new filter[3] and so extend the set of options which FFPF already offers.

The second important feature of our framework is the internal distinction between different levels (flowspaces) on which packets are processed. Each such flowspace has a different processing speed and security. From the user's point of view this is as transparent as possible. That is a filter runs in the fastest possible level where the desired filter is available. In the original implementation (by Willem de Bruijn and Herbert Bos [3]) there were only 2 levels. Filters were running either in the *Linux* kernel or in the userspace. The goal of this thesis was to extend the number of levels and to enable FFPF to run on the Intel IXP network processor. There was already a basic implementation for the IXP1200, which was very simple and unfriendly for users. Our new approach allows a transparent upload of filters into the IXP device. The actual implementation uses the IXP2400 Radisys ENP-2611 PCI card. We also support stand-alone IXP devices which are not plugged into PCI slots, e.g., a network router accessible via a network connection or a serial link. In such cases a different control mechanism must be used (e.g., an ethernet connection), but the idea and the design remains unchanged.

2.3 Architecture overview

FFPF can be divided into several layers. The topmost is the user interface which enables the user to create a filter-graph. A filter-graph is an expression describing relations among atomic filters and forms the final filter. Specifying a filter-graph could be done either on the command line or using GUI. Both result in a string that is passed to the *libffpf* userspace library.

For now we define "flowspace" as a level in the processing hierarchy that consist of nodes such as "userspace", "kernel" and "programmable network card". We will define flowspace more precisely in Section 2.4. *libffpf* represents the flowspace in userspace.

[3]the right name is a *filter-class*, the difference is explained in following sections

This library is the top of the flowspace hierarchy and all interaction with the rest of FFPF is done through this entry point and is passed to the following flowspace until it is handled or fails.

Every flowspace consists of flowgrabbers and filter classes. A filter class determines a filtering routine that can be used. On the other hand a filter associates a filter class with a filter expression. The expression tells the filtering routine how to act. For example a BPF [5] is a filter class that knows how to process BPF expressions. These two items together form a BPF filter, which will be a node in the filter graph. An expression can be empty, in that case filter class and a filter is the same and the action is fully determined by the routine itself (e.g., a packet counter). A filter in the flowgraph is known as a flowgrabber (Sec. 2.5).

Building a filter graph is divided into three basic phases :

1. *POPULATE* - this phase finds the correct place in the flowspace hierarchy for every atomic filter in filter graph. A filter can be populated only in the space where the required filter-class is registered.
2. *INSTANTIATE* - in this phase the nodes of a filter-graph are created and connected. Also connections across flowspace boundaries are built and structures from one space are mapped to the higher levels.
3. *ACTIVATE* - when all actions in the "instantiate" phase succeed, this phase activates packet processing, starts the appropriate MEs and sets the grabbers to accept packets from other grabbers that they are connected to or from devices like ethernet NICs.

The last building blocks are the buffers. Their purpose is to store data when crossing the flowspace borders. There are shared packet buffers (*PBuf*) and non-shared index buffers (*IBuf*). Some of the buffers are indirectly available to the applications and the data stored there can be retrieved if users are interested.

2.4 Flowspaces and flowspace hierarchy

The Flowspace is one of the main building blocks of FFPF. The original idea is to divide the filtering into different levels. Every level (See example in Fig. 2.3) has its own requirements and performance. For instance, running simple filters in the kernel saves cycles needed for context switching to userspace. On the other hand, running a complex pattern matching in the kernel decreases the overall system performance. The Linux kernel up to the version 2.4 was not preemptible and still it is only an option in version 2.6. That means, that once there is a task running in kernel context, there is no way to schedule other tasks on the same CPU. This leads to an overall slowdown of the operating system and user applications must wait longer for their CPU time slice. Note that "user application" may also mean that the "FFPF userspace part" reads packets from buffers too slowly. On the other hand, simple filters like packet counters or IP address matching could very well be placed close to the Linux network subsystem layer (*netfilter* [4]). Going to userspace and back to the kernel will take an extremely long time. This is definitely not desired behavior and introducing a hierarchical architecture is a possible solution.

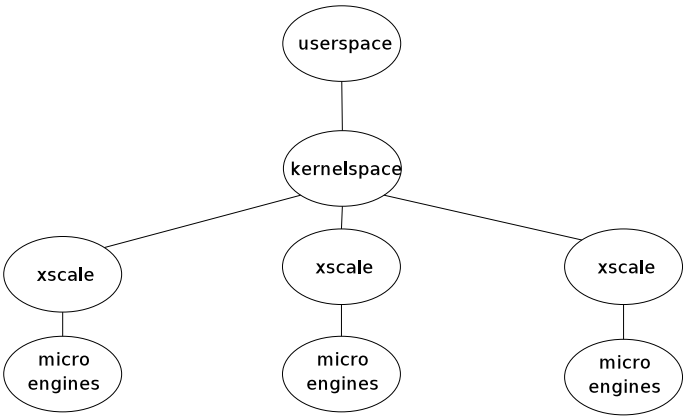


Figure 2.3: Flowspace hierarchy with 3 IXP spaces

Using FFPF, the user can decide which filterclass will be available in which space. He can directly influence the performance of the whole system.

A flowspace has an interface formed by several methods. These methods are designed to allow fine grained control over all processing and an interaction between particular spaces in the hierarchy. The list of methods (ordered as they are called in the FFPF filter life cycle) is in Tab. 2.1.

INIT	called when registering the flowspace
POPULATE	
INSTANTIATE	
CONNECT	
EXPORT	performed implicitly by *CONNECT* or explicitly by the user
MAP	
CLOSE	
DISCONNECT	
UNEXPORT	hidden inside *CLOSE* operation
FLUSH	optional
SHOW	optional
EXIT	called when closing the flowspace

Table 2.1: List of the flowspace methods

The key method is *populate*. It has a recursive nature. As we said before, we want to run a filter in the most appropriate flowspace. The first criterion is speed and the second is availability. By availability we mean not only whether the filterclass was registered, but also whether there are enough resources. We assume that a lower level is faster. It is closer to the hardware and so there is less software overhead. In the case of the IXP MEs the filter does not share its processing unit with others and it can use all the available cycles. The *populate* call is first propagated to the lowest level, where it tries to find the filterclass and checks the resources. In case that everything is alright, *populate* succeeds and returns a filter ID. Otherwise *populate* fails and backtracks to the next level up until it finds the right place or fails completely. One very important thing to mention here is

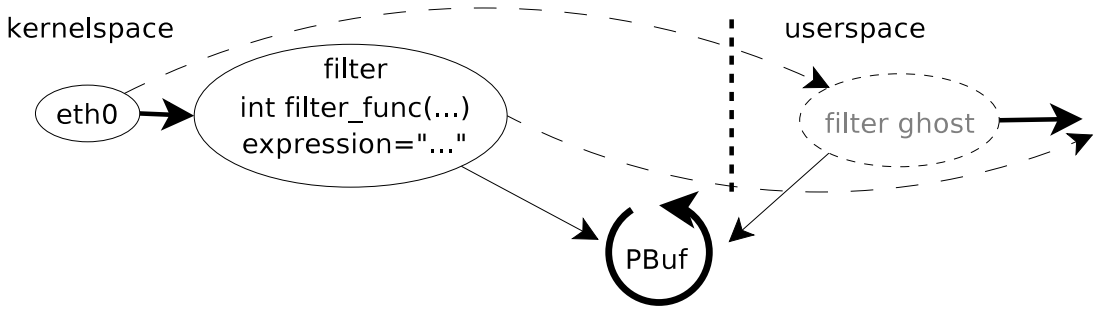


Figure 2.4: Connections between a flowgrabber and its ghost-image

that the *populate* call must act correctly in relation to other previously *populated* filters. Consider that a filter has input dependencies, then it cannot be *populated* in a flowspace lower than any of the filters on which it depends. The reason is that data can flow only upwards, from NIC to the userspace.

The role of *instantiate* is to create the actual filter, to allocate memory for a flowgrabber, to associate it with a filtering procedure and initialize all structures. In case of an ME filter it uploads the instructions into the instruction store of all affected MEs. Since FFPF puts nearly no restrictions on a filter implementation, this phase is also dedicated to retrieving the actual filter code if it is not yet present. It might involve a download of the ME binary file (proprietary code), or compilation of the source code (e.g. FPL3 [6]).

The methods *connect* and *export* belong together. Connect builds the graph. But it can connect only the flowgrabbers in the same flowspace. That is because of the fact that usually we cannot pass any pointers from space to space. Once a grabber depends on another in a different flowspace, we have to export the grabber to the higher level where the connection can be established. Two exported grabbers are never connected, but the call is propagated to the lowest possible level, where at least one of them is not exported, i.e. at least one of the grabbers "lives" in this space.

Exporting a grabber means making it available in the higher levels than where it was created (populated). If the FFPF user wants to access packets and gathered statistics, grabbers must also be exported (explicitly) out of the *libffpf*. A grabber is exported if and only if it is either a source for another grabber in a higher level (implicitly), or it is (explicitly) marked by the user as "to be exported".

Because we can export flowgrabbers, we also have to map them. We call the exported image a *ghost grabber* since it is only a transparent representation of a real structure. We use this on places where getting hold of the original object is not possible. This is done by the *map* call. Mapping is never issued before all grabbers are connected. This assures us that we know about all dependencies and constraints which can limit mapping options. This key issue is explained in Chap. **??** (Implementation).

2.5 Flowgrabbers

The most often used word in this work besides flowspace, is flowgrabber or just grabber. It is one of the main building blocks in FFPF. Our framework uses a graph structure and

a node in this graph is called flowgrabber. It can be considered a functional unit. It is associated with a filter(i.e., an instance[4] of a filterclass) and possibly with buffers. As explained in the previous section, a grabber can be exported to a higher-level flowspace as a ghost-grabber. There is no packet processing/filtering in the ghost-grabber. It was already done in the real grabber. But it is always associated with a buffer which is used to pass data between flowspaces. Fig. 2.4 shows the differences between the real grabber and its ghost image. Flowgrabbers also keep information about their predecessors and successors in the flowgraph. It creates virtual connections which are presented by the dashed arrows. They are used while processing a packet to pass it to all connected nodes. Also it is important when changing the graph structure (e.g., an application is starting/-closing its subgraph) to correctly count references. Each grabber can not be destroyed before all references are released even if this grabber was already explicitly closed by some application. On the other hand it must be freed once it is no longer used, to return its resources to the system. The connection between the real and ghost grabber is done by sharing buffers. Flowgrabbers are the most important internal structures and are discussed exhaustively in the following sections.

2.6 Buffers

One of the crucial things in packet processing is data copying. The best we can do is to filter all packets "in place". Unfortunately this is not always possible. There are different ways in which packets can be received from network devices and how they can be passed to different *flowspaces*. The original design assumes that every received packet is saved into a circular buffer and that every filter gets an index into this buffer. This approach was first given up only to be reinvented later on. As the main target platform is an off-the-shelf PC with the *Linux* operating system we have to deal with the current Linux networking layer - *netfilter*. This subsystem manages its own buffers into which packets are stored after arrival. Users can implement so-called *hooks* into this layer. All hooks are called one by one and get references to packets stored in these buffers. By incrementing reference counter on packets, we can lock them in memory an they will not be deallocated until the hook finishes its job and returns the packet back to the *netfilter*. There is no need to copy packets to other locations, since it would introduce unneccesary overhead without benefit. Therefore every packet received by our `netfilter_hook` is passed as it is to the flowgrabbers in the *kernelspace*. We wrap it into our own `struct packet` structure where additional information is stored, such as whether the packet was already saved , and if so in which buffer and on which position. Once the packet is stored, any consecutive filter knows about that and does not store the same packet again. It does not only saves cycles, but also prohibits multiple occurrence of one packet in one packet buffer or an unexpected packet in a different packet buffer.

Problems emerge when packets have to be sent to the userland. There is no way to pass kernel-valid pointers to userspace. We can only copy data (Linux provides primitives for that) or map some memory area. None of these options can be used without intermediate step. As a result the current implementation stores such packets in an extra buffer which is

[4] a filterclass together with an expression

memory-mapped into userspace. Details on memory mapping are presented in Chap. **??**. We have no extended FFPF to cooperate with "intelligent" devices like IXP network cards, packets are no longer received only through *netfilter*. E.g., there is no common driver for the ENP-2611[5]. Such a card has so many options how to handle in/out going traffic that the manufacturer only provides an SDK and users have to build their own solution which suits their problems the best. Therefore the host-kernel needs another way to get data. This lead us to embrace again the previously abandoned idea of storing all packets into buffers immediately after they are accepted by the system (on micro-engine) and the Linux kernel uses a poller to read these buffers and bypasses the *netfilter*.

There are basically three kinds of buffers in the FFPF system. One of them is the so-called *MBuf* which is just a chunk of an unstructured memory, assigned to a filter which can fill in various data. When exported, this memory can be read by the application and its content is presented to the user. Since the interpretation is dependent on its filterclass and the mapping is done in a similar way as the other buffers, we do not consider it as an interesting topic for deeper explanation. Further on, if not mentioned explicitly, we talk only about the packet buffers (*PBuf*) used to store raw data and the index buffers (*IBuf*) to handle indexes of packets classified as accepted by filters. *PBufs* are shared as much as possible to avoid any copying. But there may be good reasons to move packets to other packet buffers. These are described in Sec. 2.7, so we just mention them shortly here. We may want to separate processing done by a normal user and by root. There could be security issues to consider, e.g., once FFPF is extended to enable firewalling, routing, NAT, some processes will want to change packets but the data that are processed by others cannot be changed, etc. In contrast to the shared nature of *PBuf*, every flow-grabber which stores packets into a packet buffer keeps track of them in its own index buffer. Index is a small structure which holds references into *PBufs* together with the value assigned by the last filter. This gives the opportunity of finer-grained classification than the boolean `true/false` pair. It also keeps unique identifier of the packet buffer in which the actual packet resides.

A buffer is a circular structure where there is one writer followed by many readers. Writers can be simple, with every packet saved, the *writeindex* is incremented by one. Similarly a reader has only to check if its index is still not equal to the writer's one to find whether data are available. On the other hand this also brings some difficulties. As every integer type has limited size we have to deal with overflow when incrementing. With a 32bit counter that wraps every 4G of packets and with network speeds in the order of several Gbps, it takes at most few minutes from start of processing to the overflow. One simple solution is to use 64bit counters. These can represent such a huge number of packets that its upper limit is no more a big issue. However it brings another problem which is no less serious : atomicity. Readers and writers are concurrent processes and we are not allowed to make any assumptions about reads and writes serializability and 64bit read/write operations are not atomic on most of today's 32bit architectures. On the other hand 32bit operations are atomic as long as they access memory fields aligned to 4byte offsets. Any kind of locking is out of the question since the overhead is huge. Linux provides us with atomic operations in the Linux kernel. Unfortunately they do not apply

[5] http://www.radisys.com/

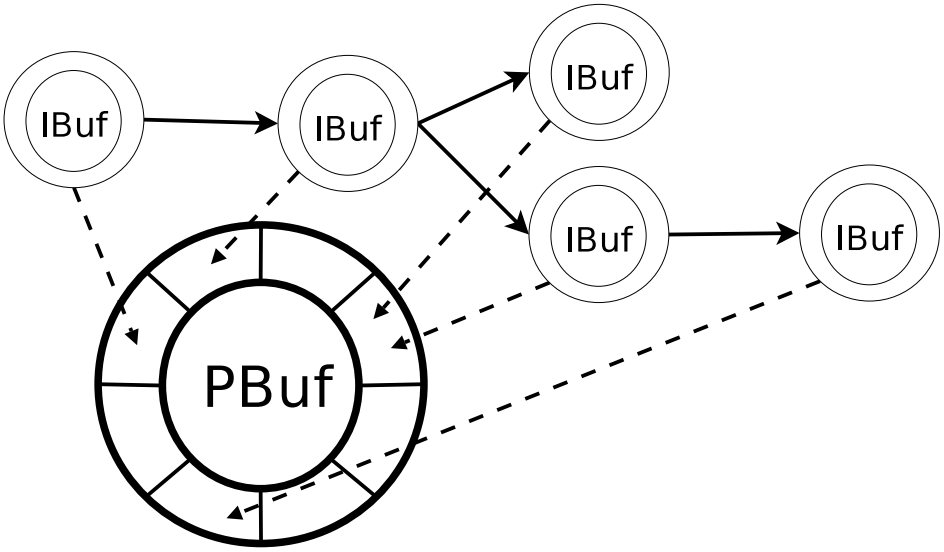


Figure 2.5: Old version of flowgroup. All IBufs points to one shared PBuf

outside of kernel space. Moreover, in the case of the IXP NIC, the writer runs on a ME and the reader may be a userspace process, in which case reads and writes are separated by the PCI bus. Aligned 32bit r/w atomicity holds across the PCI bus and this is all we can be sure about.

Considering all pros and cons we maintain the indexes on aligned addresses to make 32bit r/w operations atomic. Writing 4G packets to a buffer between two consecutive reads is not probable, so we can count indexes modulo 4G.

2.7 Flowgroups

FFPF can be used by many users on a single machine. Every application can have different requirements, permissions and goals. We believe that FFPF will become a subsystem for many different networking applications, starting with passive monitoring and continuing with active network processing, where there is a need for a fast packet delivery and work distribution on various levels or machines. Nodes in a flowgraph do not necessarily have to be filters but can change content, headers, etc. This can lead to applications like firewalls, routers, NATs, spam-filters and others. All these possible uses can interfere with each other and for this reason we need good separation. This is achieved by creating so called *flowgroups*.

The idea is that every flowgroup has its own copy of a packet received from the network and processing of such a packet does not influence data in other groups. How large such a group is, depends on the FFPF administrator and its configuration. There are basically three options but more can be added very easily. These three are :

1. *global* - there is only one global flowgroup in the entire system which contains all processes. This is a good choice if there are no clashes, e.g., on machine used for one specific task. It saves multiplexing (copying) of data and saves memory too.

2. *user dependent* - to which flowgroup a process belongs is determined by its user-information. There are two sub-categories. One puts together processes of one user, based on `UID`. The other sub-category shares data for one user-group (`GID`).

3. *PID* - every process has its own copy of packets. This is good in case of a small amount of applications with totally different tasks. Overhead grows with an increasing number of processes.

It is worth mentioning that the (statically linked) *libffpf* separates applications as well. So the upper mentioned rules apply in shared spaces, namely *kernelspace*.

A *flowgroup* does not have a real representation inside FFPF. In the original implementation there was a bijection between packet buffers and flowgroups. In other words, there was always only one packet buffer in one flowgroup and all the items in index buffers were pointing to its data items.(Fig. 2.5) This was sufficient when a user was able to build only a limited set of flowgraphs and packets were received by means of hooks into *netfilter*. *PBufs* were assigned to flowgrabbers at the moment of exporting (to userspace or outside of FFPF). At this moment it was detected whether there already existed such a flowgroup and if so its packet buffer was reused. If not a new data storage was created. This is equivalent to the creation of a new flowgroup.

The drawback of this design popped up when we allowed multiple sources to be connected to one sink. Since there was no explicit administration about which flowgrabber is in which flowgroup and the main idea is to share/reuse as much as possible, it leads to situations as shown in Fig. 2.6. What happens if one application populates flowgraph *a1)* and then another application populates flowgraph *a2)* with the same subgraph? Basically everything is fine until filters F1 and F3 are exported. (We assume that both parts are populated in the same space, e.g., kernel-space) When F1 is exported, a new *PBuf* is created (P1). What should happen when F3 is exported also? If we reuse the P1, then the second flowgroup can reach packets accepted by the other flow, absolutely unauthorized. Even so everything still works. The arrow from F1 to F3 on b) shows the connection which is created between different flowgroups (dashed boxes).

The real problem occurs when F2 was exported with a new P2 buffer. P2 is created because there is no buffer in this flowgroup associated with the F2 predecessors and packets arriving to F2 are unsaved. But then there are two *PBufs* in one flowgroup, one of them is included also in different one. More over, packets arriving in node F3 are already stored, in P1 or P2, so F3 saves only an index into its *IBuf* (I3). Where should the reader of F3 output (some poller in userspace) get the packet from?

The previous version of FFPF was dealing with this problem in a quite specific way. When a flowgrabber was being exported, it checked its predecessors to see if they already have assigned buffers. If there were none, a new one was acquired. If they were all the same, it was reused. But if there were more of them, a problem was detected. This situation was rare and unexpected, only leaves of a flowgraph were exported and no inner nodes. In that case FFPF chose one of these buffers and the connections to predecessors with other packet buffers were canceled. This was not reported to the user and his results of filtering were wrong. Unfortunately, when more than one flowgrabber sharing the same buffer in kernelspace were exported to userland it appeared as different buffers (as

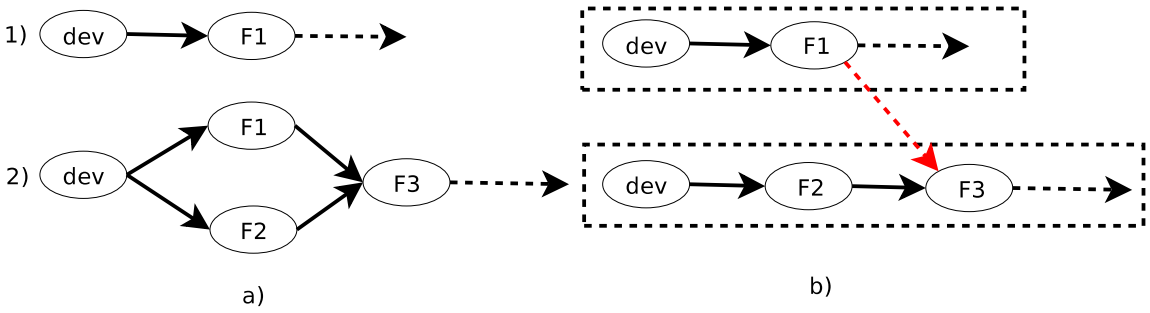


Figure 2.6: Packet buffer clash between flowgroups

explained in Sec. 3.5.1) which meant that it also broke other correct connections.

Since part of this work is not only to make FFPF running on IXP processors, but also to enable FFPF to handle more complex graphs, all buffer and flowgroup management was redesigned. How is it done is described in detail in (Chap. **??**). Here we mention only the main new features in this major change. The main point is that we no longer receive data only from *netfilter* hooks. For instance packets are also received by IXP MEs. Many packets are already saved before being processed in *kernelspace*. As network processors are becoming more popular nowadays and we want FFPF to be able to cooperate with them, we expect that there will be multiple different buffers on distinct places inside FFPF in the future.

There were basically two ways to advance toward a better solution. Both of them have one thing in common. There must be more packet buffers included in one flowgroup, therefore flowgrabbers must be able to reference them and as a result we can no longer distinguish if a node is in some flowgroup by the packet buffer with which it is associated. Also, both of them have to deal with merging data from different sources. The first option is to allocate a new buffer whenever we detect that the predecessors do not have the same PBufs. It may look like a straightforward approach. But there is a danger that packets will be saved into more than one packet buffer inside one flowgroup and we have to pass this information together with real data. The second option is to store packets only once, save information about the *PBuf*, in which it is stored, into an index structure in *IBuf* and associate all relevant packet buffers with each node. But the packet buffer identifier from an index must be translated into a valid pointer to *PBuf*. After considering benefits from both options, we decided for the latter. The most costly operation is data copying which is hereby minimized and traded for the id-to-pointer translation which is cheap if well designed. It seems to be a cleaner and more robust solution as well.

Since there are more *PBufs* in one flowgroup, the flowgroup has now become an absolutely abstract construct. We incorporated it into identifiers which are assigned to each flowgrabber at creation time. Later on only these IDs are passed into the FFPF calls and situations like the one depicted on Fig. 2.6 never happen. A user gets only the handle of the exported grabbers. With the new ID design (described in Sec. 3.2) t situation is more clear as shown on Fig 2.7. In case a) there are two graphs in the same group sharing the same subgraph (highlighted by the thick line) which is allowed and saves resources

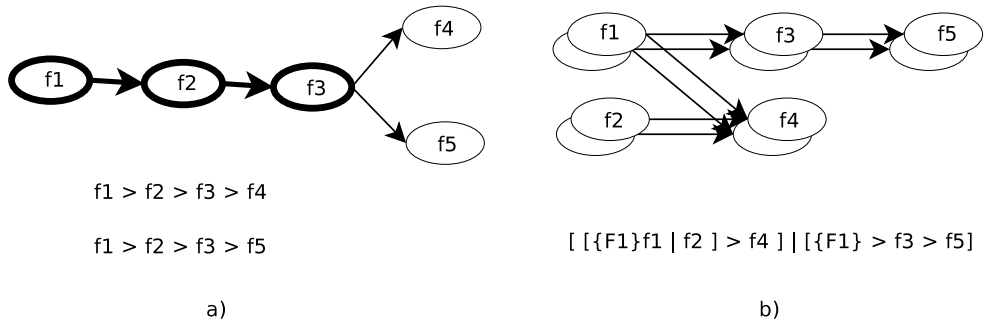


Figure 2.7: Graphs in the same and in a different flowgroup

as well as processing time. In case b) there are two graphs created by the members of different flowgroups. Because of that this graph is created twice and are both absolutely separated.

2.8 FFPF API

FFPF provides a simple API to the user. His only duty is to build an abstract representation of the required graph. He does not have (and does not need) to access any elements (like atomic filters) and does not need to connect manually filters together. In contrast with *libpcap* and its set of routines, this gives a simple and powerful filtering and monitoring tool even to unexperienced users and programmers. By linking *libffpf* into a project we get access to primitives which initiate and close library, starts, pause and stops processing. This would be nothing without access to packets which were accepted by all the filters. Therefore we provide an API for data-access too.

Filtering expression

The entry point to *libffpf* is the `ffpf_open` function where the user passes the desired filtering expression to FFPF. To accommodate the growing abilities of FFPF we have implemented support for a more flexible and complex syntax. An input expression consists of "filters" and their positions in the filter graph. We kept the language model from the previous version which uses two possible relations. Filters are either dependent or run in parallel. To mark this we use signs derived from the *Unix shell* : pipe '|' for parallel processing and redirection '>' for dependency. This is familiar to Unix users even if the meaning is slightly different. The differences become clear even in a simple serial filter with several stages. Unlike the shell where the '|' construct would be used, this relation is expressed with the '>' sign. We believe this makes sense since it redirects output from one filter to the input of another. In the graph language this operation builds edges. More complex graphs require branching. To redirect output from one filter to multiple other ones, '|' have to be placed among them.

Only these two operators would not be enough to build tree graphs which are important, for instance, filters listening to multiple devices. One way to solve that is by intro-

ducing the operators priority and associativity. Both operators are left associative. The result is presented in Fig. 2.8b. It corresponds to the expression "`((dev1 | dev2) > f1) > f2`" or "`(dev1 | dev2) > f1 > f2`". Both of these are more clear and the user also knows for sure what is the result populated inside FFPF. In example (c) a user may be surprised that he gets "`((dev1 > f1) | dev2) > f2`". This is totally different from "`(dev1 > f1) | (dev2 > f2)`" which may be more logical and was probably on the user's mind. In FFPF '`|`' has higher priority than '`>`' and '`[ ]`' can be used to make associativity explicit(e.g., `[dev1 > f1] | [dev2 > f2]`). By this the result is uniquely determined. The round brackets used above are not a correct part of the input expression and are used here only to help to show how the nodes are associated. Round brackets are used to enclose one node, here represented by `f1`, `f2`, etc. More examples of the correct form are shown later in Fig. 2.9 after all the features are explained.

Every atomic filter (`dev1`, `f2`, ...) can have a more complex structure. As stated before, FFPF can handle various kind of atomic filters which are known to its framework. To enable this the user has to provide more information. Every node in the graph is represented as comma separated list of parameters, enclosed in parentheses. The first item is special and mandatory. It tells the filterclass of this node. There are classes for which this information is enough. Examples are basic filters like `accept` or `drop` (which accepts/drops all packets) or a specific users implementation (`my_filter`) of a filter where its class name includes all other information (e.g., `tcp_on_port_1234_outgoing`). On the other hand, the idea of filterclasses is an opportunity to write one filter and use it in many different situations. There are several examples, ranging from simple to complex :

1. a basic class which represents network devices, on figures denoted as "`dev`". It expects an identifier of a physical device. This could be `ethN` for a particular ethernet NIC, `ixpN` for a filter running on the N-th micro-engine of an IXP1200, `ixp2xxxN` for receiving packets from an IXP2xxx, etc.

2. filters that need preallocated memory in order to work correctly and which can subsequently be exported so that users can read partial results. Example in this class are packet counters or byte counters.

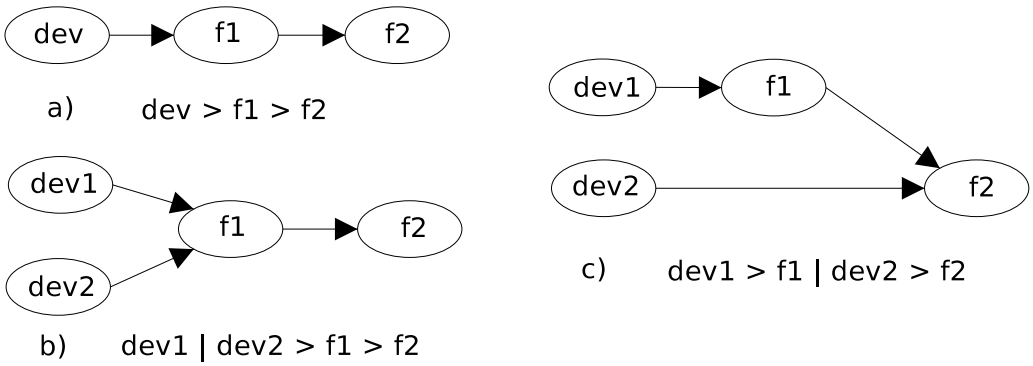


Figure 2.8: Basic filter expressions

3. virtual machines like *BPF* [5] where a user provides expressions in the BPF language.
4. just in time compiled filters, e.g., FPL1, FPL2 [7], FPL3 [8]
5. pattern matching filters (e.g., Aho-Corasick [9])
6. filters based on various tables, e.g., firewalling tables
7. hints about where a filter should be populated, ...

As one can see, there is such a large variation of filters with very different requirements that placing limits on the number and the amount of parameters is counterproductive. For all these reasons we opted for *name = value* pairs. We don't even require a *name* to be unique. The filter specific handler receives the full list. It may handle them in any way it sees fit. The list of currently used parameters and their meaning is summed up in Tab. 2.2

`expression`	This parameter corresponds to filter expressions in the original implementation [10]. Users can register *translators* for each filter class. To a translator is passed the "`expression`" which it translates into a representation expected internally by the filter. Currently used by the *FPL-1*, *FPL-3* and *BPF* filters.
`exported`	As described in Chap. **??**, flowgrabbers are not visible on higher levels than the highest level at which they are used. For that reason every grabber needed outside of FFPF (e.g. because the application needs the results) must be marked as `exported=y`. This also gives an opportunity to users to export non-leaf grabbers. This was not possible earlier. One example of how this is used is a scenario in which a packet counter is placed between filters to see how many of them are dropped later on.
`mbufsize`	Minimum amount of memory required by a filter. This memory is allocated, managed and mapped to userspace by FFPF at instantiation time.

Table 2.2: List of currently used parameters

The filter-expression syntax as described so far can handle only a limited set of directed acyclic graphs (DAG). The limitations are caused not only by the used relation operators, but also by the expression processing. We parse input string using the *Lemon* LALR(1) parser [11]. This was sufficient for the previous (simpler) syntax. A LALR(1) parser can easily handle trees as it traverses the derivation tree of the input. To handle more complex DAG structures and still keep the filter human-readable, we introduced special *tags*. A tag is a token in curly brackets '`{ }`' prepended to atomic filters[6]. The

[6]Note that the "{}" is currently being replaced with an "identifier=tag" parameter to make it more consistent with the rest of the input language

tag can be any identifier consisting of alpha-numeric characters up to a length of 32 characters. We first used this with expressions that were auto-generated by a graphical graph editor. In this case, incremental counters were used as IDs. As this is not convenient for human users, more meaningful tags are also allowed. Moreover, there is no need to repeat the parameter list and the "`{tag}()`" syntax can be used when referring to other nodes in the graph. An example is shown in the last row of Fig 2.9. These tags are also used to get a reference to a grabber from the list of all exported flowgrabbers.

Once such a tag is discovered while parsing, the node is recorded in a parser lookup-table. Whenever a *tagged* atomic filter is found, its tag is compared with the records in the table and in case of a match, the "old" node is reused instead of creating new one.

In this way the users can create filter-graphs of varying complexity. It is important to mention that FFPF does not check whether the received graph is meaningful (e.g., whether there are branches with non-empty intersection of flows). FFPF is only responsible for a correct delivery of packets to filters. It knows nothing about which sort of data can be accepted by which filter. This could lead to "multiplication" of packets in FFPF if a packet is accepted by more branches and than goes to one sink more then once. It is the user's responsibility to build correct graphs so as to avoid this kind of problems.

$(device, expression = eth0) > (accept)$
$(device, expression = eth0, export = y) > (accept, export = y)$
$[(device, expression = pcap0, export = y)|(device, expression = eth0)] > (accept, export = y)$
$(device, expression = eth0) > (sampler, expression = 2, mbufsize = 4, export = y)$
$[\{pcap\}(device, expression = pcap0) > (accept)|\{pcap\}() > (debug, export = y)] > (drop))$

Figure 2.9: Input expression examples

2.9 FFPF programming language (FPL)

This section is base on "FPL-3: towards language support for distributed packet processing" [6]

The FFPF programming language (FPL) was devised to give the FFPF platform a more expressive packet processing language than available in existing solutions. The FPL-2 conceptually uses a register-based virtual machine, but compiles to fully optimized object code. It supports all common integer types and allows expressions to access any field in the packet header or payload in a friendly manner. An extensible set of macros implements a shorthand form for well-known fields, so that instead of asking for particular bytes, a user may use 'IP_PROTO' to get IP header's protocol field, for instance. Moreover, offsets in packets can determined by an expression. Execution safety is by virtue of both compile-time and run-time boundary checks. Most of the operators are well-known from C language. FPL-2 supports conditional branching (`IF ... THEN ... ELSE`), loops (`FOR`), hash function (optionaly implemented in hardware), external functions (efficient C or hardware implementation) and packet transmission.

However FPL-2 was designed for single node processing. Its direct descendant FPL-3 extends FPL-2 with constructs for distributed processing. `SPLIT()` construct tells the

compiler that the code following and bounded by its `TILPS` construct can be split off form the main program.

The FPL-3 language hides most of the complexity of the underlying hardware. For instance, users need not wory about loading data into ME's registers before accessing them. Similarly, accessing memory which is not byte addressable is handled automatically by the compiler.

The FPL-3 compiler generates straight C target code that can be further handled by any C compiler. Programs can therefore benefit from the advance optimisers in the Intel micro-C compiler for IXPs and `gcc` for commodity PCs.

FFPF was extended in such a way that it can transparently compile FPL sources to micro-engine object files which can be automatically loaded and started.

A short FPL example :

```
IF  (PKT.IP_PROTO==PROTO_TCP)
        IF(PKT.IP_DEST_PORT==80) THEN
                "http␣processing"
        ELSE IF(PKT.IP_DEST==25) THEN
                "email␣processing"
        FI
ELSE IF(PKT.IP_PROTO==PROTO_UDP) THEN
        "udp␣processing"
ELSE
        "other␣processing"
FI
```

Chapter 3

Implementation of the extended FFPF

This chapter describes implementation issues. As a developing platform we use an x86 compatible PC, Linux kernel version 2.6 and GCC version 3.x. The GCC compiler allows us not only to compile for the host's x86 CPU but also for the XScale processor, using cross-compilation.

As mentioned earlier in Sec. 2.1 on page 8, the IXP1200 was already used by FFPF. However integration into the FFPF framework was quite limited. It was only running ME filters which were manually preloaded by the user. There was no mechanism that allowed FFPF itself to tell which filter was uploaded and started. The IXP1200 was an autonomous system and the only possible way of interaction was polling on buffers in the IXP memory mapped into the PCI space. Our goal was to enable FFPF to take full control of the IXP device (Radisys ENP-2611 PCI card, see Fig. 2.2) , to put decision logic onto the card and to make it transparent to the user. There were several issues to be solved :

- Most important was that the solution had to fit into the original design by extending the flowspace hierarchy by (an)other space(s)
- We had to extend the control mechanism so that it can cross different boundaries like the user-kernel space boundary, the PCI bus and the network.
- The performance in the data flow had to be improved. The original IXP1200 implementation allowed 3 distinct copy-policies [3] across the PCI bus. We concentrated on improving these and adding one more, in which the IXP device pushes data into the host memory, DMA[1]-like.
- We had to incorporate a resource manager. It handles managing the limited number of micro engines as well as the limited amount of memory, which has to be shared among MEs and XScale.
- The original input language and user interface needed a complete overhaul.

3.1 Flowspace hierarchy

As explained in Sec. 2.4 we introduced the notion of a multi-level hierarchy of flowspaces. Here we focus on the design and the implementation of such a hierarchy. Our

[1] Direct Memory Access

point was that we want to allow spaces not only on a single machine, but possibly on a distributed system and with different endianness.

A flowspace is in fact represented by two structures. One is a stub which manages exported structures and communicates with a receiver. Communication can be done via special control devices (between userspace and kernelspace), via the PCI bus (kernel-IXP), LAN connections (userspace-IXP), etc. The receiver module passes data to the *localspace* and the stubs of other spaces. It tries all stubs and *localspace* one by one (dashed arrow, Fig. 3.1), until it succeeds or possibly fails. Every stub has its own list of grabbers which it represents. Real flowgrabbers (as opposed to ghost-grabbers) live only in *localspace*.

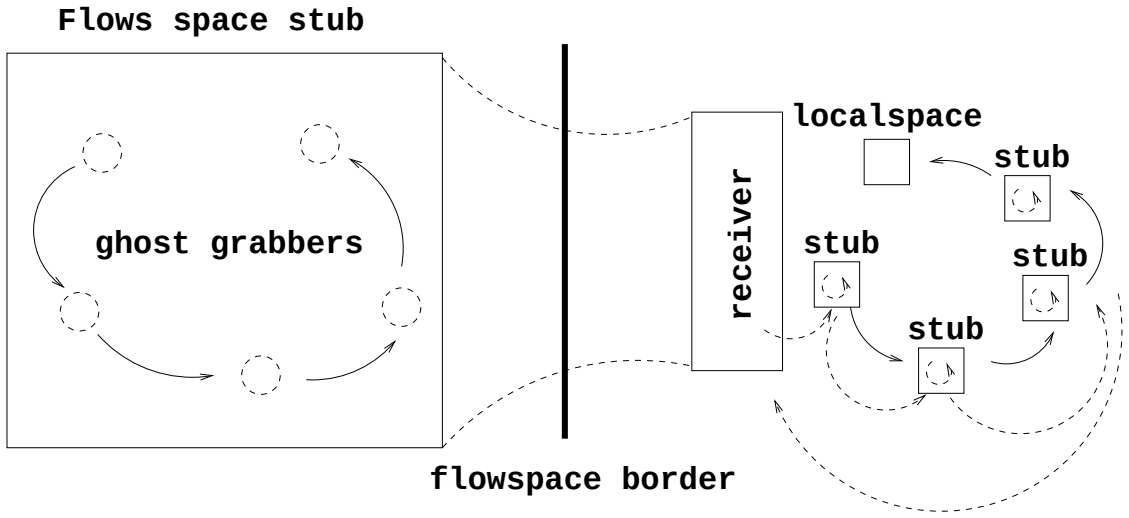


Figure 3.1: Relation between localspace and stubs

The communication between a stub and the next level is handled by code which serializes, sends, receives and deserializes requests and replies. (de)Serialization is common for all boundaries. Sending and receiving is modular. So we can send request a from userspace to the kernel as well as from kernel on the host machine to FFPF running on the IXP card across the PCI bus. There is no obstacle to using FFPF in a distributed manner and using the network as a communication channel between different FFPF flowspaces.

In our experiments, the flowspace hierarchy was purely linear. But this is not a necessity. For instance, there may be more IXP cards where each represents one *ixpspace*[2] or the user may want to receive packets from other sources. Every device is represented by one stub in kernel-space and the resulting hierarchy is non-linear.

3.2 Populate

The first step in creating a filter in FFPF is a *populate* call. As mentioned before it has a recursive nature. It's goal is to find the most suitable flowspace for a given filter. Our first

[2] the XScale and micro-engines

criterion is speed. So on every level we have a list of all spaces visible from that point ordered by speed. We simply take the first one, and pass our request to this flowspace. If it returns failure (or rather - not handled) we try in a loop each consecutive one until we succeed. Naturally it can happen that no flowspace from this list can handle such a request and so it fails on this level and returns to the next level up.

Populate not only returns a simple filter ID which is later used to build a grabber's unique ID, but also an internal flowspace identifier in which this call succeeded. So when later populating a filter which depends on this one, we can use this flowspace identifier not to go deeper than the level at which this grabber was instantiated. This satisfies the condition that data flows only upwards in the hierarchy. Another strong requirement is that dependencies are on the same path from the root (userspace). If a filter can be populated on each XScale in the system, we have to populate it on the correct one. That means on the same XScale, where it has its data sources. Otherwise it would not be possible to connect these filters and populating of the whole filter graph fails. We already mentioned that our hierarchy was linear since we have only one IXP card in our system. This can vary, and we want to handle such cases correctly as well.

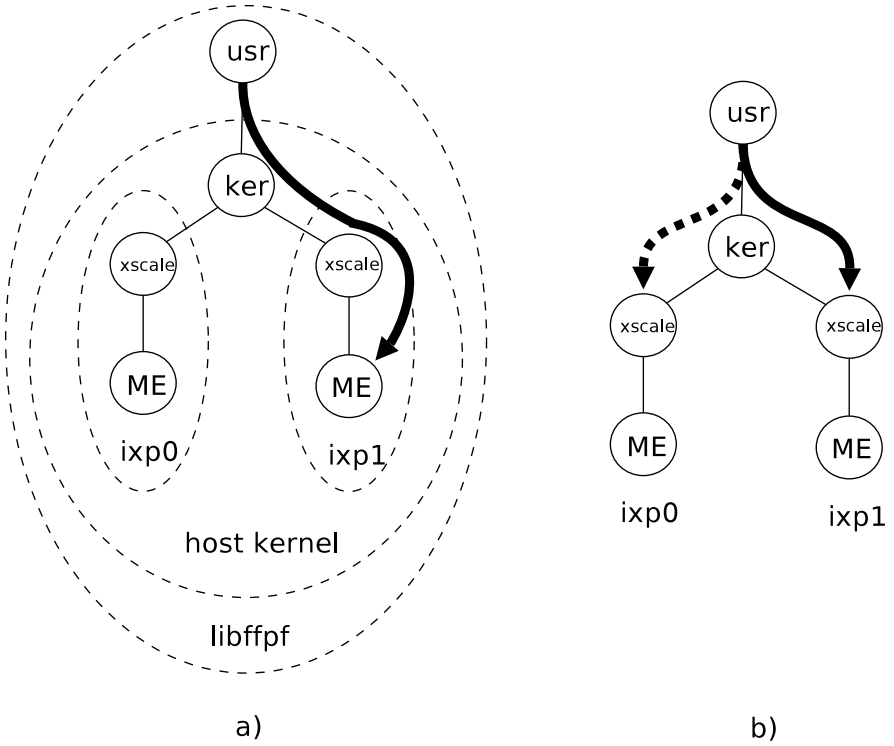


Figure 3.2: Population in a complex flowspace hierarchy

Before, it was very simple. There were only two flowspaces and both were directly visible from *libffpf*. The easiest was to remember in which of these two spaces the call succeeded. This is not the case now. As shown in Fig. 3.2a, lower spaces are overlapped by higher-level. In other words, lower spaces cannot always be seen from the higher ones. All that is directly visible is the space immediately beneath in the hierarchy. So for

instance *ixpspace* is not visible from *userspace* nor is *micro-engine-space* visible from *kernelspace*. It is encapsulated in *ixpspace* instead. This configuration has some benefits and also some disadvantages. We focus here on the latter. Suppose there is more than one "intelligent" card like the IXP in the system, and we want to run the same set of filters on all of them. The original design of `populate` simply goes to the first one (let's call it `ixp0`) and as the filterclass is found there, it returns. If we wanted to populate the same filter on the other card (`ixp1`) it tries the `ixp0` first. Again it succeeds and therefore it never reaches the `ixp1`. Our goal was to develop a simple and transparent solution to this problem. If there is a grabber populated on `ixp1` as shown by the thick arrow in Fig. 3.2a, then all other nodes which are dependent on this one are populated on the "solid" path (Fig. 3.2b) and not on the "dashed" one, which is "prohibited". Fig. 3.2a also shows how lower-level flowspaces are encapsulated in higher levels.

This apparently leads to wrong results. A typical problematic example is a simple expression like `(ixp0 | ixp1) > f1`. This would be populated correctly if and only if `f1` exists only in *kernelspace* or higher. But we cannot assume this and moreover if there is a filterclass for the kernel on the host machine, it is enough to recompile it to get a filterclass for the *XScale*. For this reason it is a serious issue and a solution had to be found.

We can imagine two different "correct" ways of populating such an expression. From a performance point of view the better would be to create two filters `f1` for each device. But we also have to consider that this filter may gather some information (statistics) which is stored in the *MBuf* and this would split it into more buffers. Moreover, one may also export this "multi-grabber". The application would have to be aware of this. For this reason we do not treat it as a correct result. The second and the right solution of this problem is to populate `f1` in the *host-kernelspace*.

For these reasons we extended the flowgrabber IDs by a special field called *flowspace_navigator* which uniquely determines the path. Details are in the next section.

Flowgrabber ID

The flowgrabber IDs have changed significantly in the new version of FFPF. As we already mentioned, `populate` was returning the filter identifier, a 32bit number calculated as a hash of the filterclass and the filter expression. Before calling `instantiate` a prefix to this value was added to build the final flowgrabber ID. The prefix was a string containing the filter identifiers of *all* predecessors. It was internally used for connecting a new grabber to its data sources. The requirements placed on ID have changed, so its internals need to be changed too. The new ID is defined as follows:

```
struct flowgrabber_id {
        uint32_t filter_id;
        uint32_t flowspace_navigator;
        uint32_t flowgroup_id;
        uint32_t var_len;
        uint32_t var[];
};
```

Flowgrabber ID consists of a fixed part (we may consider this a header) and a variable part. In the fixed part we encode 3 things : the *filter ID*, the *flowgroup ID* and the *flowspace navigator*. The *filter ID* is used when instantiating a grabber to associate it with the right filter. The *flowgroup ID* is used to make clear what belongs to which flowgroup. The last fixed item is the *flowspace navigator*, explained as follows. Because there can be more grabbers with the same filter in the same flowgroup, but in a different place in the graph, the ID must uniquely distinguish them. As more applications can share a part of the flow graph, the IDs that refer to the same node must be exactly the same. Since the only part of *FFPF* which has knowledge about the entire graph structure at the time of `populate` and `instantiate` is the front-end (*libffpf*), the final ID must be created here. To build IDs which are created by distinct applications but express the same position in the graph, we use a similar approach as with the prefixes. The subgraph on which a particular node is dependent is encoded in the variable part of the node's ID structure. Note however, that the meaning of the variable part can change in the future.

Now let's return to the so-called *(flowspace) navigator*. We have encoded the flowspace position in a single 32bit value. This value is virtually divided into 8 4bit fields. This limits the depth of flowspace-hierarchy to 8 with a branching factor of 14, as two values are reserved. One is used to prohibit descending from the current level to lower ones and allows the populate call to succeed only in the localspace on the current level. The other permits descending to all branches. We consider this to be a reasonable limitation, since having too many processing levels would increase latency in the FFPF system, which may not be desired in certain circumstances. Note that the number of devices (which increases branching) is also limited by the hardware and whether the host machine can handle the received traffic by all these NICs. Throughput of the PCI bus is one of the bottlenecks.

The *navigator* is assigned to a grabber ID in two different ways. When populating the bottom-most node (device) we cannot foretell in which flowspace the device-grabber will be populated. Therefore a special *navigator* has to be assigned that permits to descent into all branches. Once a filterclass is found and the backtracking to the userspace starts, all flowspaces on the path put their mark into the navigator. The mark is given to the flowspace during initialization, the level ("depth") of a flowspace is hardwired. As mentioned before, the *navigator* is divided into 8 fields, each for a mark from one level. A *navigator* can be presented as a hexadecimal number where each digit is for one field.

When populating a dependent filter, we take *navigators* from all its direct predecessors and calculate the longest common path, starting in the userspace. The first field where the *navigators* differ is marked by one of the reserved values which stops further propagating of the populate call and enables to succeed only in localspace. If there is only one predecessor, the navigator sent to populate is identical. But it can change as the population returns from higher level. The returned value is prefix of the predecessor's.

As a flowgrabber is not yet created by the `populate` but later in the `instantiate` phase, the ID also navigates this call to the right place (Fig. 3.2b). From then onwards all other flowspace-methods find the grabber even if not directed.

A few examples :

`ffffffff` allows a flowgrabber to be populated wherever

`00000000` limits flowgrabber population only to the userspace

`10000000` allows a flowgrabber to be populated in the first flowspace in the list kept in userspace (i.e., starting from the userspace, it may be instantiated in the first flowspace bellow it. It is usually kernelspace), but not deeper

`13000000` allows a flowgrabber to be populated in the third flowspace in the list kept in kernelspace (e.g., ixp2)

3.3 Flowgrabbers vs. Ghost-grabbers

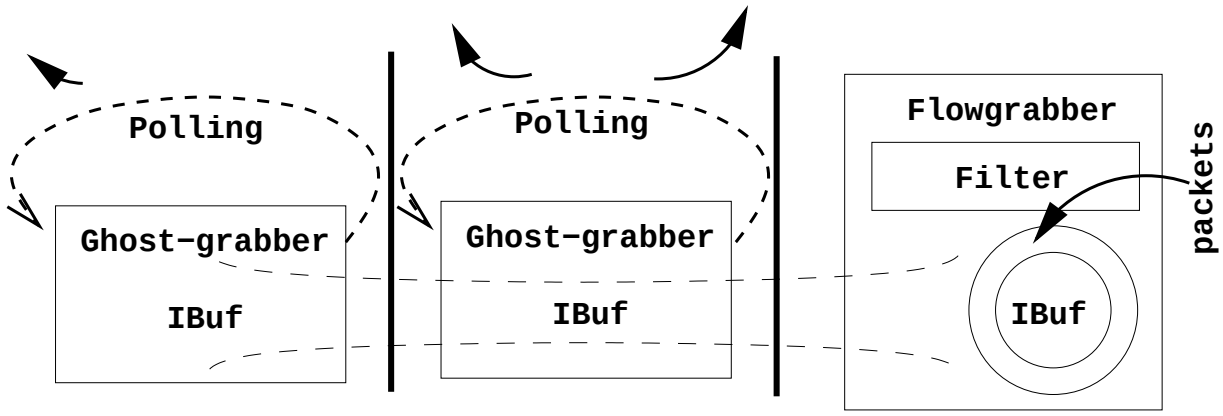


Figure 3.3: Mapping a *flowgrabber* to its *ghosts*

As discussed in Chap. 2 we use two categories of flowgrabbers: real-grabbers and ghost-grabbers. In this section we describe them in detail. The *real* grabber is always associated with a filter (Fig. 3.3) and does the real filtering. In contrast, a ghost grabber is the real-grabber's exported image. Its only role is to keep the information about the mapped buffers. The dashed loop depicts registration with some kind of reading mechanism (usually polling) which grabs data from mapped buffers and participates in making the graph connections. The difference between real and ghost grabbers is an internal issue of FFPF and transparent to the user.

Unlike a real-grabber, a ghost is never an intermediate node in the part of a flowgraph on some level in the flowspace hierarchy. It has only forward connections, i.e., connections that are used to pass packets to next grabbers for further processing. Backward connections are managed on the lower level where its real representation lives. The interconnection between these two incarnations is done in the sense of reading data from the mapped buffers. When a packet is read from a buffer, a handling function checks all connections from the ghost and passes the packet to each successor, one by one.

An example is shown in Fig. 3.4. The original flowgraph is in Fig. 3.4a) whereas the populated result is shown in Fig. 3.4b). Two nodes representing ethernet devices are exported to userspace. Connection to the ghosts only assures that all data received by eth0 or eth1 will be delivered to all dependent nodes. Whether these ghosts represents NICs

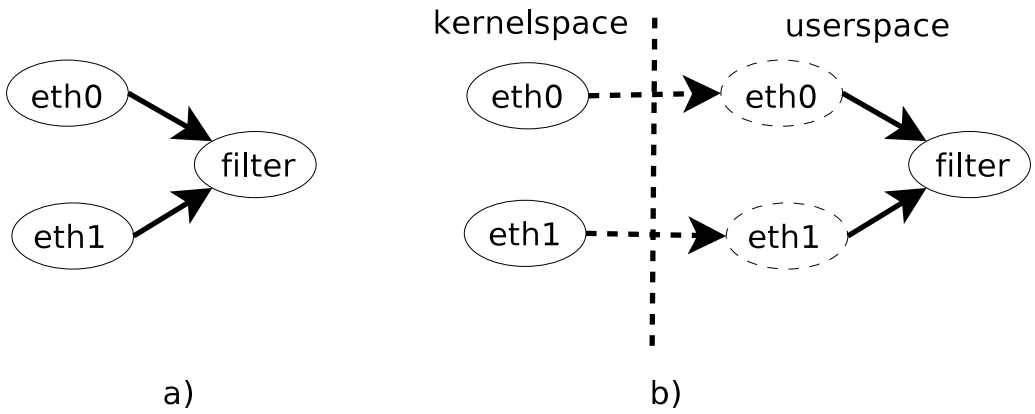


Figure 3.4: Device ghost grabbers

or leaves of a more complex graph which is populated in kernelspace (or even deeper) is not relevant, as both structures are treated in the same manner.

3.4 Crossing the PCI bus

All interaction with the ENP-2611 card is done across the PCI bus. We can directly access DRAM from the host and, vice versa, the IXP2400 processor on the card can access host memory. Moreover, the IXP2400 control-status registers are also available in the PCI memory area. But what exactly do the CPUs on both sides see? They see the Intel 21555 PCI-to-PCI bridge (PPB). No communication may cross from one side to the other without going through this bridge.

3.4.1 The Intel 21555 PCI-to-PCI bridge

The Intel 21555 [12] is a PCI device that performs PCI bridging functions (connecting two PCI buses). It has 64bit primary (host) and secondary (ixp card) interfaces with 66MHz capability. It is a non-transparent PPB. Any local processor can independently configure and control local subsystems. Unlike a transparent PPB, the 21555 interconnects two processor domains. It enables :

- Independent primary and secondary PCI clocks
- Independent primary and secondary address spaces
- Address translation between primary and secondary domains

The i21555 forwards transactions just like a transparent PPB, but it also performs address translation. It enables one to hide subsystem resources from the host processor and resolves any conflicts that may exist between the host and the local subsystem. Besides the address translation we also use the ability to interrupt the host processor when a special (doorbell) register is written. No other interrupt is forwarded to the other side of this non-transparent PPB.

3.4.2 Control protocol

As control messages are sent from the userspace to the kernel by `ioctl` calls on a special `/dev/ffpf/control` device, they must also be sent across the PCI to the IXP card. For this reason we implemented a very simple protocol. We use a buffer of fixed size (by default 1MB) allocated in IXP DRAM to exchange data. In our protocol, all communication is initiated by the host. The IXP is only replying to requests.

Whenever data are written in the exchange buffer, a signal must be sent to other side. For this purpose we use the doorbell interrupts. When we ring the doorbell an interrupt handler on the other side receives the message. When sending a signal from the host to the IXP, we can use either the doorbell register on the i21555 PPB or on the XScale. We opted for the latter one. This register is mapped in the PCI memory area. Once this register is written with a non-zero value, the XScale core receives an interrupt. This register is 32bits wide. A written value does not only raise an interrupt, but it can also be read and this way distinguishes what kind of message is pending. An IXP reply is signaled by writing into the 16bit doorbell register on the secondary side of the i21555 PPB.

When exchanging a small amount of data, we use the *mailbox* registers on the XScale. There are four of these registers, all of them 32bits wide. We use this e.g. in the data-pushing negotiation.

3.4.3 Mapping IXP memory to host

Our ENP-2611 card has 256MB of DRAM. Actually when initialized it configures only a 64MB window in the PCI area which is visible to the host. This window can be mapped to any contiguous area in DRAM. The mapping is done by setting up a window size (at boot time), which determines how many bits of an address are used as an offset in this window. The i21555 adds this offset to the value in `translated_base` register. The results is an address in DRAM.

Since this is the only part of the DRAM that is visible to the host, all buffers must be allocated here. 1MB is dedicated to the control data exchange buffer. Into the first megabyte of the same PCI window IXP2400 control-status registers (CSR) are mapped. This leaves 62MB that can be used for buffer allocation. Unlike on the host, there is only one packet buffer. It is written by ME0 and read by other filters. The rest of the memory is available for index buffers and extra memory buffers (*MBuf*). This is further divided into two pieces : one for micro-engines and one for filters running on the XScale core (see the layout in Fig 3.5. Both parts are managed by different memory allocators implemented in FFPF.

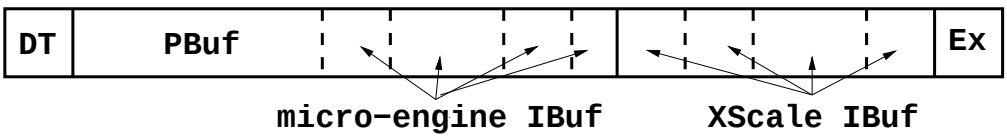


Figure 3.5: IXP DRAM layout

3.4.4 Mapping the host memory to IXP

The IXP can also directly access the host memory. This enables faster data transfer via the PCI bus into the host memory using DMA. This is explained in detail in sections 3.5.3 and 3.5.4. The host CPU is not involved in this and data are copied once, regardless of how many times they are used later on. There is an argument that it is not necessary to copy the entire packet every time e.g., when only a header is going to be used. However, we have opted for copying the entire packet for the following reasons :

1. It is expected that there are filters running on the host (moreover in userspace) which will need the entire packet for complex inspection (like pattern matching)
2. Depending on the PCI width (32 or 64 bits) and frequency (33 or 66MHz), the bandwidth is sufficient for maximum data transfer between 1Gbps and 4.2Gbps, while the host CPU is not involved in that and the XScale only partially.
3. There is always the possibility to make data-pushing more complex to reflect the graph structure and filters in such a graph and make decisions based on the expected traffic, etc. Then it would be possible to copy only the parts which will be used and in doing so save some PCI bandwidth. The current filterimplementation does not provide any of the necessary information.

Mapping the host memory is more tricky. Rather than a single `translated_base` register, there is a *translation table*. The window is divided into 64 equal-size pages. Each of these pages can be mapped to one memory chunk on the host. From the secondary side's point of view, it forms one contiguous memory area which is not necessarily contiguous in the host RAM (see Fig. 3.6). In our case this window is also 64MB and so every page has 1MB size. Unfortunately, there is no simple way to allocate bigger chunks of physically contiguous memory (e.g., in the order of megabytes) in the Linux kernel. There is only `kmalloc` which allocates memory that is contiguous in virtual as well as in physical address space. But it can allocate chunks only in small multiples of `PAGE_SIZE`. As we want to allow the IXP to push data to the host, we need to reserve some memory, which Linux would exclude from its memory management and we can map this memory via the PCI. The Linux kernel accepts boot time parameters. One of them (`mem=`) can limit the amount of memory to be used out of the total amount of memory that is physically present. This leaves all the memory above that limit for our use. Filling the *translation table* to point to this area is then trivial. Moreover there is no obstacle to use one contiguous memory area anymore.

3.5 Mapping buffers

Since we receive packets at the bottom of the flowspace hierarchy, but process them in all levels, we need to access them. There are several ways to do this.

The first option is to explicitly copy every packet to the higher level space. This is a straightforward approach. After a packet is processed by a grabber which has no more dependencies in the current space, the packet is copied to the next level. This would require a receiver on the other side which would pass the packet to the dependent filters.

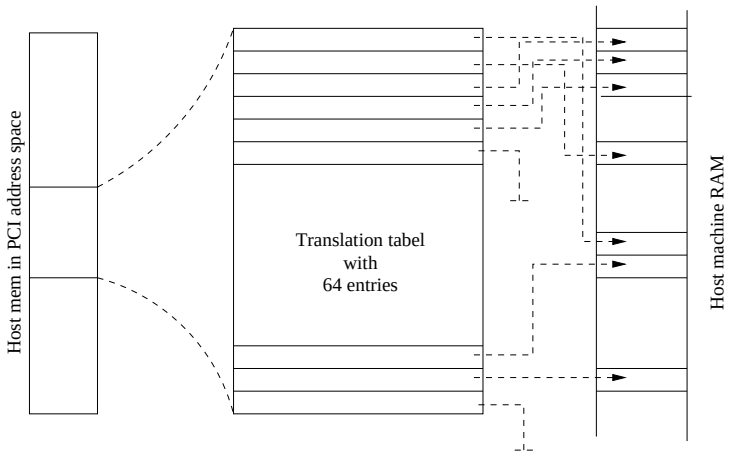


Figure 3.6: Translation of contiguous PCI "push" window into non-contiguous host memory

The main problem is that it could possibly lead to multiple copies of such a packet. This is not desired for reasons of performance.

The second option is to map memory (buffers) between spaces. This gives us more freedom and flexibility. The upper space grabs packets from these buffers on its own, using polling together with signalling. Polling is a good choice for high traffic loads. Most of the time, when a reader tries to get the next packet, it is already available. Signalling would lead to a big overhead, since the operating system spends too much time in signal processing. On the other hand, signalling from time to time is not bad and enables sleeping between polls. The system can use its power for other processes instead of constantly checking empty buffers. Whenever a new packet arrives, a signal is generated and polling can do its work. Possible improvement is an adaptive polling which adjusts time between consecutive polls. More about our implementation of polling is described in Sec. 3.7

3.5.1 Mapping from kernelspace to userspace

The easiest way to enable applications in userspace to read kernel or device memory is to create a pseudo character device and use the standard memory mapping offered by Linux. When exporting a grabber the `/dev/ffpf/dataN` [3] device is created. Later

[3] e.g., data5, data20, etc.

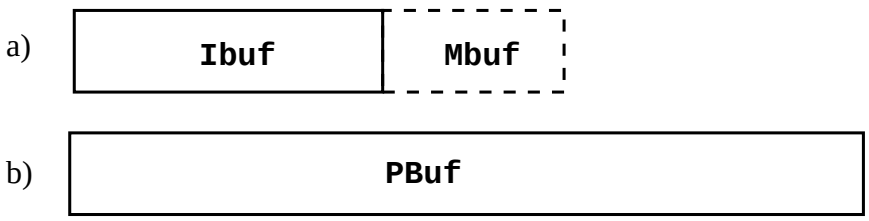


Figure 3.7: a) /dev/ffpf/dataN and b) /dev/ffpf/pbufN layout

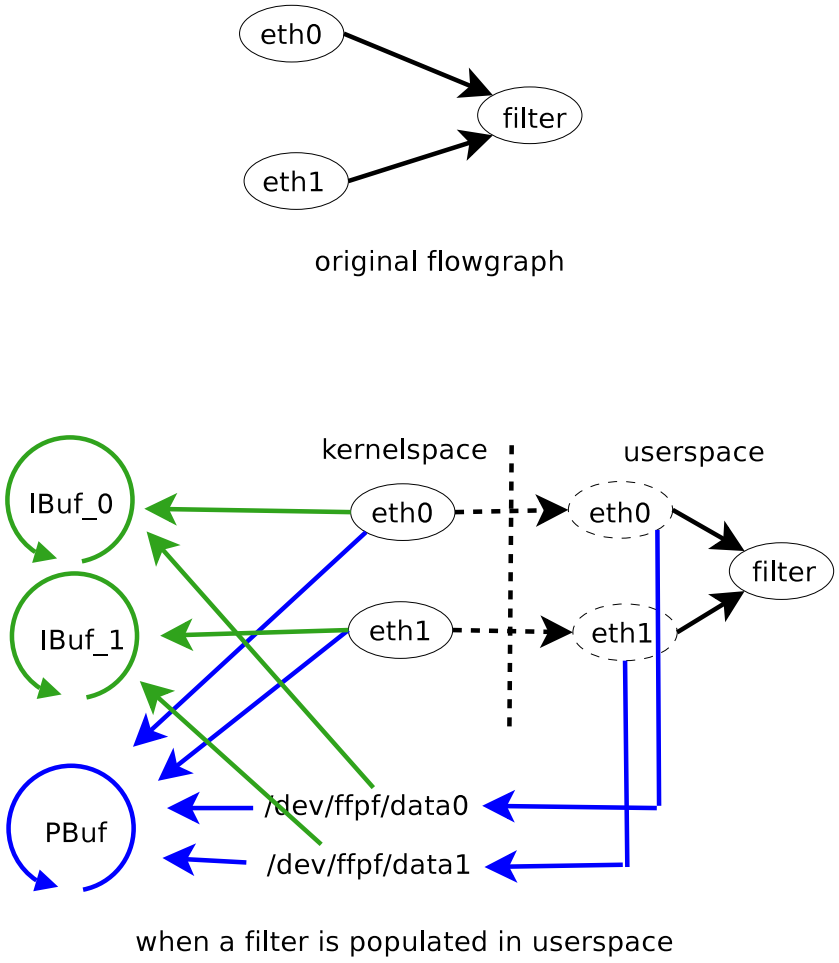


Figure 3.8: Multiple mmapping of one kernel packet buffer

on, when the grabber is mapped, the kernel sends the size of every buffer to userspace. The `export` routine in the kernelspace stub then `mmap`s areas in device files as buffers associated with the ghost-grabber. A pseudo data device is partitioned (see Fig. 3.7) so that the kernel can recognize which buffer is requested and map it correctly. *MBuf* is only optional, not every filter will use it. If we try to `mmap` it even if there is none associated with the grabber, it leads to an error.

PBufs are mapped as separate devices `/dev/ffpf/pbufN`. This was a necessary step when we decoupled packet buffers from flowgrabbers. Once an index is read from an *IBuf* one needs to get data from different *PBufs*. This was not the case before and the packet buffers were mapped to the so-called `dataN` files. Usually one buffer is referenced by more than one grabber and therefore mapped to many device files. We have now an extra device for *each* packet buffer. Also every packet buffer is mapped just once into each instance of *libffpf*. This simplifies many things. One major problem it helped to solve was that a mapped buffer from the kernelspace should not look different when referenced by a different exported (ghost) grabbers. In this case the packet buffer

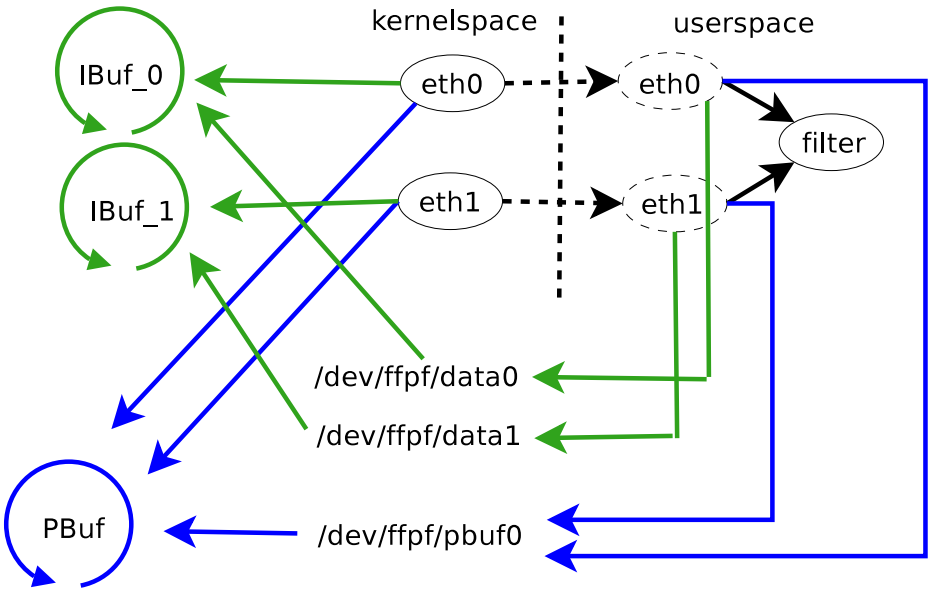


Figure 3.9: New mmaping of kernel structures into userland

was mapped into multiple `dataN` pseudo devices and therefore the same buffer was represented by more base pointers as illustrated in Fig. 3.8 which depicts the original situation for a simple example. Two ethernet devices are exported to userland since they are used by a "filter" populated in *libffpf*. The devices share one packet buffer. There was no way to tell if two pointers represents the same packet buffer or not. In contrast Fig. 3.9, where each ghost grabber has its own `dataN` file, but both points to the same mmapping of `pbuf0`, presents the current situation.

From the kernel's point of view, there are two distinct memory mappings. The first and easier one is to map real device memory. This memory is usually contiguous and all that we have to do is to call `ioremap_page_range` which does all the work. A new association between the physical memory and the page frames in the VMM [13] subsystem is then established. From that moment onwards, whenever userspace process accesses mapped area, it accesses the device memory directly. This is used for buffers which resides on the IXP cards.

Another, slightly more complicated procedure is required for buffers which are allocated in the kernel by `vmalloc`. This is a non-contiguous memory and whenever a page in this area is accessed that was not yet associated with the virtual space of the current process, the `nopage` method is called to handle this page miss. (More details on memory mapping in [14]).

3.5.2 Mapping between IXP and host

Mapping memory from the IXP to the host is much simpler than mapping between kernel and userspace. IXP DRAM is accessible in the PCI memory space. When *ixpspace* is initialized it requests a memory range which represents the IXP DRAM and uses

`io_remap_nocache` to map it to the kernel virtual space. This operation gives us a base pointer. All we need to get from the IXP to `mmap` any buffer living in IXP DRAM is an offset from the beginning of the visible area. Exactly this information is returned by the *map* call to *ixpspace*. It can be mapped then to userspace as explained in the previous section.

We use the `io_remap_nocache` to make reads consistent. It may happen that the XScale or an ME writes something in area, which is in our local cache and is not invalidated.

This access to the IXP memory was used already in the IXP1200 implementation. Except that the *map* call was not implemented. Instead, the available range was partitioned in fixed chunks and every chunk was assigned to one filter.

In our new approach, where part of the FFPF logic is on the IXP card, we introduced data pushing from the IXP card to the host. In certain circumstances the data-pushing policy can be negotiated. In previous versions whenever packets in buffer were referenced, a PCI bus transaction was initiated. When some data were used too often, there was much traffic over the PCI bus. In such cases it is better to copy the data once to a buffer in the host memory. It leads to the COPY-ONCE or the ZERO-COPY policies. The latter policy means, that packets are always in the DRAM on the IXP card. The first one means that there is an instant, when the entire packet is copied to another location (via the PCI) and all following processing is referencing the copy. The copying was initiated by the host and was performed by the host CPU. Data pushing by the IXP card, on the other hand, bypasses the host CPU as the IXP XScale processor copies the data to buffers in the host memory. Mapping such a buffer to the userspace is no more difficult than mapping any other chunk of kernel or device memory.

In the previous version, the copy-policy was determined when loading the FFPF module. In the current implementation, we have to specify where the reserved memory is for the data-pushing as FFPF module parameters. The default setting assumes that there is no memory reserved. Whether packets are copied or not is decided by the control-part of FFPF in the runtime, as described in the following section. How the IXP copies data to the host memory is explained in Sec. 3.5.4.

3.5.3 Data pushing negotiation

In the previous implementation there was no *map* method since there was only exporting from the kernel to userspace and mapping was done during this phase. This is no longer sufficient. We do not need only to postpone memory mapping until the whole graph is built, but also to enable a more complex mapping process like the data-pushing negotiation.

When the *xscale* receives a *map* request, it can start the pushing negotiation (Fig. 3.10). Unlike in the other calls, here the initiator is the IXP card. This option is available only for "output" filters. An "output" filter is a filter, that does not have any filter that are dependent on it on the IXP. Only the IXP card can detect this. Since it is exported (otherwise *map* would not be issued) FFPF knows that packets accepted by this filter will be used on the host by other filters.

After the host receives a request for pushing, it has to allocate a buffer in the memory

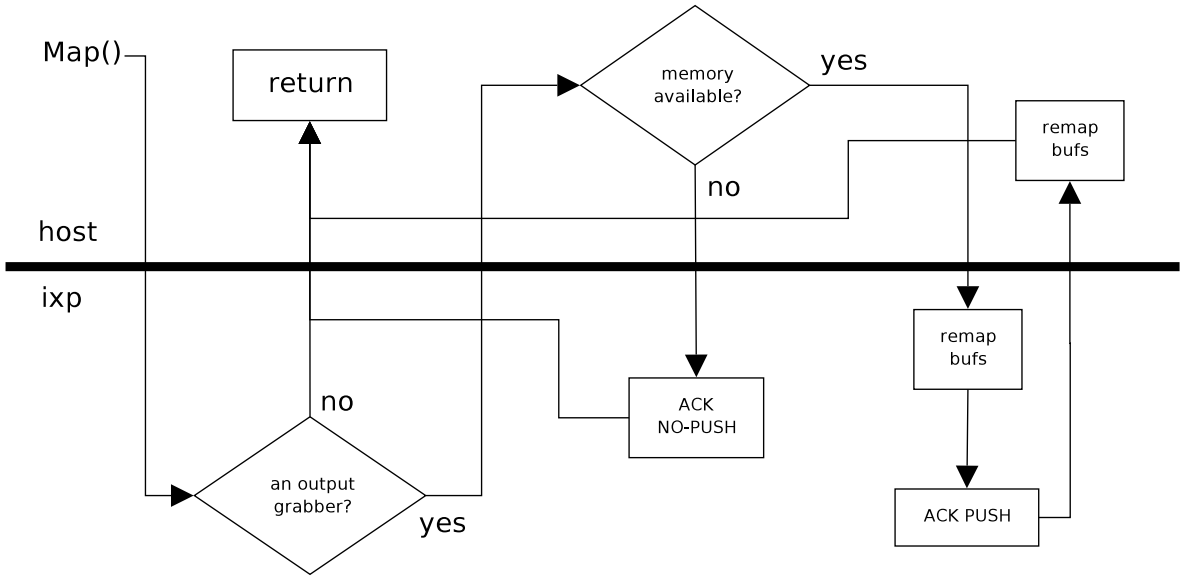


Figure 3.10: The data-pushing negotiation

area to which the IXP can write. If there is no such area or there is no more free space, the request fails. However, this is not an error and FFPF simply uses the buffer in IXP DRAM as if there is no pushing at all. We decided that there is only one packet buffer for data-pushing as there is one packet buffer in DRAM. This allows us to detect very simply whether a packet was already saved or not. As a result every packet is pushed only once across the PCI bus and only the index is saved into the *IBuf* for every filter. In the current implementation the host returns the same packet buffer when ever it is asked.

3.5.4 DMA

Using DMA for transferring data to the host memory is the major improvement in the FFPF data-path. Leaving packets in the IXP DRAM when they are processed on the host needs a lot of reads via the PCI bus performed by the host CPU. This reduces the performance in the case that a lot of packets have to be processed on the host. It would be better if these packets are already in host memory. In the previous version copying a packet to host memory was implemented in the host part of FFPF. Results presented in [10] show that this is suboptimal and it leads to many drops.

The IXP2xxx chips provides us with a DMA controller. We can use this to transfer data from the DRAM to the PCI address space which is mapped to host memory, as explained above. The host CPU is not involved in this process and it is an asynchronous data write for the XScale. We do not use all the features which are offered by the DMA controller but in comparison to the data writes via the PCI bus performed by the XScale itself, the performance is up to $4\times$ higher. The FFPF design itself limits the use of the DMA. The idea is to copy a packet only once whereas it might be accepted by more than one "output" filter. But this means that an index must be written to more then one *IBuf*. We copy a packet immediately as it was accepted by the first "output" filter. This trans-

fer is done completely by the DMA controller and the XScale continues with a further processing of this packet in another branch of the flowgraph. The indices written to the *IBuf* are very small structures which, in contrast to packets, are not stored in the DRAM yet. Therefore we write these structures by the XScale synchronously. We also have to synchronize with the DMA before we increment the write indices to make sure that data transfer is finished before we allow the host to poll on the newly written buffer slot. After the DMA operation is finished, the XScale is interrupted. We mask this interrupt since it would reduce the performance. Instead we poll on a DMA controller register where a bit is set when the transfer is finished. This synchronization is marginal in the case that the transfer is finished before processing of the packet is done.

In comparison to DMA performed by an ordinary NIC, FFPF on the host can benefit more. First of all, the host CPU does not dynamically allocate any memory for incoming packets, the host CPU does not have to communicate with the card to specify the DMA transfer destination and moreover, all the data are written directly to FFPF buffers.

3.5.5 Mapping of microengine-space

The DRAM memory is accessible by both the micro-engines and the XScale core. The XScale core is allocating memory for the ME filters and the grabbers from the *micro-engine space* are implicitly exported to the XScale kernel space. We can directly use pointers for this.

3.6 Many PBufs in a flowgroup

As already mentioned several times, this is one of the major improvements in the original framework. Here we describe its impact on the internals of FFPF, on the user and performance.

A look at a complex flowgraph in one flowgroup (See Fig. 3.11), shows that some of the nodes receive packets from different sources. This is highlighted by different arrows. Dashed arrows are for not-yet-saved packets, all others are for packets already stored in buffers with the same label. The icon of each buffer is placed in the flowspace, where it is created. One exception is *PBUf* B. Here we see that flowgrabber `f1` in the XScale kernel is an "output filter", as described in the Sec. 3.5.3. As such it is a candidate for data-pushing and we see that *PBuf* B indeed was created by `f1`, but as its location was negotiated between IXP space and host kernel space, it was placed in host RAM. We should mention that a grabber does not only receive data of different solid arrows, but it also passes them further. An example is the grabber `f5`. This flowgrabber is interesting for one important thing. It changes the packets arriving along the dashed arrows to solid ones. What does it mean? These packets were not stored yet, since they are placed into FFPF by the ethernet device from *netfilter*. But `f5` is connected to node `f6` which is populated in userspace. This results in its exportation into userland as well. As already mentioned on various place before, all packets used in userspace from lower levels must have been saved in some packet buffer which can be mmapped. Therefore `f5` must be associated with a new *PBuf*. It can either reuse *PBuf* C (if `f5` is in the same flowgroup as `f2`) or create a new one (D). The choice is made by FFPF. This does not change the fact

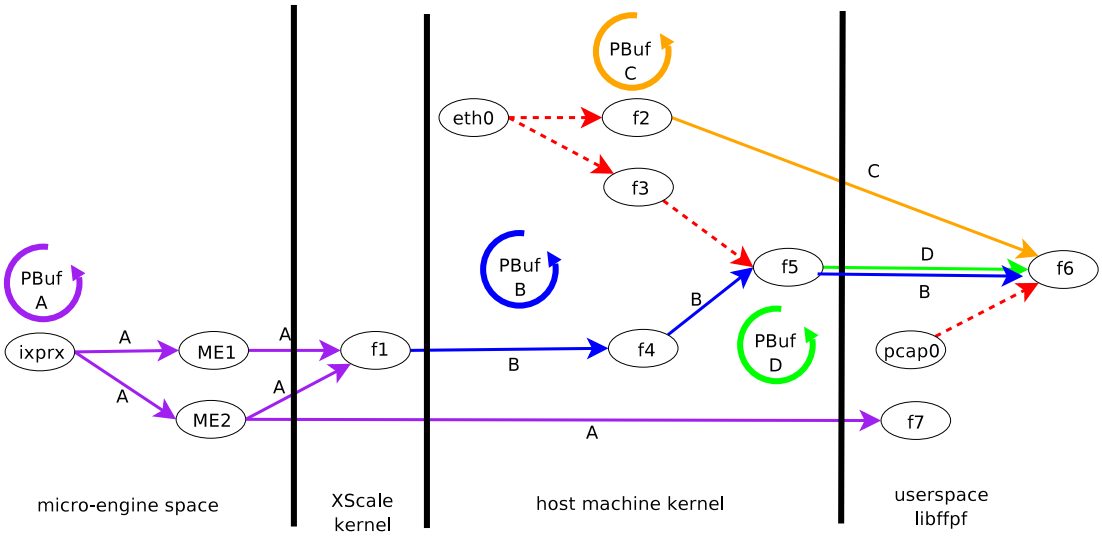


Figure 3.11: Complex data flow with merging of different data origin

that packets classified by `f5` as accepted are from that point on in physically different areas. We cannot do this unless we have more packet buffers inside one flowgroup and a node can be associated with more than one buffer as well.

3.6.1 Grabbers and buffers

First of all we have to mention here, that not every grabber has buffers. Many "inner" nodes just process data and pass them to the next "hops" without needing to save any data. They just classify. Buffers are useful only for nodes on the border of flowspaces. As every boundary has two sides, grabbers and buffers on each side have different roles.

First, consider the flowgrabbers that are exported to the upper level. Here, the grabber needs a *PBuf* and an *IBuf* to store data. It acquires a slot in the packer buffer and then it stores data to this slot while a reference is stored in the index buffer. If the packet was already marked as stored, only the reference is saved.

Next consider the ghost-grabbers, i.e., the exported flowgrabbers. They need buffers for the opposite task - for receiving data. *PBuf* and *IBuf* of the real-grabber are mapped into the upper level and the ghost keeps this information. A grabber can be registered either with a polling system or another method can be used to get data.

Storing data in the new implementation is virtually unchanged. The main differences can be found in ghosts, buffer assignment and propagation.

3.6.2 The old way

In the original design, there was at most one packet buffer associated with a grabber and indexes retrieved from the index buffer were pointing into this *Pbuf* (see Fig. 2.5). Only the index itself was not enough to locate a packet. It carried no information about the data storage, only position and classification. Index retrieved from an *IBuf* associated with a

grabber was pointing to the single packet buffer associated with the same flowgrabber. That was to restrictive and problems and limits were already discussed before.

3.6.3 The new way

In the new design we made these major changes :

Index structures

After retrieving an index from *IBuf* (of which there is only one for every grabber) we have to identify the storage where the real data resides. We cannot use any kind of pointers stored along with the index (offset) in the index buffer. The reason for this is simple. Pointers are not valid in different flowspaces and the identification must be global. After all, there can be a filter running on IXP micro-engine which writes data into buffers that are read in userspace on the host machine. Therefore a *Pbuf* ID was added to the index structure.

PBuf identifiers

From the previous paragraph it is clear that we need to identify packet buffers from the information stored in the index buffer. And this identification must be fast since it may be done for every single packet in the data flow. Therefore we decided that for such an ID it must be possible to use it as an index into a translation table from where we can get a pointer that is valid for this level in the flowspace hierarchy. Another constraint for these IDs is that they must be valid in different endianness environments. We don't have time to check it and maybe convert the byte order. This would represent a serious slowdown in packet processing. And as observed previously the system performance is crucial.

The result of these constraints is that we use a 32bit identifier field in the *Pbuf* header which is copied into every index pointing to this buffer. 32bit were chosen to make all structures aligned in memory but this is not consistent because of endianness[4]. For example, the IXP card uses big endianness whereas an i386 host uses small endianness. Therefore the real ID is only 8 bit wide and placed into all 4 bytes of the identifier field. Once comparing the same byte (e.g., the least significant) we get a valid value everywhere.

These identifiers must be unique on all levels. If this is not fulfilled we cannot map IDs to pointers. We dedicated (by convention) the upper 4 bits to a "flowspace" identifier and the lower 4 bits to a sequence number. This limits the number of buffers in *each* flowspace to 16. We are aware of this limitation but we do not consider it a big drawback. In theory, as there are flowspaces like ixpspace where is only one packet buffer and we don't need all 16 possible IDs we could share this range among all cards in our system and use only 3 or 2 bits for "flowspace identifier" and more bits for sequence number. If the user (administrator) still feels limited, he can decide at compile time to change the translation function between ID and pointer to a hash routine which slows down the processing a little but allows to have many buffers on one level. Even so, we do not

[4] different byte ordering of 16, 32 and 64 bit numbers on different architectures

recommend this because using many buffers also means a lot of data copying and storing which is slow in itself.

ID to pointer

Once we get the index together with a buffer identifier, we need to get the pointer to the real buffer. This was not necessary in the previous version since the pointer was kept in the flowgrabber structure. Now, there is one level of indirection. We take the ID and look up the desired pointer in a translation table. Even though we do not expect that a grabber will use too many sources, we extended its structure by adding an array of 256 items which are directly addressable by the 8 bit ID. This translation function is implemented as a C-preprocessor macro and can be substituted by another function (e.g., hash) if the FFPF user so desires.

Building the translation table

When a grabber is exported it sends its description table to the new ghost-grabber. This table is build recursively at the moment, the real-grabber has to store data because of the exportation from its flowspace (Fig. 3.12). Once FFPF wants to export a grabber (e.g., F6 in Fig. 3.12), it gathers the translation tables from all predecessors and merge them into a final table. The information about buffers that are known to the grabber (reflecting information about sources of packets which can possibly arrive into this node) is propagated along all connections.

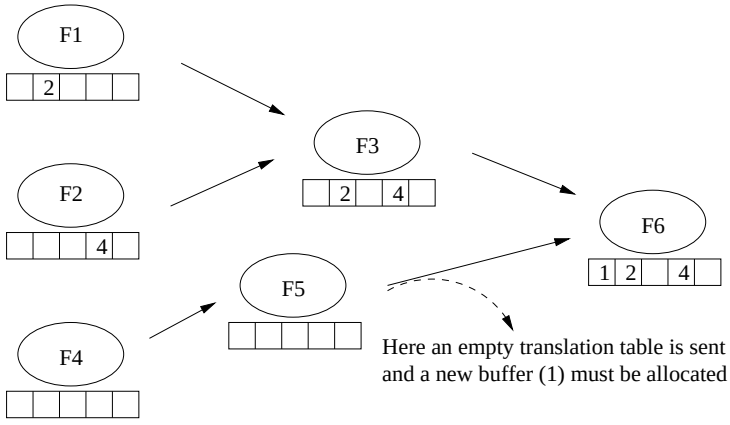


Figure 3.12: Recursive translation table merging and unsaved data detection

At this point it is important to mention that here we can find out whether some packets might arrive unsaved, by detecting that some direct predecessor (e.g., F5 in Fig. 3.12) returns an empty description table. In this case we can characterize the behavior of FFPF as "*packet buffer on demand*". This node needs to store the unsaved packets into some place. It asks the buffer management subsystem for storage. If there is space which can be reused, it is returned. If not, a new one is created. In both cases its descriptor is placed into the translation table and all successors will know about this buffer once they are exported as well.

As the pointers stored in the translation table of a grabber that is being exported would not be valid in the upper level, these are not sent. So how to complete the creation of a "ghost" table? The answer is that we have to query an extra translation table which is global for the entire flowspace level as described in the following section.

The flowspace global translation table

The global translation table in userspace is also built *on demand*. When exporting a flowgrabber, all *PBuf* IDs are copied to the ghost (Fig. 3.13a). In the mapping phase, we have to complete the ghost-grabber's translation table. First, we query the global translation table (Fig. 3.13b). If there is a pointer associated with this ID, it is returned (Fig. 3.13g). If this ID is not present in the global table, it must be added and the appropriate *PBuf* have to be mapped to userspace. First, *libffpf* asks the kernel which `/dev/ffpf/pbufN` device to use to map this buffer and what is its size (Fig. 3.13c and Fig. 3.13d). If the same buffer is used by multiple applications, the device is shared. This query looks up a global table which is maintained in the kernel. This table is filled in when the IXP buffers are mapped or a kernelspace buffer is created. We need `pbufN` device name and the size to call `mmap` (Fig. 3.13e) which returns the packet buffer base pointer. It is filled in the global translation table (Fig. 3.13f) and in the ghost-grabber's local table (Fig. 3.13g). This processes is repeated until the ghost-grabber's table is complete.

This is a strong decoupling of packet buffers and flowgrabbers in kernel and userspace. There is just a single *PBuf* on the IXP card so use the original simpler way of mapping. Therefore the translation overhead is not present. But in kernel and userspace we have to merge data received by different IXPs and other devices.

Introducing this global table helps to simplify two tasks. First of all it is easier to count references to each packet buffer, separately on every level. When destroying a flowgrabber we call `put` on each item in the local table and it decreases the reference numbers. When this counter drops to zero the destructor is invoked. The destructor is a special function associated with the global table and differs for each flowspace. A flowgrabber can simply forget about its packet buffers and the destructor takes care about correct deallocation or unmapping itself. The second improvement lies in the single mapping of every buffer. In the previous implementation it was mapped as many times as there were ghost-grabbers referencing this particular buffer. There was no benefit from this and moreover it used too many `vmalloc` resources inside the kernel.

3.7 Polling

One of the crucial parts of the data-path in FFPF is polling. It is responsible for quick retrieving of packets from buffers and passing them to the grabbers for processing. Polling is used on all the flowspace borders. Polling in userspace is based on the standard way of polling on a character device in the Linux system. More interesting is our new implementation of polling in the kernel, on the host and on the XScale.

The previous version used a very simple way of pure polling on the IXP1200 buffers. There was a timer which initiated reading from buffers every time it expired. The amount of data read at-once was limited by the number of items present in the buffer when the

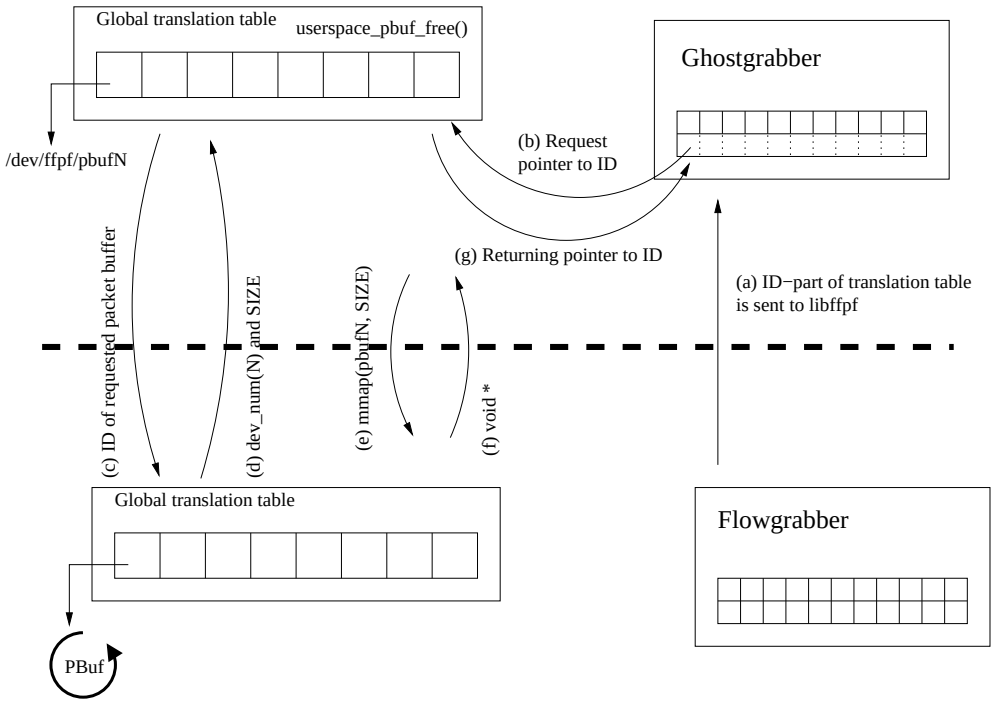


Figure 3.13: Exporting a packet buffer from kernel to userspace

polling started. Polling is done on the *IBufs* and there is more than one in most of the cases. Therefore a constant polling on one buffer until there are no more data would lead to starvation of grabbers dependent on the other *IBufs* than the one currently processed. By giving up after a limited number of retrieved packets it gives a chance to the others to be read too. After all buffers were checked, the timer was set again and nothing was done even if there were new data already stored in the buffers before the timer expired again. This is an obvious drawback, resulting increase of latency and a performance leak. The polling subsystem is redesigned in the current version as follows.

The basic idea is already used in the NAPI [15], the Network API for Linux drivers and all driver implementors should use this API instead of the old one, which is still supported because of backward compatibility. The old network drivers were interrupt driven. After data were available, an interrupt was raised and a software handler was woken up. This approach was enough for low speeds. As the bitrates are increasing, operating system would spend most of the time in the interrupt handlers and not in the data processing. This leads to the NAPI approach of combining the interrupt driven driver with polling. The idea is that when data arrival is signaled by an interrupt, the IRQ is masked (i.e. the same IRQ cannot be raised during data processing) and after all packets are finished, the driver checks if there are new data in a buffer. If there are none, it enables the IRQ again and waits for a signal.On the other hand if data are present, they are processed immediately. The result is that with an increasing load, less interrupts are received. If the traffic is so high that new data are always available, it turns to a pure polling. This has the ad-

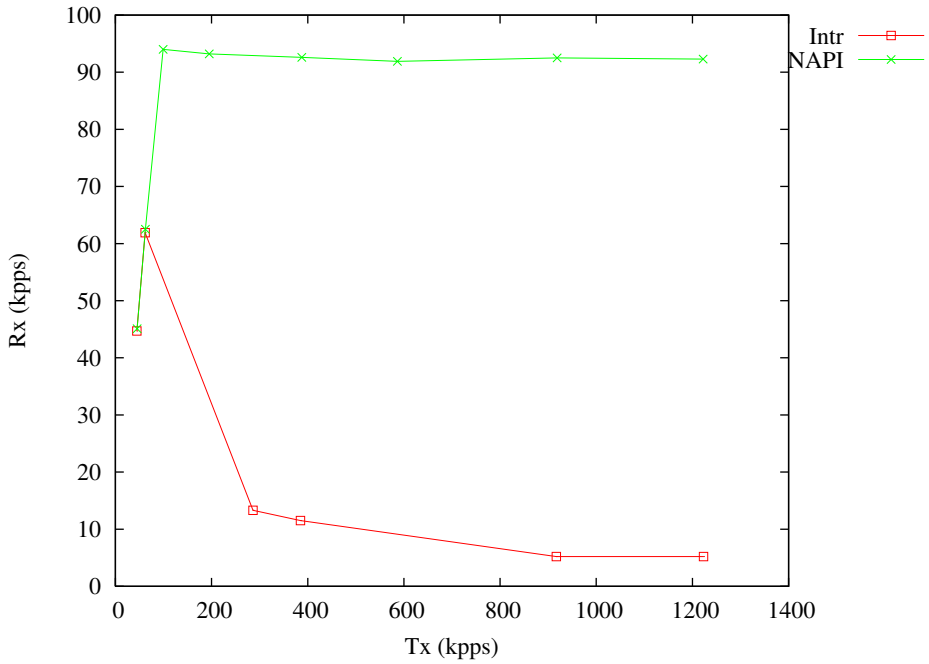


Figure 3.14: Comparison of a NAPI and a non-NAPI driver in the Linux

vantage that it can process approximately constant number of packets, when overloaded. This is not the case in the older drivers, as presented in the Fig. 3.14. We used the FFPF for packet counting in kernel to get these statistics. Packets were received directly from the *netfilter*.

As we do not have an ordinary NIC, we cannot reuse the NAPI, but we incorporated the idea into the FFPF. There is a special polling thread which is woken up by the coming interrupts. Most of the code is shared by the XScale and the host. There are only two differences : receiving interrupts and scheduling.

On the XScale, an interrupt is sent by an ME when a new entry in an *IBuf* is written. Signaling every packet does not affect performance. According to the benchmarks we have done, on average it costs less then 2 ME cycles to send an IRQ to the XScale. Once an IRQ is received by XScale, other interrupts from MEs are disabled and are not delivered during the processing. Every interrupt on the XScale is forwarded to the host via the i2155 bridge. The host polling behaves in the same way, only the code of the interrupt handler is different since it runs on different hardware.

The polling thread on the XScale has the maximal real-time priority in the system. That means, if there are data available, only this process is scheduled. And is the first to run after an interrupt is received. As it is a kernel-thread, it uses 100% of the CPU time. This thread never yields the CPU voluntarily as all the processing is done in its context. Only threads with the same priority can compete. Therefore the controlling thread runs with the same priority and can suspend the poller while reconfiguration of the FFPF structure is in progress.

In contrast, the polling thread on the host has the priority of a normal process. The

reason is to let also other processes to run. We do not want to overload the system, because it must be able to respond to the user's requests and moreover, the userspace part of FFPF, *libffpf*, must be able to run as well. Otherwise all results of the packet processing would be lost.

This kind of polling has the benefit of a reduced latency. In the case of the previous timer-based implementation, packets stored in a buffer just after the previous processing is finished and the timer was set again, were processed no sooner then after the timer expiration. The polling is aware of them immediately after the first one is signaled.

3.8 IXP flowspace

As we stated before, our flowspace hierarchy has a strict tree-structure. On every level, only the local space can be seen and stubs which represent the levels below. So the host kernel sees only one object which represents all of FFPF running on the IXP card. We decided to call it simply *IXPspace* and describe it in one section even though it is actually divided into two parts, *IXPkspace* (in kernel on XScale) and *uespace* (on micro-engines). As some of the important issues were already covered, here we want to present the remaining interesting aspects of our implementation. In Fig. 3.15 we illustrate the main concepts discussed in the next few subsections.

3.8.1 IXP space - XScale kernel

There are two active FFPF components in the kernel : local space (which differs only slightly from local space in the host-kernel) and kernel communication thread (for exchanging control messages). Let us concentrate on the receiver at first. As we mentioned already we use the doorbell interrupt to signal a control message arrival, e.g. sent by host over the PCI bus. Handling interrupts in the Linux kernel is done in three common ways. The handler itself is supposed to be a very short and fast routine. This is to prevent loosing the next interrupts from a device. Moreover there is only a small amount of interrupt lines on some architectures (e.g., x86) and these lines are shared by multiple devices. Spending too much time in one handler means loosing messages for the other devices. Only a few handlers do all the work which must be done in the handler itself. A more common way is to clear the interrupt and leave the actual processing for some better occasion. What time is better is decided by the Linux kernel itself. How to postpone an interrupt processing? A *bottom-half* or *tasklet* can be registered. Both options are quite similar, unfortunately none of them suits us well. This is because the (interrupt) handler is not running in any process context and so it is not possible to sleep/wait in such a handler. This requirement is because of the communication with the micro-engine manager (uEM) (See Sec. 3.8.2). On the other hand we do not need an extremely fast response to the request. It is the control part of FFPF and does not affect the runtime performance. Considering all these reasons and limits we decided to split our receiver into a kernel-thread which acts as a *bottom-half* and an interrupt handler which wakes up our waiting thread.

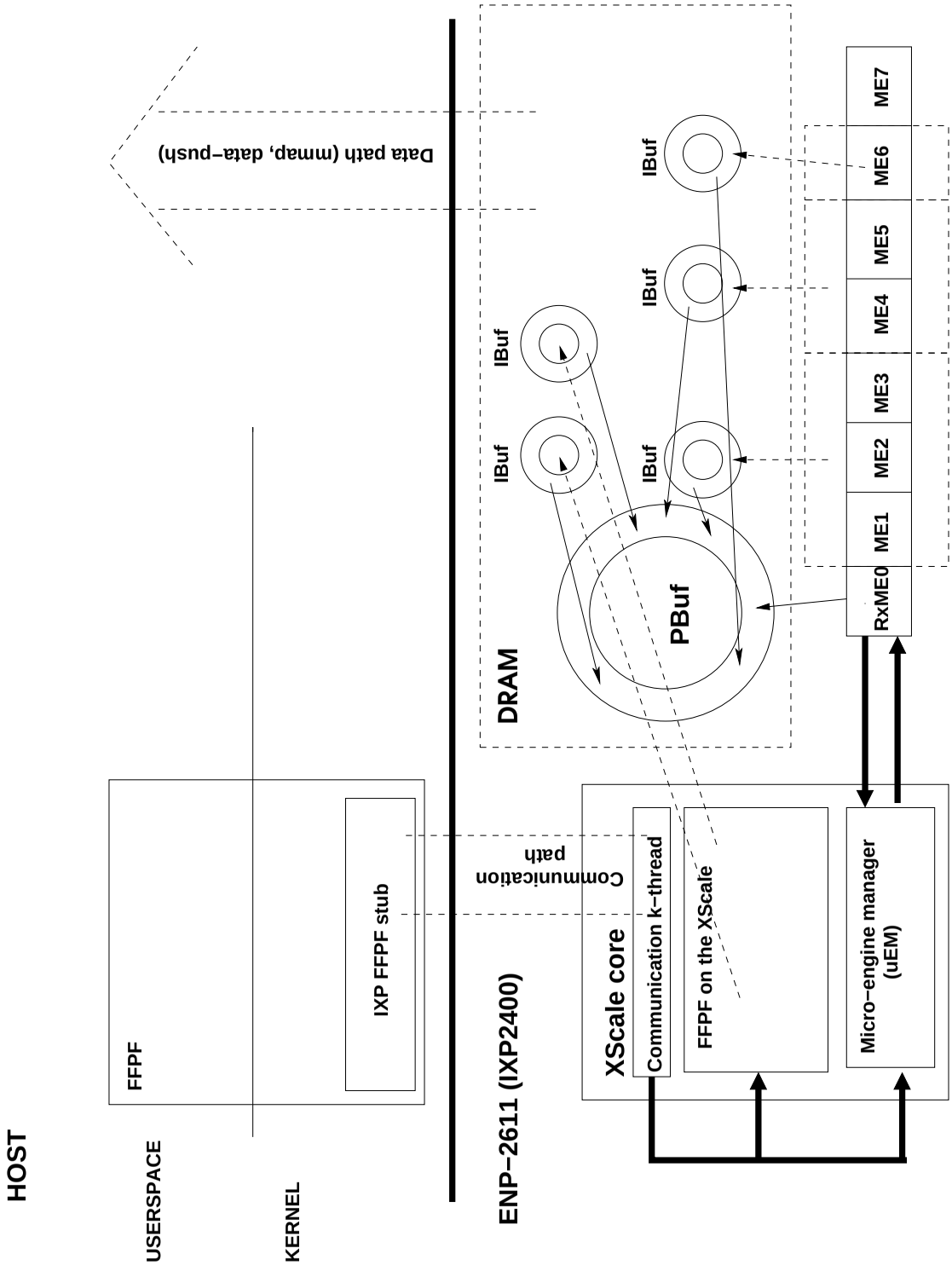


Figure 3.15: The IXP space overview

The kernel space on the XScale core does basically the same as any other flowspace. The only difference is that it implements the *map* method which can negotiate with is host-stub the possibility of data-pushing. For details, see Sec. 3.5.3 where it is discussed.

3.8.2 IXP space - micro engines

Micro-engine space is the more interesting part of the IXPspace. From a design point of view, it is wise to split FFPF on the IXP card into two parts. Not all IXP devices have an XScale core and for that reason, we are not able to run Linux on such hardware. Nevertheless we do not want to exclude such devices from the use of FFPF. The micro-engine space logic can be easily moved to the host machine or to another machine which is controlling the device. The only thing to be changed is how the uEM controls the MEs.

As any other flowspace, uespace has its stub in the XScale kernel which communicates with the heart of uespace, the uEM. uEM is a userspace process (discussed later) so we have to enable the kernel to communicate with it. In the current version of Linux the simplest method to make data structures available to userspace is *sysfs*(`/sys`) and *kobjects* [14] KObjects are a new device driver model which made its appearance in the 2.5.45 kernel. It does not only increase the "object oriented" way of programming in the kernel for device driver implementors (reference counting, grouping of similar objects into sets, etc.), but in connection with *sysfs* it gives a more comfortable way of kernel-userland communication. In contrast to the *procfs* every kobject is represented by one directory in `/sys` and every single value of the object is a file inside that folder. A userspace process can communicate in the sense of reading and writing such files. As this filesystem is virtual, a programmer can implement his own read/write operations. We implemented four such files :

- *request* - Reading this file is a blocking operation. If there is no request pending, the process is put into a wait queue and is woken up by our receiver thread when a request arrives. The request is then copied to userspace. The only limit here is the size of such a request. It cannot exceed `PAGE_SIZE` Moreover, if needed we can use different class of files, used for uploading binary data into devices, usually firmware.
- *reply* - In contrast to the *request* file, it is write-only and all written data are sent as a reply to the stub in the kernel.
- *error* - The same as the reply, but it signals that some error occurred.
- *ueconfig* - R/W file used for the description table, where the uEM stores information used by the ME initialization code.

Micro-engine manager

The uEM is the core of the micro-engine space (uespace). It is responsible for the following tasks :

- Managing available filterclasses - previous to their use, all classes must be registered with the uEM. Registration tell how much memory a filter requires, now many

micro-engines, how many contexts (threads), how to handle the *populate* and *instantiate* requests, etc.

- Managing micro-engines - uEM allocates the required number of micro-engines, loads the code to the micro-engine instruction store, starts and stops the execution of this code.
- Managing memory - uEM implements a memory allocator which partitions the available memory as requested by instantiating various filters.
- Preparing the "in memory" structures for micro-engine code.

There is only a limited number of MEs on each IXP chip. The IXP2400 has 8 of them, the IXP28xx has 16. uEM is responsible for allocating MEs for each filter. If there are no more free engines, the *populate* call fails and FFPF tries to find a corresponding filterclass in another flowspace, if there is a non-ME implementation available. We do not limit a filter to use just one ME. IXP2xxx series provides a set of registers which are shared between neighboring MEs. These can be used for fast communication between two MEs. Therefore we enable a filter implementation which requires more than one ME. These engines must be in a contiguous block. If such a block is not possible to allocate, populate fails with the same consequences as described before.

This brings us to another important issue : interaction with the micro-code. Once a filter is uploaded before it is started, it needs information about where the buffers are. We want dynamically allocated buffers[5], dependent on actual memory size of the IXP device and on the needs of the filters. Filters can require different amounts of extra memory. Either in the form of *MBufs* which are then exported to the user, or in the form of memory which they need for correct execution. The amount of local memory to each ME is very limited and larger structures must be allocated in SRAM or DRAM. All memory except the small local store is shared with the other MEs and the XScale. Therefore every ME has to get its own piece of memory not to interfere with the other processors. Moreover, a filter may run on a different ME every time it is loaded and thus use a different memory area. Any hard-coded memory areas in the ME code are therefore illegal.

For these reasons we keep a *description table* from where a filter can read where the memory is which was allocated for it. This table is on the fixed address `0x0` in DRAM. The first megabyte is not reachable by the host machine and therefore it is not suitable for buffers allocation which may be exported. It is up to the filter programmer to read it in an initialization code before the body of the filter is executed. The description table has one entry with the address of the global *PBuf* and entries with the location of an *IBuf* and a *MBuf* for each ME.

The micro-engine code

The micro-engine code is produced by the Intel compiler, included in the SDK. A filter can be implemented either in the micro-C [16] or in the IXP assembler. Both are compiled to `.list` files, one for each ME. Further, these files are linked to one `.uof` file. This file includes code for multiple MEs which can by loaded to the instruction store of each ME as a single IXP application. This is done by a kernel module, which is included in

[5] dynamically allocated in the load time, NOT runtime

the SDK. This module is compatible with Monta Vista Linux kernel 2.4 and we had to port it to the current 2.6 Linux kernel. This module performs load-time relocations of the `.uof` code and initialization of ME registers. This limits our uEM in loading filters to the MEs. We cannot use multiple `.uof` file, therefore code for all MEs must be prepared in one file. The ME number to which a filter is loaded is encoded in the `.uof`.

In case the programmer writes a filter which runs only on a single ME and the compiled code does not need any relocations while loaded, it is possible to get the instruction from the `.list` file and load it directly into the instruction store. This is usually the case of code written in the IXP assembler. Loading ME code in this way is not supported by the Intel SDK. This method is very inconvenient for the filter writer and not many of the users we target will opt for that. More likely they will use the FPL language which is more abstract and hides the hardware details. Code written in the FPL is also compiled into a `.uof` file. Therefore the uEM was designed in such a way that it can handle `.uof` file correctly, using the Intel proprietary SDK libraries, but it can be easily changed to work with any other kind of sources.

3.8.3 Endianness

One of the problems we were facing are different byte orders on different platforms. We already came up to that problem in Sec 3.6.1 In the previous version, the FFPF was completely running on one platform, so there was no problem. Only the code handling data coming from the IXP1200 had to deal with a possible difference since the IXP chips are using big endianness by default. It is configurable, but as the big endianness are native for network, we keep that setting. The IXP1200 buffers were hidden to the rest of FFPF, so this code could have been optimized and supply already converted values to the filters.

We made all the IXP buffers public for the entire framework. Every level can access data already stored in a memory without copying them into a different location. The problem is that buffer headers and also buffer item headers are written on the IXP, in the opposite byte order than the one used by x86 host. We provide a set of macros which returns values in the correct byte order used by the actual CPU. This slows down the packet processing. On the other hand, it is used only in userspace and partly in the host kernel. The code running on the XScale always uses native order and the polling subsystem on the host deals always with big endianness and no conversion is needed.

3.8.4 Packet receiving on the IXP

On the IXP only the micro-engines can receive packets. Therefore micro-engine ME0 (the first one) is used only for receiving data from the gigaports. It writes them to *PBuf* and *IBuf* and sends an interrupt to the XScale. In spite of the limit that writes to the packet buffer must be synchronized (the writeindex can be increased iff all slots were filled, i.e. there are no gaps between the read indexes and the writeindex), one microengine can save 64B packets on the full speed of a gigabit link.

Chapter 4

Performance of the extended FFPF

The purpose of our testing and benchmarking was to measure limits of different flowspaces in FFPF. We were also testing how fast the inter-flowspace communication is and how much one space influences performance of the others. To find out how many packets can reach each part we used a simple packet counting filter. This filter classifies all the packets as accepted and for every packet it increments its counter. This counter is a 4 byte number in an *MBuf*. It can be read by an application after being exported. Our goal was not to evaluate the speed of a particular filter and its implementation but the maximum throughput of the data path.

4.1 Fast-reader preference simulation

In our framework we use the so-called *fast-reader preference* on the IXP card. The writer does not wait until the data in the buffers are processed by all readers, but writes at the full speed of the incoming traffic. Therefore the reader has to check whether the data just retrieved from a buffer are still valid. After processing it must be checked once again. In the case that they are not valid anymore, they are dropped. We opted for such a policy because it does not require a writer synchronization with readers and no fast reader ever has to wait for a slow one (as in tail drop). This is simple if there is only one reader. Our framework allows multiple readers which can be added and removed dynamically during processing.

We ran simulations which showed that this policy does not lead to excessive drops. Under a reasonable load, its behavior is similar to the tail-drop, which is mostly used. Packets dropped after being processed are the last ones before the whole buffer is skipped. The number of such packets is not significant. The difference appears when the system is so overloaded that only a few packets are processed before the whole buffer is overwritten. In contrast with the tail-drop policy, fast reader preference collapses in the extreme situation when the writer rewrites the whole buffer before a single packet was processed. That means that the reader process thousand times less packets then the writer, which is not the expected case. If that happens, it is up to the user of FFPF to copy data to another place or handle such a problem in another specific way.

Our testing included one writer and one reader with adjustable speed, so we were able to test different ratios of writing and reading. As we target the high speed traffic, we

used a writer with a constant time between writes. It corresponds to ethernet behavior under maximal load. In the model, processing time was also constant. We included delays which the poller spends in the main loop or sleeping in a wait-queue. It turned out that the dropping is constant if we can process at least a single packet before it is overwritten. In the reality, there are not only minimal size packets, but the traffic changes its characteristics. This will give an average reader more time as the intervals between writes vary and not all the packets pass trough all filters and therefore the processing time is shorter than the constant in our extreme simulated case.

4.2 XScale core performance

Due to the general design of FFPF, there is an inherent overhead as the processing part does not get the packet handle directly. The poller is reading data from an *IBuf* to get an index into the packet buffer. Getting an index involves testing whether it is still valid, or already overwritten. Another test is done after retrieving the packet, before it is passed to the filtering graph. We finally excluded this test as the simulation and also the real runs showed, that only a few packets are dropped here, but it costs us many cycles. Breakdown of the FFPF overhead for the 64b packets on the XScale is presented in Tab 4.1. The right-hand column shows how many packets could be process if only the actions in the left column were performed.

Retrieving an index from IBuf	450kpps
+ retrieving the packet from PBuf	357kpps
+ processing the packet	305kpps
+ saving index to exported IBuf	215kpps

Table 4.1: Overhead of actions taken during a 64B packet processing on XScale

To test the maximum number of packets which the poller is able to get from the receiver on the first ME, we turned off the packet processing as well as the getting hold of a packet from the *PBuf*. The XScale turned out to be less powerful than we expected. As this can not be further optimized in the current FFPF design, we consider this a hardware limit for FFPF. There are two reasons for this limit. First of all, the XScale core is running on 600MHz which limits the number of operations per second. Another important reason is the memory bus bottle-neck. The memory is heavily written by the micro-engine receiver as it stores whole packets into the *PBuf* as well as the indices into the *IBuf*. One item in the index buffer has the size of 24 bytes. As an index is saved with every packet, this can introduce an overhead of up to 33% of the received data size for minimum size packets (64 bytes of packet plus 8 bytes of each *PBuf* slot header). This increases the memory bus usage. The XScale has to wait until its memory operations are performed. Caching on the XScale does not help since with every new packet we reference a different memory location. Moreover indices written to DRAM are to be read by the host and caching memory on the XScale would be inconsistent with the host.

The XScale, the micro-engines and the PCI bus can access memory simultaneously. All the read/write requests are sent to the DDR DRAM memory controller and the con-

troller performs the operations. Processed requests are reordered because of the performance and are not executed in the original order. Ordering between read and write operations is preserved. There is a performance penalty associated with switching read and write requests. The scheduling algorithm attempts to schedule requests of the same type in succession so that the switching between read and writes is minimized. The memory bus is 64bit wide and therefore all requests need to be rounded to multiple of 8 bytes and aligned to 8 bytes offset. Therefore each write request smaller then 8 bytes results in a read-modify-write sequence. The micro-engines can also transfer only multiples of 8 bytes. Therefore all the structures that are shared between the XScale and MEs are aligned to that boundary. Nevertheless, the XScale performs 32bit memory operations, individual writes results in the previously mentioned read-modify-write operations, which probably have a negative effect on the XScale performance.

After enabling the full processing with the `packetcount` filter and a reader on the host, the maximum dropped down to approximately 215kpps. In the Tab. 4.2 are the values measured with flows of equally sized packets of various sizes. For this experiment we used only a packet counter on the XScale. The small difference in the numbers might be caused by the different memory bus usage pattern by MEs and the limited granularity of our packet generator. Three of the flows did not reach their maximum as the packets were too large and therefore then packet number was low.

As a packet generator we used Intel IXP1200 evaluation board. One gigabit port of the IXP1200 was directly connected to one of the IXP2400 gigabit gigabit ports. One micro-engine was generating up to 750Mbps. We were able to generate UDP packets of various sizes between 64 bytes to 1520 bytes and with various payload. Actually we have not inspected the payload as our interest was only in how many packets we are able to pass between processing stages.

64b	**128**	**256b**	**512b**	**1024b**	**1520b**
214.5	229.5	210.5	—	—	—

Table 4.2: The maximum number (in kpps) of received packets on the XScale for different sizes

For testing and debugging we changed names of basic filter clases in such a manner that there were different names for each flowspace. For instance, the names for the XScale core were prefixed by `ixp_`, names for the host kernel were prefixed by `host_` and names for the userspace were unchanged. That allowed us to changed easily places where the filter graph was populated just by changing the input expression.

Fig. 4.1 presents results measured for packet counting on the XScale. The flowgraph expression used in this test was :

```
(ixp_rx)>(ixp_packetcount,mbufsize=4,export=y)
```

Exporting the `packetcount` to userspace on the host (to enable reading the counter in the *MBuf*) means, that with every packet an index is saved into the memory mapped *IBuf*. This buffer was placed during this test into the DRAM on the IXP card. This introduces another additional traffic on the shared memory bus. The size of the packets was not interesting in this experiment as the content was not inspected. Still, we present the measured values in Mbps instead of packets per second to make them easily comparable.

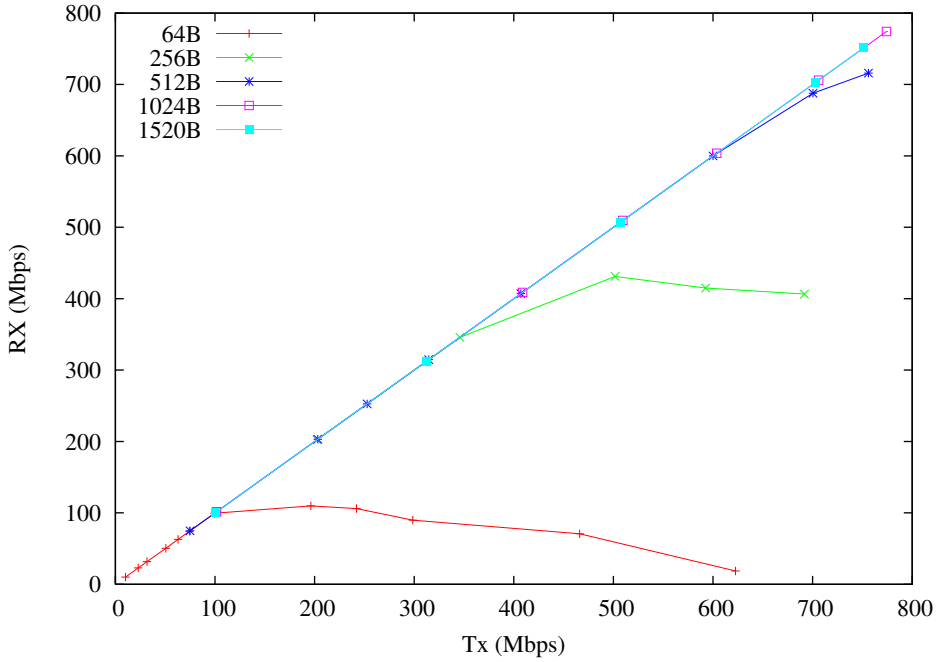


Figure 4.1: Performance of XScale, using 'packetcount' and saving indices

These results show two important things. First of all, the poller works as expected and after reaching its maximum, it approximately keeps that level. It is slightly dropping because of the increasing number of dropped packets. The other observation is that the XScale has the potential of hosting a simple filter. Assuming that the average packet size is not too small or enough packets are filtered out by the ME filters. This exactly fits into the idea of our hierarchical architecture. Measurements of the XScale performance and the data-pushing is presented later it Sec. 4.3.2.

4.3 Host performance

We divided performance evaluation of the host in two parts as there are two options from where the host can read the data. In the first scenario, the host part of FFPF was reading data stored in the IXP card DRAM. In the latter, the XScale copied all the data directly to host memory.

4.3.1 Reading from the IXP DRAM via the PCI bus

In this experiment all buffers are located in the DRAM on the IXP card. Therefore all reads on the host are done via the PCI bus. Tab. 4.3 shows the limits of how many packets can be referenced via the PCI bus (64bit/66MHz). The numbers are not significantly smaller than on the XScale, but it shows that the PCI bus is a bottle-neck as the host CPU (P3 1266MHz) is much faster than the XScale core. We do not copy much data over the PCI, therefore the maximum bandwidth is not limiting. Latency together with

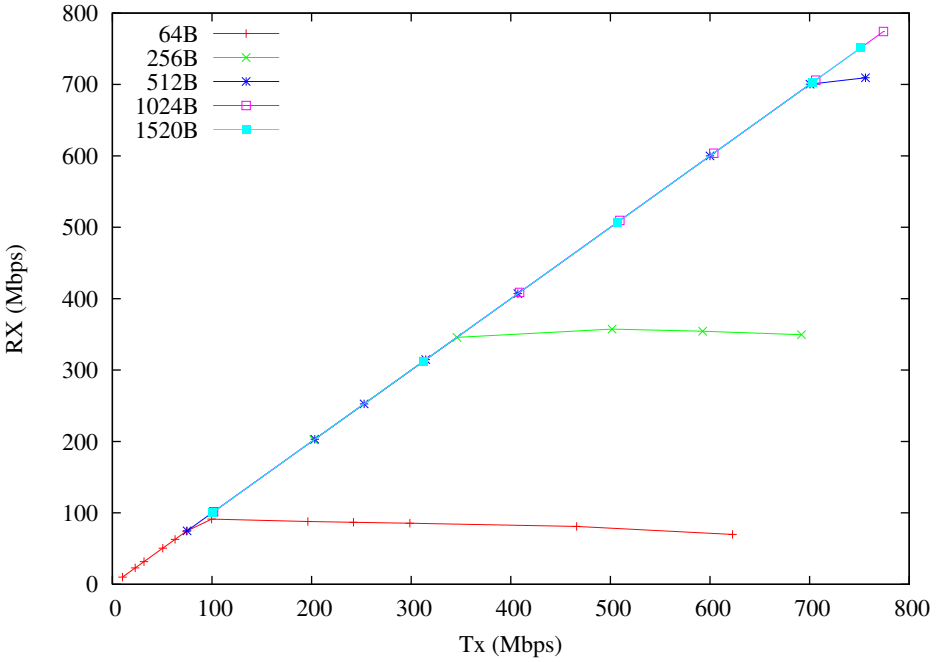


Figure 4.2: Performance of the host CPU, reading from DRAM via PCI, running 'packetcount' and saving indices

an additional load of the DRAM cause the lower performance. It takes much time to get values from the DRAM to calculate a packet position in a buffer. Many of them must be dropped. Another annoying observation is that once the poller reaches its maximum, the host machine stops to respond as the poller thread uses nearly 100% of the CPU cycles, this is a livelock. Therefore the userspace part of FFPF gets few or none packets. In Fig. 4.2 the behavior of the poller is presented. We did not reach the maximum number for packets larger then 1kB. The amount of packets which were successfully processed in the userspace was dropping with an increasing amount of time spent in the kernel polling and processing. The flowgraph expression was :

```
(ixp_rx)>(host_packetcount,mbufsize=4,export=y)
```

64b	256b	512b	1024b	1520b
178.2	174.5	(173.2)	—	—

Table 4.3: The maximum number (in kpps) of received packets on the host via the PCI

4.3.2 Reading from local memory

Reading data from local memory gave us an expected result. The host machine (P3 1.2GHz) was fully responsive (70% idle) and there is no packet drop between kernel and userspace.

The following issues have to be mentioned :

- There is no dropping between the IXP and kernel-space on the host. The XScale does not receive another packet until the data are copied over the PCI to the host memory. As the host CPU is faster than the XScale, it can handle all of them.
- Significant packet loss on the XScale is caused by synchronizing with the DMA engine before the write indices can be updated

The limiting factor of the data-pushing is the XScale processor together with the *IBuf* overhead. Results measured with the 64bit/66MHz PCI bus are shown in Fig. 4.3. The XScale itself limits the maximum number of packets which can be transfered to the other side of the PCI bus. We can copy only as much data as we can process on the XScale. We use a DMA engine which is provided by the IXP2xxx NPU. This allows us to send packets asynchronously and it hides some memory/PCI latency. It is, naturally, more significant for large packets then for the small ones. Still, some items in buffer headers must be accessed/changed by reading/writing via the PCI bus. Finally, we have to check that the DMA transfer was finished before we increment write indices. To hide some more latency we signal data to the host prior to incrementing these indices. This means that the sleeping poller-thread should be already woken up by the time the index is increased. Waking up a thread is not an immediate action. It depends on the scheduler and the current host activity when the poller continues its execution. If the poller was running, the signal is not necessary and is not delivered at all.

Fig. 4.3 presents data measured for the following filtergraph expression :

```
(ixp_rx)>(ixp_packetcount,mbufsize=4,export=y)>
>(host_packetcount,mbufsize=4,export=y)
```

The counter is maintained in the host memory. Each update results in a read and a write via the PCI bus. This slows down the packet processing and therefore also the amount of packets which can be passed via the PCI bus as the DMA is not issued before the filter finish. For comparison, this figure includes also the same measurement with a trivial `accept` filter which classifies all packets as accepted. Role of this filter is to negotiate the data-pushing policy. This filter does not involve XScale in any processing. The difference is significant. The results were improved 1.42× for 64 bytes packets.

```
(ixp_rx)>(ixp_accept)>(host_packetcount,mbufsize=4,export=y)
```

4.3.3 Copy-once vs. zero-copy

Our experiments show that reading data over the PCI bus is slow. Too much reading of buffer headers slow down the packet delivery to filters. After optimizing this the performance was improved by the factor of 2 when reading via the PCI bus. If the filters need a repeated random access to a different part of a packet, it will reduce the overall performance. In that case it is much better to copy the packet once to the host memory. We have implemented a simple filter `host_sum` which accepts all packets but also calculates the sum of the first 6 dwords (32bits). Only counting packets can handle approximately

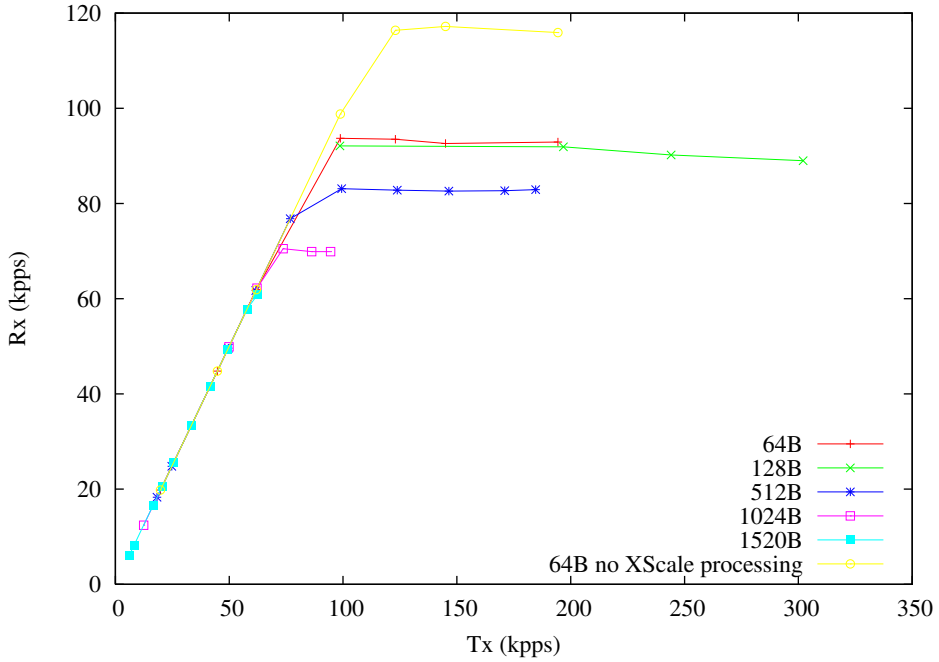


Figure 4.3: Characteristic of the data-push method (64bit/66MHz PCI)

170kpps, whereas one such a filter (`(ixp_rx)>(host_sum,export=y)`) which accesses a constant size of the packets itself reduces the number to 83kpps and two such filters connected in series (`((ixp_rx)>(host_sum))>(host_sum,export=y)`) reduce the number to 54kpps. The same two filters together with the DMA-like data pushing from the card can process 132kpps.

```
(ixp_rx)>(ixp_accept)>(host_sum))>(host_sum)>
>(host_packetcount,export=y)>(packetcount,export=y)
```

In this case it means that such a simple packet processing does not have any impact on the performance, in contrast with the "via PCI" access. More over the host CPU was approximately 50% idle in the copy-once case.

Therefore we think that the right direction of future work is in further exploiting the DMA-like data-pushing. It does not only improve the usage of the host CPU, which can spend more cycles on the processing, but it also gives the host more time for dealing with each packet. If the packet stays in the DRAM *PBuf*, it will be overwritten very soon. The assumption is that the higher the processing goes in the flowspace hierarchy the less packets need to be inspected. Therefore the utilization of the buffers in the host memory is lower and overwriting a whole buffer is less frequent.

4.4 Comparison of a normal NIC and the ENP2611

To compare how many packets can be processed on the host we have chosen an *Intel PRO/1000XF Server Adapter*. It is difficult to find a fair comparison as FFPF handles

packets in a different way in both cases. Packet received from the ENP2611 are always stored in a packet buffer and are never copied inside FFPF. On the other hand, packets received from ordinary ethernet NIC have to be saved to a ring buffer before entering an application in the userspace.

We have to admit that the Intel card can supply FFPF with 2.3× more packets of the smallest size than the IXP. As already described, this limitation is because of the FFPF implementation on the XScale. This is a price we pay for the extensibility. On the other hand these packets should be already processed by either micro-engines or the XScale itself. Viewed from the FFPF point the use of an IXP card is much better in the case when the user is interested in many packets which have to be read by the application. As mentioned before, all the packets must be saved in a ring buffer. Copying a packet to such a buffer is very costly operation which overloads the CPU very easily. This results in a huge dropping (82%) already in kernelspace and moreover nearly no packets are processed in userspace.

On the other hand, FFPF on top of the IXP card can use DMA to transfer data directly to buffers in the host memory. Therefore all the host CPU power can be used for retrieving references to packets and processing them. There are no drops between kernelspace and userspace. Naturally, this is also dependent on what part of the filter graph is populated on the host. If the filtering overloads the CPU, it will start dropping packets as well.

4.5 Micro-engine filters

The crucial part of the FFPF monitoring system on the IXP network processors are filters running on the micro-engines. MEs have the potential to keep up with the full speed of the incoming traffic. Therefore the vast majority of the filtering which does not need excessively expensive analyzis should be implemented as micro-code. This will give the XScale and the host an opportunity of processing the rest without dropping. ME filters were out of the scope of extending the FFPF except for their management. Management of this code (loading, starting, stopping, memory allocation, etc.) does not influence the processing results. Performance of the ME filters is discussed in the following sections.

Chapter 5

An IXP as a special purpose device

Previous chapters describe how an IXP device can be used as a set of execution units that can atomic filters, which form the resulting monitoring application. In contrast, this chapter discusses how to integrate an IXP card as a device that is executing a single task. From the user's point of view it is a blackbox that might consist of many simpler interconnected units or not. The only significant thing is that it acts on incoming traffic according to given rules and results are passed to the rest of the system by a unified interface. This chapter discusses what the main benefits are of using an IXP in such a way and presents Ruler, a pattern matching language, as a sample implementation.

In contrast with the previous chapters, which are mostly avoiding the IXP microengines, this one presents code that is mostly running on top of them. Sec. 2.1 briefly presented general IXP features. To understand the rest of this chapter it is necessary to show more details on the MEs. Each ME is capable of executing up to 8 simultaneous hardware threads that share the same code of up to 4k (IXP2400) or 8k (IXP2800) instructions. Only one thread is executed at any moment while the others wait until they are scheduled. There is no preemption on these processors, instead, threads are rescheduled every now and then automatically by hardware when a thread is waiting for a signal to be delivered. It is mostly because of synchronization and memory accesses as explained later. When a thread is waiting for a signal that has not been delivered yet, the thread is removed from the ME and other runnable thread is scheduled. The programmers goal is to write his code in such a way that there is always a runnable thread and the ME is not idle.

MEs use unusual asynchronous memory access. When an instruction needs to read or write data to or from memory, it issues a transfer and waits for a signal that the transfer is finished. Waiting for a signal would make the ME idle, therefore another thread can be scheduled. The ME becomes idle only if all threads are waiting and none is runnable. This results in swapping running threads frequently. Of course, a thread can also execute other instruction between issuing a memory transfer and receiving a done-signal. If processing the current data chunk takes enough cycles, it can happen that the signal is delivered to an ME even before it executes the wait-for-signal operation. In such a case no other thread is scheduled. We discuss this later in this chapter.

Besides the asynchronous memory access, the memory controller allows MEs (in contrast to XScale) to use other features that highly improve performance. A short list is

presented here and others are discussed later as need be.

- *rings* - hardware support for rings of different size (elements count). They allow enqueue and dequeue atomic operations to store and remove 4 byte elements. Using them avoids locking. Both Scratch and SRAM support rings.
- *atomic operations* - atomic operations can be used to implement synchronization primitives, to test and set flags by multiple processes and, e.g. to acquire buffer slots in DRAM atomically

Because each ME has limited speed, it may (e.g., RX/TX tasks) or may not (e.g., routing, pattern matching) be capable of executing the whole task. Therefore it is a common way to distribute the work-load over several engines. Each ME can run exactly the same code or it can be split in several stages to use the MEs as a pipeline, as proposed in [17].

5.1 IXP programming model

The main bottleneck of the architecture of the extended FFPF on the IXP processor, as discussed previously, is passing packets between atomic filters. Implementation in the PBuf/IBuf fashion is acceptable for general purpose CPUs which lack support for hardware queues and rings, in contrast to an IXP. The main disadvantage of such a buffer implementation is a problematic multi-threaded access. Storing a packet in a buffer is hardly an atomic operation. If more threads want to store (change or remove) a packet, they must be mutually excluded. In general, this is done by locking or some other means of synchronization. Locking is quite an expensive operation and if used too often it degrades the overall performance. On the other hand, avoiding synchronization leads to incorrect results and corrupted buffers. For example, Linux networking subsystem uses Read-Copy Update (RCU) [18] to minimize locking as much as possible by modifying a copy.

In the PBuf/IBuf way producers and consumers are synchronized only by incrementing read/write pointers in a buffer head. This is suitable when there is only one producer. The write pointer is incremented no sooner than the packet is written to the buffer. If there are more writers, it is difficult to determine when and how much to increment the write pointer. Even if we guarantee that a buffer slot is accessed by a single writer, we have to make sure that a buffer slot is not read before the packet is completely written. This may happen when the write pointer is incremented too early. Having multiple consumers is also not trivial. If all of them have to read all data in a buffer, each can keep its own read pointer. However the producer must always compute the minimal read pointer, otherwise a slot that was not read yet by some of the consumers, might be overwritten. Even more difficult is a situation when each buffer item can be processed only by a single consumer. In this case, consumers must share the read pointer, which must be incremented atomically.

All these problems can be solved by using hardware support for atomic variables and rings. Especially rings enable implementing high-performance lock-less buffers for passing packets between execution units.

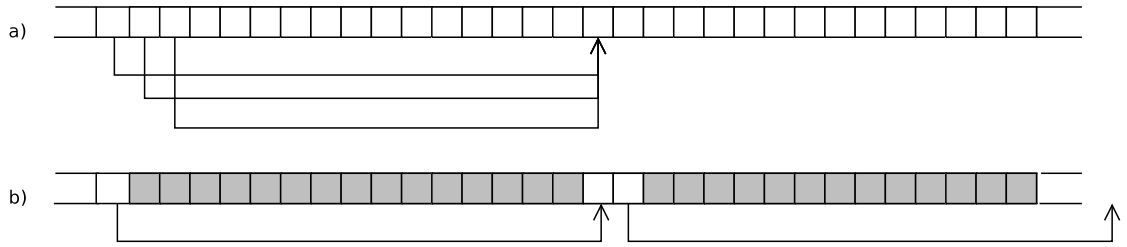


Figure 5.1: Comparison of asynchronous (a) and synchronous (b) memory model - arrows show when data transfers are initiated and when they are finished. Each square represents a single machine cycle. The white ones stand for cycles that are available for executing instructions, the gray ones are lost because of synchronous memory access.

The following sections describe what design patterns are used in general by IXP developers as well as how we use them in our applications.

5.1.1 Memory

The most important thing that makes MEs different from ordinary CPUs is asynchronous memory access. Memory latency is a well-known issue and manufactures are trying to work around it as well as they can. One approach is to use cache memories for storing often used data close to the CPU in a small but a very fast memory. There are many different configurations (several levels of caching) and various algorithms that select which data should be stored in cache and which not and how the main memory is updated, for instance write-back and write-through policy. Since the caching is transparent to the code running on such a processor, it almost does not affect application design[1]. However, there are applications that touch all or most of the data only once and caching does not help. In such a case the CPU has to wait until data are fetched from the main memory.

Packet processing clearly belongs to the set of applications where most of the data are touched only once. If it is known in advance what chunk of memory will be accessed (as in sequential packet processing), data can be prefetched to a buffer or cache. If there is other work to do between the moments when the decision to prefetch data is made and the moment when the data are ready to be used, we can keep the CPU busy instead of idle (Fig. 5.1). This leads to an asynchronous memory model that is adopted by IXP MEs. The advantage of this model is that the programmer can hide most of the memory latency or even all of it. The disadvantage is that the programmer must design code with special care. In addition, It is still a challenging task for compiler designers to develop efficient optimizers for such processors.

Asynchronous memory access with a hierarchy of memories that have different sizes and speeds makes the *"caching"* entirely exposed to the programmer and the programmer has to take in account what data-structures are placed where in which memory. In a common design the memory is used as follows :

- *DRAM* is a memory with the highest latency, but it is large and has a decent throughput once data are read sequentially. Therefore DRAM is mostly used for storing

[1] programmer or compiler can produce code more or less suitable for certain cache environment

packets.

- *SRAM* is suitable for larger structures, like various headers and descriptors that may be accessed randomly. Since the SRAM is split in two independent banks, it is wise to store data that are read or written at the same time to different banks
- *Scratch* is only 16 kilobytes in size and is primarily used for frequently accessed structures, queues and for atomic operations.
- *Local memory* has 640 cells, 4 bytes each. It is local to an ME and is accessed via 2 index register. Each thread has its own pair of registers, but a single pair can be shared by all threads of the ME. First access costs 4 cycles because of loading the registers, but any consecutive access is as fast as if data were stored in a general purpose registers.

Another benefit of asynchronous memory is that more transfer requests can be issued consequently. Without this feature, a CPU has to wait until one request is finished before another request can be started. This approach saves many cycles because all requests are processed in parallel as you can see on Fig. 5.1. Arrows show when an ME begins a transfer and when the transfer was finished. Each square is a cycle. Gray ones represent cycles that were lost due to waiting for memory controller to finish memory access. Clearly, we can see that the IXP model on **(a)** has more cycles available for useful work. Although, if there is nothing to execute while waiting for a done-signal, we waste only a fraction of cycles compared to a synchronous memory model **(b)**. Of course, there are situations where a synchronous access is required. In this case rescheduling and letting another thread run hides the latency.

5.1.2 Atomic variables

The IXP memory controller offers a set of atomic operations to be used with Scratch and SRAM memory. All of them are available for the MEs whereas the XScale core can use only a limited subset. Atomic operations can be used to access 32 bit wide entities. There are no special operations for reads and writes since these are atomic by nature. All atomic operations work in the read and modify way.

Atomic operations are used for two main reasons. Firstly for synchronization, secondly for incrementing and decrementing in-memory counters without need to wait for operation completion[2].

There are several ways of how to use atomic operations for synchronization or interprocess communication :

1. *Flags* - if actions performed by a process are determined by another one, we can signal it by setting some flags in a data structure. We cannot achieve a consistent operation first by reading the flags field updating and storing it back to memory. Doing so concurrently by multiple threads would lead to loosing some flags or setting a flag that was previously deleted by another thread. Atomic operations do not suffer from this disadvantage.

[2]For more details, see the IXP programmer's reference manual

2. *Bit-locks* - locking can be implemented by a test-and-set bit operation and unlocking by a clear bit operation. Such locks are implemented as spin-locks that are know from commodity operating systems. If there is only a single thread per ME, busy waiting does not harm performance. If there are more threads running on a single ME, the one which wants to acquire a lock does not exclude other threads from running. It is because waiting for a test-and-set operation to complete schedules another runnable thread.

   Since executing one spin of a lock cost only a few cycles, interleaving it with other processes is acceptable. If many threads do the same the overhead can be significant. Especially in scenarios when all threads on an ME are trying to acquire a lock that is held by a thread on the same ME. All of them get a chance to test if the lock is already available. In certain cases, it is possible to avoid it, e.g., by writing non-preemptible code inside the critical section or using signals. Such a solution depends very much on the application and can be hardly generalized.

3. *Acquiring buffer slots* - since we have more execution units that can run many threads in parallel, we don't want them to wait on each other because of writing data to buffers. We can use atomic an test-and-increment operation to acquire a slot in a ring buffer atomically. When a thread wants to get a slot, it reads the actual write pointer and prepares a new value for another thread. This guarantees that only a single thread is storing data to the acquired memory location and data are not being overwritten by others.

It is possible to think about other options of how to use atomic variables and operations. For example instead of using bits or flags as locks, we can spin until a counter reaches a certain value and then let a thread in the critical section. We use a similar approach for implementing non-reordering locks that are discussed next.

5.1.3 Non-reordering locks

In packet processing, it is important not to reorder traffic. Some protocols like TCP[3] can handle out of order arrival of packets. However, it reduces the speed of a connection. For other protocols, like real-time audio/video, keeping the packets in order is critical, since the end point can rarely wait until the out-of‘-order packets arrive. Reordering in-order packets can also slow down processing within the monitoring application, as discussed in Sec 6.3 on TCP reassembling.

Not to reorder packets, we need to avoid reordering threads that handle packets. Using bit-locks may break such a condition. In a scenario where a thread spends too much time inside a critical section, it can happen that other threads are trying to acquire a lock. As described before, they are spinning and time to time each one gets a chance to test the lock. There is no guarantee that the first thread that tried to enter the critical section is also the first one to get the lock. The simplest way to solve this issue is to put threads on a waiting queue as they are arriving and wake them up one by one.

We do not have any waiting queues were we can put threads to wait for an action to happen. Having a fixed size queue per event, which is the faster option, consumes too

[3]Transmition control protocol, RFC 793

much memory. Using linked-lists as waiting queues involves a lot of memory access, which we want to avoid primarily.

Instead, we use two atomic counters. Both are initiated to the same value. Whenever a thread wants to enter a critical section, it gets a token from the first counter and increments it for use by another thread. The current value of the second counter signals, which token is allowed to enter the section. If a thread owns the tokenof the same value, it can freely enter the section. Otherwise it has to spin in the same fashion as when using bit-locks. Finally, when a thread is exiting the critical section, it atomically increments the second counter and allows the next thread to enter.

Because each counter takes only 4 bytes and the relatively large SRAM supports atomic operations, this solution scales much better with growing numbers of events or memory structures that require synchronization. On the other hand, this solution suffers for the same problems as the bit-locks when there are many threads waiting to enter a critical section.

5.1.4 Rings

The most important hardware feature is the IXP support for rings (also called queues). This enables to put elements into a queue without need of locking. And the same holds for retrieving them again. Rings are supported by both Scratch and SRAM memory types. A ring is a circular buffer of 4 byte elements with a fixed number of slots, which is configurable. In hardware they are implemented by keeping the head and tail pointers. Scratch memory supports only a limited number of rings, whereas the number of rings in SRAM is limited only by its size. On the other hand, each SRAM channel has only a table for 64 rings that can be used at the same moment. If an application needs to use more, descriptors in the table must be swapped, which slows down access to the rings. Also the number of elements per scratch rings is much more limited.

In most applications, scratch rings are an ideal solution for inter-process communication. Producers place data (indices) on a ring of about 128 or 256 elements and consumers pop them off the queue. There is no need for a huge queue since it is expected that consumers can keep up with producers. Moreover, producers can check (in a single cycles) whether the ring is full and therefore if data must be dropped. Using larger rings does not make sense. If consumers cannot keep up with producers, dropping would be postponed only by a few packets. The only way how to avoid dropping is to design application for the peak bandwidth. However, these rings can help to absorb short bursts, SRAM rings are better suitable for long bursts though.

In contrast, the slower SRAM rings can have up to 64k of elements, and are perfectly suitable for implementing pools of unused structures. Whenever a thread needs to allocate some data-structure, it just pops it from a ring. Popping is not a blocking operation and if the ring is empty, it is signaled. After the structure is not needed anymore, it is just placed back in the pool. Reasonably fast runtime allocation without this hardware support in such a multi-threaded environment would be otherwise impossible.

5.2 ENP2611 as an Ethernet device

There is no common way how to integrate the ENP2611 board with the host Linux system. It is because of the complexity of this device and because each usage may require a completely different approach. We opted for an ordinary ethernet driver, similar as discribed in [19] for the older IXP1200 serie. The benefit of this approach is that any host application that wants to use it does not require many changes if any at all. The only thing that makes it different from other ethernet devices is the need of uploading code to this card. Since Linux 2.6 offers a standard way of uploading firmware to a device, our driver does not differ that much.

From the user's point of view, the driver offers a transparent way of uploading an application by writing its code to a special file in *sysfs*. This application is started at the moment when the user tries to bring the interface up. Without loading an application (or firmware) in the ENP card, it cannot even receive a single packet. By supplying different versions of firmware users can easily change the board's behavior, even though implementing the firmware can be quite complicated. The following Chapter 6 presents that also generating the firmware can be automatized to a certain extent and a user needs to know only a simple language to be able to reconfigure the this board.

The driver is split in two basic parts. The host part is an ordinary driver that must be loaded to the kernel before one can use the card. The other part is running in the Linux kernel on the XScale core and is part of the blackbox. Its role is to receive signals from the host-part of the driver. There is a userspace daemon on the XScale, which waits for requests to load the MEs with code and to start or stop them. The XScale code is included basically because of the Intel IXA framework and code-loader. It is possible to load and govern the MEs directly via the PCI bus, but the Intel compiler produces object files that must be loaded in this complicated way.

The ethernet part of the driver closely cooperates with the micro-code, which is sending packets over the PCI bus. Since we don't want the micro-code to wait until the packet is processed at the host side, we are allocating a buffer for 128 packets in the host memory. Packets are stored in this ring buffer and the host kernel is notified by an interrupt. By storing data in a buffer, we can implement the receiving part of the ethernet driver in the NAPI way (as described before). After the kernel is notified that there are new packets pending, it schedules the polling routine of the driver and packets are picked up and processed later on. Consumed buffer slots are returned back to the micro-code to be filled with new incoming data.

Since the card is a part of a passive monitoring system, the driver does not support packet transmition. However, there is transmition on the card itself and packets that are not redirected to the host or dropped, are routed and sent back to the network.

Chapter 6

Implementing Ruler on an IXP

Ruler is a language developed by Kees van Reeuwijk [20] to describe pattern matching and packet rewriting rules. Based on these rules the Ruler engine can generate a deterministic finite automaton (DFA) for matching these patterns in parallel. It is called a tagged DFA since it can mark positions of sub-patterns within the input by tags. These tags can be used to rewrite the input to a different output. As presented in our work [21], we use this language for classifying packets in an intrusion detection system (IDS). We try to detect attacks by inspecting both packets and whole TCP streams. Ruler allows us to match patterns (e.g., malicious values) in packet headers as well as in the payload. Since data passed to offline processing can contain private information, we can anonymize the traffic by rewriting packets and deleting irrelevant content. To do so, we use the tags.

There is also a tool that can translate most of the Snort rules to Ruler rules. Snort [22] is a standard tool for packet monitoring and large sets of well tested rules exist, both paid and free.

```
filter test1
       "ABC" => accept 1;
       "A" * "B" * "C" => accept 2;

filter test2
        ("A"|"a") * ("B"|"b") * ("C"|"c") => accept;

filter test3
        "A" * "B12" * "C456" => accept;

filter test4
        "ABBBCCC" * => accept;
        "AAB" * => reject;

filter test5
         start:("aaa") * middle:("ZZ" * "ZZ") end:(*) =>
                                                 end start middle;
```

Figure 6.1: A sample set of Ruler rules

You can see an example of Ruler rules on Fig. 6.1. The left-hand part of a Ruler rule is a pattern matching expression similar to regular expressions. User can name parts of the expression to remember positions of the sub-match and use the name in the rewrite part to copy the substring. Ruler is not limited to packet matching, but it is its main purpose. Since a packet usually has a well defined structure, and some parts are not necessarily interesting or relevant for an application, Ruler can jump over them without any inspection. This speeds up processing.

The right-hand side of a rule says what to do with a packet if the pattern on the left-hand side matches. There are several options. A packet can be simply rejected or accepted. Optionally an integer value can be assigned to accepted packets. Finally, a packet can be rewritten if a rewrite rule is specified. New packets can be composed of new byte strings or by parts of the original packet that were tagged on the left-hand side of the rule.

The DFA itself contains inspection states (to check an input byte), jump states (to skip a number of bytes), tag states (to set a tag value) and match states (that execute the right-hand side of the matched rule). Since some packet headers have various lengths (e.g., IP and TCP headers because of option fields), the DFA also contains so-called memory inspection states that can use a value on a given position in a packet to compute the next action. At the moment there is not other use for such states than skipping IP or TCP options.

Fig. 6.2 shows a simple DFA which is matching `hello` anywhere in a packet. Circles represent inspection states, a pentagon is for a match (in this case accept) state and squares are edge labels. How the set of input rules is converted in a DFA is out of scope of this work. Speaking in compilers' terms, the Ruler engine is a front-end that produces a target independent representation of that DFA. In the rest of this section we discuss the IXP back-end that generates code for the IXP micro-engines.

6.1 Inspection states

Besides jump, tag and match states, the core of the Ruler DFA are inspection states. Only these states decide what path in the DFA is taken. Since the vast majority of execution time is spent in these states, we have to optimize this code.

In general, a memory inspection state has two parts. First, it must check whether there is not an end of the input. If not, it must get a new data chunk from the input. The second part is inspection of this chunk. Inspection states work with the granularity of a single byte, however, it is possible to match single bits. This is done by a trick inside Ruler, which generates all possibilities for a byte based on bit mask in a rule.

6.1.1 Getting input bytes

As explained in Sec. 5.1.1 accessing memory for each byte is not possible. Not only would reading a single byte from DRAM be unbearably slow, but it is also not possible. DRAM granularity is 8 bytes and we cannot read smaller chunks. Therefore we implement our own caching. Each read from DRAM (where the packets are stored) can transfer up to 64 bytes of data. We store them in a local memory buffer and while the

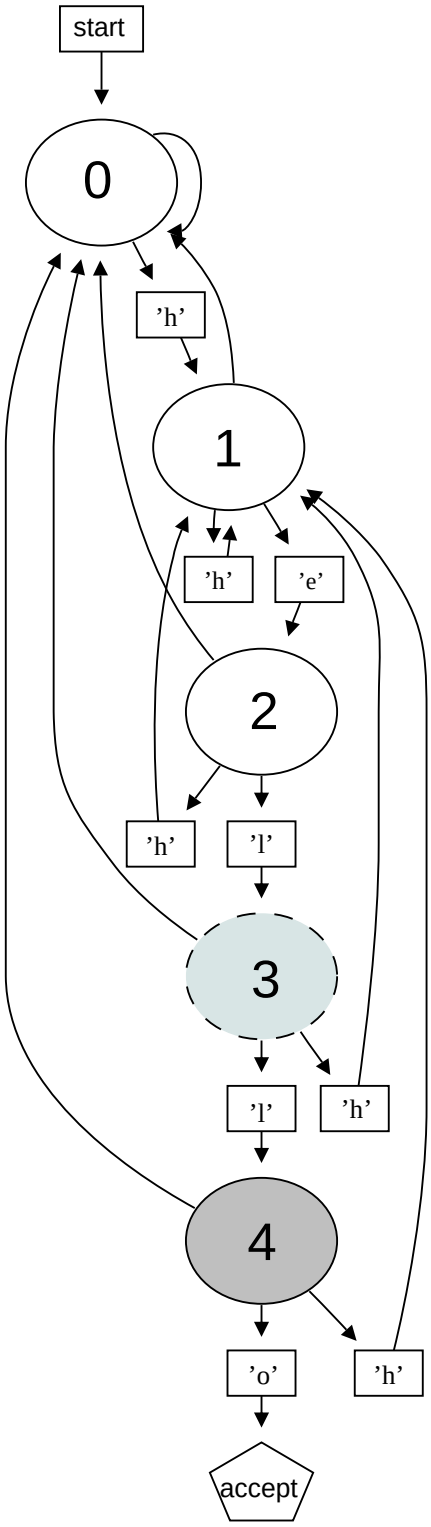


Figure 6.2: Ruler sample matching "*hello*"

current chunk is being processed we issue a new transfer from DRAM. This mechanism has 3 main features :

1. Consuming a whole 64 byte buffer takes so many cycles that the new chunk will be already received and ready for processing. So the DRAM latency is completely hidden.

   As a consequence, no other thread is scheduled on the same ME unless the current thread yields the processor voluntarily.
2. The first 64 bytes cannot be prefetched and must be read synchronously after we know were is the packet stored. First, the packet location must be computed, the data fetched from DRAM and only after this we can start the processing.
3. We prefetch data beyond the packet since we cannot decide early enough if there is an end of a packet already. This prefetch does not harm performance.

The code is designed as a subroutine and is called from each state to return the next byte and prefetch data. The part that returns a byte from the cache chunk can be inlined in the inspection states. This saves a few cycles per byte which makes a significant difference on a gigabit link. It saves about 20% of execution time of the smallest inspection states. However if we are running out of instruction store, we can optionally choose not to inline this and save some instructions store for other states. It makes sense only if the DFA fits in the the instruction store afterwards.

6.1.2 Switch statements

Unlike getting a new byte, which can be achieved in constant time, the number of cycles needed for inspecting that byte varies very much. Basically, determining what action to take based on the input byte is an ordinary switch-statement known from C-like languages. Different methods are used in modern compilers to speed up switch-statement execution. Examples include implementations based on hash tables [23] or trees [24]. The IXP code generator implements several algorithms and has an infrastructure to add more. Since we don't have enough memory to store a hash table for each state and because accessing this table would outweigh the performance gain, we decided to implement a binary tree in the instruction store as the most complex algorithm.

The observation we made is that the default branch of a switch statement is the most often taken. It is because in most cases no matching string is in the packet and the default branch is the one that just consumes the input in a star-loop (as a '*' in regular expressions). It is the most expensive branch since it is taken only if all other tested options were rejected. We create trees such that the average number of cycles spent on a default option is minimal. In each node one or more bits of the input byte are checked to decide which of the two subtrees to search.

This approach has more problems then benefits, though. First, the tree consumes a lot of instructions. Secondly, testing proved that the generated code is superior to a trivial switch-statement implementation only if the input is random enough. Unfortunately this is not the case for packet headers and many protocols that are used in practice. For example HTTP[1] is a human-readable text protocol with not an equal distribution of characters.

[1] Hyper text transfer protocol RFC 2616

On the other hand binary data or data sent over encrypted channels are good input. However, it is difficult to find any useful patterns in such streams. At the end of the day we decided to use a trivial comparison of a byte to all possible values with only a simple improvement where applicable. This improvement separates bytes that do not belong to the range of the non-default options. Since such a test also consumes some cycles, it is excluded where executing a trivial statement with only a few options works better.

We had to postpone our efforts in this area until some more research can supply information about the distribution of characters in the network traffic. It would be necessary to analyze Ruler rules first to determine what kind of traffic the user is interested in before such a key information for switch statement generation can be obtained.

6.1.3 Instruction accounting

Because the number of DFA states might explode, we opted for splitting the DFA. Essentially, we support storing some states in memory (mostly SRAM, but Scratch is also an option). When we are generating the IXP code, we do not care about the number of instructions that was generated. First we need to know how many instructions are consumed by the whole DFA and after that we need to add how many instructions are needed for other support routines. These routines must be placed in the instruction store as there is no other option. The routines are responsible for getting the next packet, TCP streams, interpreting in-memory states, etc. Therefore the support routines are not simply included as separate files, they are encoded in C wrappers that increment a counter whenever an instruction is generated, and they are part of the code generator's source code.

We do not account only how many instructions were generated for the entire code, but we also keep track of how many instructions are used per state. This allows us to split the DFA so that the almost all the instruction store is filled with useful code and instructions are not wasted. Of course, this highly depends on the nature of the states.

When some states are placed in memory, we have to take into account that an interpreter must be also included in the instruction store.

The DFA is basically a tree with backward edges of loops that are consuming input bytes by a star pattern. We assume (and practical experiments support this theory) that the hottest states (most frequently executed states) are close to the starting state. Therefore we first try to move the most distant states to SRAM and keep the hot states in instruction store.

To implement this, we assign a so-called level to each state of the DFA which represents the number of hops on the shortest path from the starting state. The algorithm iterates over levels, from the deepest one and tries to remove states, one by one. Until the number of instructions is reduced enough to fit in the instruction store.

Notice that it is not possible just to delete a state's code and place it in memory. We need so-called memory entries that tells the interpreter where the interpretation should start. Whenever a state is removed from the instruction store, it is substituted by a memory entry stub. This stub cannot be removed as long as there is an edge from an in-instruction state directly to this state. Therefore, when we are moving a state to memory, all its outgoing edges must be checked whether they point to a memory entry stub that can be safely removed after the current state is placed in memory too.

To illustrate this, consider the DFA on Fig. 6.2. Suppose that the instruction store is so small that the 6 states cannot fit in. After splitting the DFA we want to place state 3 and 4 in memory. Since there is an edge from state 2 to state 3 it is a memory entry state (light gray). The state has an entry in the in-memory table, but there is also a stub in the instruction store that jumps to the interpreter. In contrast, state 4 is a pure in-memory state (dark gray), there is no incoming edge from the instruction store. Match states are always encoded in instructions.

6.1.4 Interpreting memory states

Interpreting states that are stored in memory is fairly simple. Each state is a row in a table of all in-memory states. The id of a state determines the base address of that row and the input character decides which cell to use. We have two different kinds of cells. One stores just an id of another in-memory state. The other stores an address where to jump back in the instruction store if the execution exits the interpreter. These two kinds are distinguished by a bit-flag.

At first glance, it might look the in-memory states are significantly slower. However, this is only true if the memory latency is not hidden by executing another code. If more threads are running, execution of an in-memory state is bound to 30 cycles, which is only slightly more then cycles consumed by an average state implemented by instructions. Moreover the number of cycles does not depend on the complexity of a state. Secondly, since threads have to reschedule because of memory reads, the ME is more fairly used by all threads. This is not true for code that executes only instructions. To get a similar fairness, we have to force the code to yield the ME from time to time voluntarily.

6.2 Packet rewriting

Packet rewriting is a complicated issue and the IXP hardware is not the best choice. It is a very memory bound task and much better suited for general CPUs as our other experiments proved. On the other hand, reasonably simple rewriting is possible on a high bitrate and rather the matching DFA is the bottleneck, though. In general, Ruler can rewrite an input packet to whatever output. It can release a part of a packet, it can swap two parts and add new ones. Clearly, on this level of generality, the task uses memory very intensively, unaligned data require special and expensive handling and the local memory, that is the only suitable for a temporary buffers, is too small.

To overcome all presented difficulties, we assemble the new packet in a local memory buffer that is only large enough to handle a single full sized ethernet packet at any moment. As a result we implement packet rewriting only for a single thread per ME. Using the local memory has the benefit of fast access that is especially required when appending a new unaligned chunk to an unaligned tail of the newly constructed packet. The tail must be first read and only after that the rest can be appended. This is not possible with any other type of memory at a reasonable speed.

The rewrite action is represented by a chain of structures, each describing one of two possible atomic actions. One copies data from the original packet to the new one. The other inserts a byte-string. Interpreting such a chain uses similar caching mechanism

as processing the packet. It also first prefetches a chunk of 32 bytes (SRAM limit) from memory and executes actions as long as all the bytes were not consumed. At this moment a new piece of the action chain is prefetched and ready.

6.3 TCP reassembling

TCP streams carry a lot of the network traffic. If an attacker sends some malicious data via TCP, the dangerous content is likely chopped into pieces by the TCP stack and is distributed over several packets. For this reason scanning single packets in a security context is not enough and we have to go beyond. We must identify and reconstruct the TCP protocol and scan its payload as a contiguous stream of bytes.

This chapter describes how TCP packets are preprocessed before being passed to the Ruler TCP processing, which is presented in the following Sec. 6.4. We identify each TCP flow and separate packets belonging to each of the flows. Since some packets might be missing we must take certain measures to assure that the flow passed to Ruler is reconstructed.

TCP reassembling distinguishes 2 kinds of packets: in-order packets that arrive with an expected sequence number and out-of-order data that must be placed in a separate buffer to wait until a missing in-order packets arrive. Naturally, there are various ways how to reassemble a TCP stream. We can either drop all out-of-order packets and let the end points of the connection deal with this as they have to do with ordinary network failures. Basically, this is not reassembling, we deal only with in-order data. Or we can do a full TCP stream reassembling.

Since accessing a lot of memory structures (e.g., walking linked lists, etc.) consumes too many cycles, and doing nothing is also an ineffective option, we decided to implement a limited version of TCP reassembling and to leave the rest for the endpoints.

Our TCP reassembler does not implement a full TCP stack, nor complete features of the TCP protocol. Its basic purpose is to handle simple problems like a lost (rather delayed) sequence of packets without dropping out-of-order packets. Out-of-order packets does not have to be retransmitted by the end-points. More serious corruption of a TCP stream leads to dropping of at least some packets. We assume that this is not a common case on a gigabit link near one of the end-points. In addition, our tcp reassembler does not copy any of the packets' data, only indices are placed in buffers.

However, this version of a TCP reassembler assures that all packets that are passed to following stages of processing (e.g, tcp-ixpruler) forms a contiguous flow of tcp sequence numbers.

Fig. 6.3 presents an overview of the packet path through the TCP processor. Thin arrows show how a packet travels, whereas thick arrows describe how TCP flows are scheduled by the TCP reassembler and grabbed and released by Ruler engines. After the receiver receives a packet, it stores its index in a ring that is polled by the TCP reassembler, which places packet indices in stream buffers. Streams are then scheduled for processing. After grabbing a stream from the scheduled-queue, Ruler keeps processing its packets until the buffer is empty. Indices of processed packets are placed on another ring that is polled by transmitter, similar to the receiver-reassembler commutation. If the

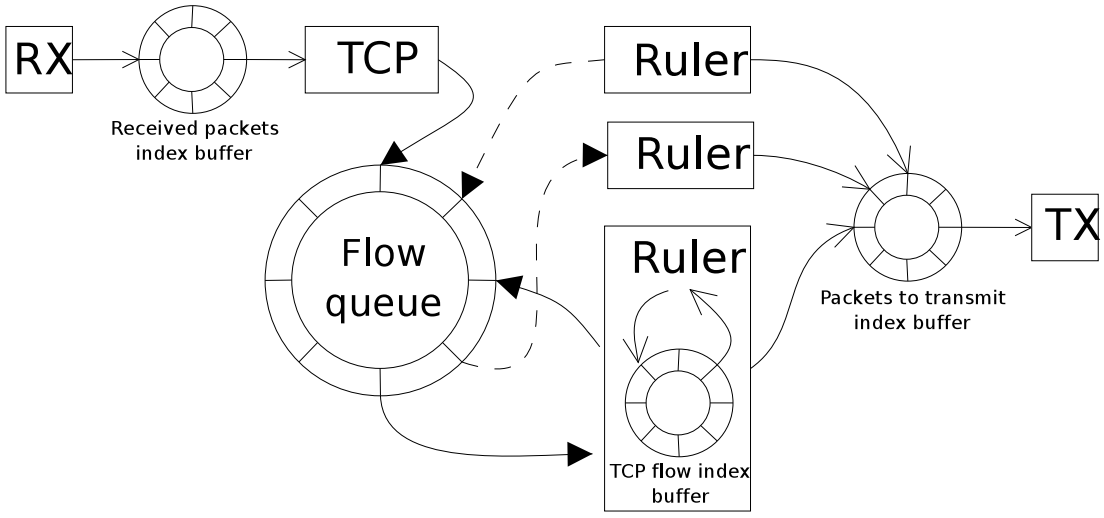


Figure 6.3: Packet path through the TCP processor

TCP reassembler is not used, Ruler engines poll on the receive-buffer directly. Recompilation for pure packet processing of Ruler is required.

Limitations of the current implementation

- The number of simultaneous TCP flows is limited to 1536 due to limited amount of memory
- The number of out-of-order packets kept in a separate buffer is limited to 64. Since we do not copy data from out of order packets to any special buffers and we just keep pointers to packets, the number of bytes is fully determined by the incoming packets. We do not assure any number of bytes that will fit in the buffer, except the number of packets.
- Once the out-of-order buffer is full, we start dropping out-of-order packets that belong to this flow until at least one entry in the in-order buffer is freed. The end points of the affected TCP flow will retransmit the dropped packets. This slows down this TCP flow, but it does not harm correctness of the TCP protocol.
- Because of the performance reasons of our TCP reassembling implementation, we assume that the most common reason for out-of-order data is loss of a few packets, whereas the rest arrives in-order. Therefore we optimize for this particular case and we may drop packets instead of buffering them in other cases.

6.3.1 Reassembling

As mentioned before, TCP reassembling classifies packet in two groups, depending on whether they were expected or not. Each group of packets is handled in a different way. This section describes how are incoming packets treated and how the TCP reassembler uses its internal buffers for TCP stream sanitation.

In-order data

In-order packets are placed in an in-order ring buffer. Each index in this buffer keeps information about which packet is referenced, the length of this packet and the payload offset (how many bytes should be skipped by Ruler). Of course, there is space for flags as well.

An in-order packet buffer has 128 entries. Once all are used (the ring is full) and and a new in-order packet arrives, we place this packet in the out-of-order buffer. The idea is that it is better to drop out-of-order data than in-order, which would have been retransmitted. Because it is too expensive to shift packets in a ring buffer (or insert a packet between two others) we drop all packets in the out-of-order buffer to make space for in-order data.

Also, whenever an in-order packet is placed in the in-order buffer, we try to flush the out-of-order buffer and append as much as possible of its packets to the in-order one. There is a simple test whether it is possible or not. If the head of the out-of-order buffer does not succeed the last in-order packet, we cannot append anything.

Out-of-order data

If a packet does not have the expected sequence number, it is an out-of-order packet. Such packets are not directly visible to the consumer, they are placed in a separate buffer. The task is trivial if the buffer is empty. If it is not, we have to make sure that the buffer is not full and that the actual packet is the expected one in the out-of-order sequence. As we mentioned above, we also keep out-of-order packets in a sequence. At the moment we do not support more complex corruptions of the TCP stream. There are two main reasons :

1. Either we would have to insert packets in the buffer between two other packets. This is not a complicated task, but it involves a lot of copying. That is too expensive. Another option is to use a linked list instead of a ring buffer. This is both complicated and too memory consuming.
2. All solutions suggested above are too slow because of extensive memory usage.

Nevertheless we keep more information in the out-of-order buffer than we actually need to enable future extensions to the current scheme, e.g., a sequence number for each packet. Moreover, we keep the sequence number of the first packet as well as the number of payload bytes in the buffer. This enables us to compute the next expected sequence number without need of reading the last packet descriptor. In addition, we know how many *contiguous* packets follow the head of the out-of-order buffer. We can determine how many packets can be moved to the in-order buffer, plus it allows us to place packets of a more corrupted TCP flow in the same out-of-order buffer in the future.

As mentioned above, we flush the out-of-order buffer and append packets to the in-order buffer always after an in-order buffer is received. We do the same when a FIN packet arrives. Of course, there is not always space enough to copy out all packets to the in-order buffer. The buffer is kept as a ring and emptied slots are "appended" to the end of the ring If this buffer is full or if the arriving packet does not continue the sequence, it is dropped.

6.3.2 Hashing

We keep all flows in a hash table. Because of handling hash-collisions, we opted for hashing with linear chains. We decided to use linear chains since it is the most effective solution for hash tables with frequent *remove* operation. We also assume that hash collisions are rare. We also use hardware support to speed up computation of the hash value. The input of the hash function is the unique identifier of a TCP flow, formed by source and destination IP addresses and port numbers. The output is a 128-bit hash value. The hardware function (fully described in the Intel IXP2400 hardware reference manual) is designed so that the lowest bits vary the most. Therefore we use only lowest 10 bits.

Hash table

The basic hash-table structure is an array that points to hash-collision chain heads. It has 1024 entries and is stored in the local memory (LM). Each entry is 16 bits wide, therefore a single 32 bit LM cell is shared by two entries. This saves a lot of memory (which is a very limited resource) and is traded for a little bit more complex hash-table operations. Storing twice as many hash chains also decreases frequency of hash collisions. In addition, using LM speeds up checking if a hash-chain for a given hash-code exists. It is an order of magnitude faster then using the second fastest available memory type (scratch pad).

TCP flow descriptors are split in two parts (scratch + SRAM). Since usually at least one TCP port number is automatically generated, we assume that the pair of ports is mostly enough to decide, whether the flow was NOT found. Scratch also contains pointer to the next member of the chain. Therefore walking a hash-collision chain needs synchronous memory accesses to check if the requested flow was found.

The IP address must be checked to confirm a hash lookup hit. Since IPs are too large, we store them with other TCP related data in SRAM.

Locking

Our goal is to enable multiple threads to access the hash table simultaneously. Therefore we want to lock only as small a part of the hash table as possible. However, we need to protect concurrent access to hash-collision chains. The reason is that some thread might be removing a member of the chain whereas another thread is walking the same chain, etc.

Simple locking is not enough, though. As already explained in Sec. 5.1.3, trivial locking implementation is a source of reordering packets that belong to the same tcp flow and were received in-order. Again, imagine the following scenario :

- Several packets of the same flow are received in sequence.
- All threads are spinning on the same hash-collision chain lock.
- The flow is unlocked and a thread that handles a packet, which was not the first one in the sequence, acquires the lock.
- The result is that this packet is served sooner than the leading packet of the pending sequence. This means that this packet is handled as an out-of-order packet.

A solution to this problem is to grant entry to the critical section in the same order, in which the threads tried to acquire the critical section lock. Using the non-reordering locks as presented in Sec. 5.1.3 does the job. We have such a lock per hash-collision chain as well as per tcp-flow descriptor.

SYN packets

A hash table entry is opened when a SYN packet is received. A hash lookup must be performed to ensure that a flow with the same signature is not opened yet. If so, the SYN packet is dropped.

To open a new flow, an unused tcp flow descriptor is required. The TCP reassembler is initialized with a pool of unused flow descriptors ids, which is kept as an SRAM ring. Whenever a new descriptor is required, it is enough to get one from this ring. If there are no more unused flow descriptors, the maximal number of simultaneous flows is exceeded. In this case the ME is halted and XScale is interrupted. The XScale then gets a chance to resolve the lack of resources.

After a flow descriptor is filled and written back to memory, the flow is inserted in the appropriate hash collision chain. We do not handle an unlikely corner case when a SYN packet is immediately followed by data packets since this is naturally not possible. Such data packets will be dropped because of a hash lookup miss.

The SYN packet is placed as the first packet in the in-order buffer. Its payload offset (headers' length) is set to zero. As a result the tcp-ixpruler will process also all headers of the first packet which gives more information about the entire flow.

FIN/RST packets

When a FIN or RST packet is seen, the flow should be closed and removed from the hash table. The situation is usually trivial in case of a RST packets (RST and FIN handlers are identical at the moment) since RST is mainly used during incorrect processing of the three-way-handshake.

In contrary, a FIN packet might be received when the TCP flow is still being processed by a consumer (e.g., ixpruler). We can receive FIN in several common situations :

1. There are no pending data and a consumer is neither processing the flow nor the flow is scheduled for processing. The flow is only removed from the hash table and its descriptor is returned to the pool of unused ones.
2. There are no pending data in the out-of-order buffer and the flow is either being processed by a consumer or is scheduled to be processed. In both cases, the flow is marked as closed and the flow descriptor is removed from the hash table

   It is the consumer's responsibility to return the flow descriptor to the pool of unused ones after it finishes processing of the stream.
3. There are pending data in the out-of-order buffer. This is certainly a bad situation. In this case we drop the FIN packet and wait until the missing data arrives. Either the endpoints should handle this or (in case of their crash) the XScale cleaner will release this flow at some point.

Mutual exclusion with consumers

Because of the situation 2 mentioned above, we need to make sure that the flow is not released by the consumer while the TCP reassembler is about to set the `CLOSED` flag. Therefore the TCP reassembler must lock the stream while it processes a FIN packet. The consumer thread must do the same when it is postponing the flow, which it is handling at the moment.

Flushing a flow buffers

If a flow is marked by a `DROP` flag and a FIN packet is received, there might be some pending data (old) data in the buffers. All packets must be dropped and the buffers cleaned. Consumers do not take care of this, since it would imply heavy locking and consequently a significant slowdown.

6.3.3 Dropping flows

Since the TCP reassembler was originally designed as a part of an IDS, it is assumed that the consumers (following stages), may decide to drop a flow. This action needs tight cooperation with the reassembler. Our goal is to minimize this as much as possible, to avoid locking and therefore unnecessary slowdown. The result is that the consumer only marks flows by a `DROP` flag. This must be done under a flow-lock. Once the reassembler realizes that the flow is marked like this, all new arriving packets are dropped without saving in any of the buffers. The only exception is a FIN packet, which initiates flushing of all packets in the buffers before it is dropped.

If the flow was marked as to be dropped after the flow was already closed, the flow remains in the hash table. Either a FIN packet is retransmitted and so the flow is removed, or the flow must be removed externally. This is not supported at the moment.

6.4 TCP processing

The internals of the DFA implementation are more or less the same for both, packet and TCP processing. It always consumes a single packet. Only the TCP processor skips packet headers and inspects only the TCP payload. The major difference is in the supporting routines that provide the processing core with a next byte and with a next packet. The problem is simple in case of a packet processor. There is a ring of received packets and whenever a Ruler thread is idle, it picks one packet and proceeds. In case of a tcp processor, the things are more complicated. We cannot let each thread process arbitrary packets, we must keep the state that was reached when the previous packet of the stream was consumed till we start processing the next one. Therefore it is also not possible to process packets that belong to the same stream by different threads concurrently.

In our solution, we have one more levels of indirection before Ruler gets hold of a packet. The TCP reassembler schedules a TCP flow in an SRAM ring when there are unprocessed data available in its buffer. Ruler, instead of getting a single packet, gets a whole TCP stream. This TCP stream is assigned only to this particular thread and

removed from the scheduled-queue so that it cannot be acquired by another thread. The stream is marked as `ASSIGNED` and is not scheduled by the TCP reassembler anymore, unless its consumer releases the stream. The assigned thread then processes the TCP packet buffer as long as there are pending data.

Since only a single thread is processing a flow at any moment, we can easily keep the last reached state in a register while a new packet is being retrieved from the packet buffer. The more troublesome issue is when a TCP buffer is exhausted. On a fairly used gigabit link there are many more streams waiting to be processed. Therefore we must postpone processing of the current stream since it would make the thread idle and precious cycles would be wasted. We save the state of the current stream to its descriptor and pick another waiting stream. Once there are data for the postponed one, it is put on the scheduled-queue again. In such a design, we keep all available threads busy. As a result, one flow is rarely processed only by a single thread, but it migrates not only among threads, but also among MEs.

Saving a state of a stream is not that difficult. We postpone its processing if and only if an entire packet is consumed. Therefore we do not need to keep any position in the stream, we will continue with the next available packet. The sole information required for resuming processing of a stream is the next state of the DFA that should be visited. We identify the state either by the instruction pointer where its execution starts or by its id in SRAM memory table. IXP architecture has space in its instruction store for at most 8k of instructions. We need only 13 bits to encode the return instruction and a state in the memory table is encoded in 16 bits. There is enough space to store this and some additional information in the SRAM TCP flow descriptor.

The current state is saved when the routine to obtain a new TCP buffer (packet) is called. In this function we already need the full return information and thus the only action needed is to save this in the flow descriptor when the stream is being released.

When inspection of a stream is resumed, we load the return register with the saved instruction pointer. After all actions that are necessary for full recovery of the stream processing are finished, we jump to the return address, i.e. to the saved DFA-state and Ruler continues as if the TCP stream processing was never interrupted.

Also when a stream was already *accepted* and an accept value was assigned, we need to store this value. We do not inspect packets of an accepted stream and we just assign the accept value to each packet and pass it to the next stage (transmitter).

Chapter 7

Performance of Ruler packet filter

We carried out several performance tests to determine what is the top achievable throughput of our Ruler implementation on IXP2400. First of all, we were interested whether it is possible to reach one gigabit. Secondly, whether it is also possible if we inspect all received bytes.

In contrast to the evaluation of the extended FFPF (Chap. 4), getting hold of a packet is not such an expensive operation. We do not experience cache thrashing, simply because there is no cache. Therefore the number of cycles needed by an ME to retrieve a packet and start its processing is constant and moreover, it is fully determined by the instructions and memory reads that we must execute. An exception is if only a single thread is running on every ME and so threads cannot interleave their execution to hide the memory latency. In this case, we have to include also delays caused by memory reads in the cost of getting a packet reference from a scratch ring. Otherwise, we can account only the cycles really spent by executing instructions. As a result, the number of packets we can process is interesting only if the number of inspected bytes per packet is also constant. Otherwise, if the the number of inspected bytes is determined by the packet size, it means that potentially all bytes can be checked. In this case, we are interested in how many bytes we can scan.

It is hard to figure out what the theoretical throughput is, since the average number of cycles spent per byte varies for each input set of Ruler rules. Considering that the cheapest inspection state consumes 18 cycles, the ceiling for the number of bytes inspected by a single 600MHz ME is 266Mbps. However, the experiments show that the real throughput is slightly lower. First, it is because of the constant overhead per packet, second, it is because of the high memory usage. Table Tab. 7.1 shows what bitrate a single ME is able to process for various packet sizes. In this experiment we used a simple DFA with states that only request next bytes and the first branch rejects the byte. This is equal to inspecting all bytes by the simplest possible states. The traffic was generated by iperf [25] running on several machines. Iperf can generate UDP traffic similar to audio stream. The bitrate and the packet size is configurable.

Another experiment proved that the processing power scales linearly with the number of used MEs. The same table also shows that the number of bytes processed for each packet size is dropping as the packets become smaller. This is because the overhead to get a packet comparing to the number of cycles needed to inspect the packet is more signifi-

packet size (bytes)	measured bitrate on 1 ME (Mbps)	theoretical throughput on 6 MEs (Mbps)
1470	251	1506
1024	247	1482
512	238	1428
256	220	1320
128	191	1146
64	152	912

Table 7.1: Measured maximal throughput of the Ruler packet filter for different packet sizes when all bytes were inspected

cant and less cycles can be used on the packet inspection itself. Also scratch memory is becoming overloaded as new packets are saved and requested more frequently. Another aspect that has impact on the measured performance is that the traffic generator is not sending packets at a constant rate, but rather in bursts with bitrates significantly higher than what an ME can consume. Therefore the overall dropping is also higher. Essentially, the real throughput is higher than what we measured. In addition, more MEs reduce impact of bursts. Our other experiments support this. In spite of the fact that processing a stream of minimal size packets cannot be achieved on full gigabit, it shows that we can go beyond since it is not the usual average case.

The measurements just described assume that all bytes of every packet are checked, there is no match and the byte is skipped immediately and next one is requested. It depends on the user whether this is a valid assumption or not. This measurement only shows the upper bound which is reachable. The more interesting lower bound is completely dependent on the complexity of the rules supplied by the user. Therefore the user must take this into account. The user must use filtering expressions that can determine as soon as possible which rule matches. This helps the overall performance tremendously.

As already mentioned, the advantage of the in-memory states is that the execution time per byte is constant. Another big benefit is much more fair scheduling of threads on an ME, and so better multithreaded performance. Therefore we do not have to make such assumptions as in the experiments described before. Given 6 MEs the theoretical throughput with the largest ethernet packets when executing only in-memory states is about 900Mbps on the IXP2400.

The most usual format of packet matching expressions first classifies packets according to their headers. There are only a few interesting positions in most headers with only a limited number of values. This can be very efficiently matched by a DFA encoded in instructions. However, inspecting the payload creates states with a high number of possible transitions and the 32 cycles spent in in-memory states can be easily exceeded. Therefore to get the optimal performance in the worst case we suggest to combine both methods.

Our experiments also show that the IXP is able to copy data to the host at the link rate if the hosting hardware has the required throughput. This means that it should preferably have a 64bit/66MHz PCI bus which is not overloaded by other PCI devices. And of course, the host operating system together with the host CPU must be able to process this amount of data. Since copying packets to the host is handled by the transmition engine only after packets were classified, dropping a packet due to the overloaded host by the transmition ME means that all cycles spent on inspection of this packet were wasted. In

contrary, if the ruler is overloaded, packets are dropped already by the receiving ME and the loss of cycles is insignificant.

One of the tasks of Ruler is to anonymize packets, i.e. rewrite them. First of all, the performance is harmed already by the single threaded nature of the rewriting code. It is worth to mention the the new IXP2350 has more local memory which allows us to enable 2 threads per ME in our next release. However, we experienced a very good performance when zeroing IP addresses of all IP packets in our experiments. A single ME was able to delete IP addresses from all packet at about 540Mbps. Therefore only 2 MEs are sufficient for full gigabit. We see the main reason for this performance in the simple DFA which filters only IP version 4 packets. This filter takes only a few cycles per packets. If more complex filtering prior to packet rewriting must be performed, executing the DFA becomes the bottleneck of the packet anonymization. Even this simple scenario gives a promising result for using a IXP2350 with only 4 MEs to be used as an anonymization preprocessor for monitoring e.g., a campus backbone. Besides that the IXP2350 MEs have more local memory, we will also benefit from 900MHz comparing to 600MHz of the IXP2400 which we use in our testbed.

Chapter 8

Related work

Ideas used in FFPF can be found in other networking projects too. The European SCAMPI [26] project also divides processing in different levels, userspace, kernelspace and hardware. SCAMPI uses DAG and COMBO6 cards, both of which are FPGA based PCI boards. These NICs are configurable and programmable, but most of the logic remains on the host. Unlike FFPF, SCAMPI uses only a linear chain of filters. This chain is built by the MAPI (monitoring API) developed for this project. Also the userspace implementation is different. FFPF is using a statically linked library, while SCAMPI applications communicate with a single MAPI-daemon. This daemon is the entry point to the lower levels (kernel, devices). Our group already started porting the MAPI to the FFPF framework and it will soon be possible to use SCAMPI monitor with FFPF without any change.

The SILK project [27] (SCOUT [28] paths in the Linux kernel) uses an idea of different flows. It divides packets which belong to different applications, streams and protocols into separate flows. These are processed in various "paths" in the Linux kernel. Each path is a chain of processing steps. SILK uses its own scheduling policy and each path runs in a separate SILK-thread, in order to be able to perform customized scheduling and prioritize some of the threads.. In contrast, FFPF flowgroups are based more on various application requirements, protection against conflicts and security levels, than on different protocols. One FFPF application can process more SILK-like flows in a single flowgroup. FFPF copies packets for each flowgroup. Like FFPF, the dropping in SILK is done as soon as possible, before too many time was spent on the processing.

The CLICK project [29] uses a very flexible graph architecture. Unlike FFPF, it allows oriented cycles. Each node has a number of ports with different meanings. It depends on packet classification to which port it is sent. In FFPF, packets are passed to all connections. There are two types of ports in CLICK, pull and push. It is not possible to connect an input and an output port of a different kind, therefore the user has to keep this in mind. The push-ports send packets immediately to the next node. In contrast, the pull-ports enable the successor to ask for data when it is possible to process them. Nodes with the push-input ports and the pull-output ports act as queues or buffers. Buffering inside FFPF is implicit and transparent to the user. Polling on *IBuf*s is similar to the pull-inputs in the CLICK sense. Another similarity is FFPF polling and CLICK scheduling. CLICK has a queue of tasks. After a dequeued task is processed, dependent tasks are executed automatically. One task in a queue can be compared to an unread packet in the

FFPF buffer. Retrieving a packet also starts its processing in the connected grabbers. We can understand processing in a grabber as a task. Unlike FFPF, CLICK was designed as a kernel-based router. FFPF works on network processors, kernel and userspace and targets monitoring applications.

Among the most widely used network filters are Linux kernel's *iptables* [4]. This subsystem might serve many tasks. Its filtering abilities are used for instance for firewalling. The iptables based firewall uses a sets of rules, known as chains. Every rule says what should happen with the current packet. Targets, so-called jumps, start processing the packet in another chain. Other targets like ACCEPT, DROP, REJECT stop the packet processing and does not return the packet to the superior chain for further examination. On the other hand the target RETURN returns from the current chain to the superior one. In this case or if the packet did not match any of rules, processing continues on the next rule behind the jump. An administrator can use this to build a fairly complex filtering graph. Also a large variety of tools was developed to make the work with the iptables more comfortable. Iptables are used for filtering on the basis of IP, TCP and UDP header. Matching is not used to inspect the payload of the packets. Iptables are closely connected to the *netfilter* [4] layer. Therefore it cannot use processing on a special hardware. Packets have to go through the device driver before they reach the filter.

The closest project to FFPF is described in "Extensible Routers for Active Networks" [30], which the IXP1200 as a network processor. Processing is also divided between micro-engines, StrongARM (XScale for IXP2xxx) and the host. Micro-engines are used mainly for receiving and transmitting and distributing packets into queues for processing on upper levels rather then for deeper processing itself. Processing on the StrongARM is very dependent on the packet size. With minimum sized packets, StrongARM has no cycles left for processing and is used only as a forwarder to the host. On the other hand the processing of maximum sized packets is possible. There is a very lightweight kernel running on top of the StrongARM core. FFPF uses the XScale core mainly for processing and forwarding is only an option. There are stand-alone IXP based devices which do not have any connection to a host machine and here the XScale does all complex processing which cannot be done in the micro-code. In contrast to FFPF, the micro-engines in [30] are fully excluded from the configuration (code injection). The FFPF user can upload new code to the MEs, start and stop them at any time as proved by Mihai Cristea [31], who successfully used the FFPF to manage code on the IXP2850 in in a Token Based Swith project.

After the NP emerged, and since the IXP architecture is one of the most popular among them, there are many projects that are trying to offload different task from the operating system networking stack to an NP. [32] presents how to reduce the Linux TCP stack to less then 10% of its code by offloading TCP processing to an IXP chip. Similar to FFPF, motivation of other projects that are using IXPs is to avoid memory copying and context switching between kernel and userspace. SpliceNP[33] exploits IXP architecture for implementing a content-aware switch. They stress that the XScale core is too slow for intercepting TCP streams and therefore all processing is implemented only on the MEs. The troughput of HTTP sessions was increased dramatically in comarison to a Linux splicer, however the set of features they have implemented is wide, but limited. At the moment, there is only one implementation of the TCP for the IXP2xxx, we are aware of.

Since it is a comercial project, we are not able to say anything more specific about it.

It is worth to mention that thanks to Willem de Bruijn the FFPF framework evolved from a pure packet processor to a stream processor called Streamline and it can use the Ruler on IXP as a preprocessing filter. In addition, the IXP Ruler is integrated to the SCAMPI's MAPI as a part of its successor LOBSTER [34].

Chapter 9

Conclusions

This thesis presents a full integration of a network processor into a highly extensible and flexible network monitoring and packet filtering tool. The implementation was explained in details. We focused on redesigning the previous implementation in such a way that it fully covers the IXP based PCI card. The new design is general enough to enable easy integration of other network interface cards as well as stand-alone devices into one monitoring system.

Our implementation shows that such a complex device, can be hidden to the user of the system and still even an unexperienced user can benefit from most of the features without any special knowledge about programmable network devices. The syntax of our input expressions gives a simple way of creating a flowgraph on the command line. Moreover a GUI is also provided. The FFPF API is simple enough to be understandable by an unskilled programmer.

We proved that it is possible to implement such a framework where a packet does not have to be copied during its traversal of the whole flowspace hierarchy. This saves a lot of computing time. The hierarchy consist of levels with different processing speeds, on the host machine as well as on the network device. In contrast, there are situations where moving a packet to another location can be very useful. The most important finding is that this can be done transparently to the user and at high speed without use of the host CPU and with significant offload of the NPU. Offloading the host machine in that case is significant which was one of the main goals of this project.

Our tests showed that processing a large number of small packets is the main problem as for any other packet processing framework as getting hold of the packet is the most expensive action in the data-path. The number of packets that can be processed is more important than the bitrate itself. We proved that the idea of pushing the packet processing to the lowest levels enhances the performance. The MEs has the ability to filter out most of the uninteresting traffic and by that to reduce overloading of the successive stages and to enable filtering with only a limited or even no dropping on a high speed.

The high degree of extensibility and flexibility of the FFPF framework faces two problems. The first is caused by the Intel SDK for IXP, which does not enable us to implement as wide a functionality of the micro-engine manager as we wish. The second and more important is that the complexity and commonness of the framework has its own overhead which is difficult to prune.

In order to support our claims that filtering traffic as close to the wire as possible can offload the host machine dramatically, we implemented a packet filter based on the Ruler language on the IXP micro-engines. Clearly, there is more suitable hardware for such pattern matching (e.g., FPGAs), although we proved that IXP chips can be used very efficiently for a certain class of filtering tasks. Moreover they can be successfully used for online packet anonymization on the link rate to remove private and sensitive information before data are handed to other parties for further processing. Without this, organizations are reluctant to let others to monitor their networks. However, without understanding Internet traffic, it is not possible to effectively improve services and deal with threats that the community is facing.

We implemented several tests to measure FFPF performance in various scenarios. We tested reading data over the PCI bus from the host as well as copying packets directly to host memory. The latter proved to be better overall even though it needs more support in the control part and the data-path is more complex, it gives promising results for future work as not all the options in this direction were exhausted. We also carried on performance measurement of the Ruler implementation and showed that we can achieve full gigabit with for real world scenarios. Naturally, it is possible to create examples that reach limits of the IXP hardware. Nevertheless the processing speed of such complicated tasks still reaches several hundreds of megabits per second.

We believe that our work will contribute to the further development of packet monitoring and to enable easier use of IXP based devices by the wider public.

Bibliography

[1] Intel©IXP2XXX Product Line of Network Processors. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[2] Intel©XScale Technology. http://www.intel.com/design/intelxscale/.

[3] Herbert Bos, Willem de Bruijn, Mihai Cristea, Trung Nguyen, and Georgios Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, San Francisco, CA, December 2004.

[4] The netfilter/iptables project. http://www.netfilter.org/.

[5] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*, pages 259–270, 1993.

[6] Mihai Cristea, Willem de Bruijn, and Herbert Bos. Fpl-3: towards language support for distributed packet processing. In *Proceedings of IFIP Networking'05*, Waterloo, Ontario, Canada, May 2005.

[7] Mihai Cristea and Herbert Bos. A compiler for packet filters. In *ASCI2004*, 2004.

[8] Mihai Lucian Cristea, Claudiu Zissulescu, Ed Deprettere, and Herbert Bos. FPL-3E: towards language support for reconfigurable packet processing. In *Proceedings of SAMOS V: Embedded Computer Systems: Architectures, MOdeling, and Simulation*, Samos, Greece, July 2005.

[9] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

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

[11] The LEMON Parser Generator. http://www.hwaci.com/sw/lemon/.

[12] Intel©21555 Non-transparent PCI-to-PCI bridge. http://www.intel.com/design/bridge/21555.htm.

[13] Bruce Perens. *Understanding the Linux Virtual Memory Management*. http://http://www.phptr.com/perens/.

[14] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, ISBN: 0-596-00590-3.

[15] Salim J., Olsson R., and A. Kuznetsov. Beyond Softnet. In *5th Annual Linux Showcase and Conference*, pages 165–172, November 2001.

[16] Erik J. Johnson and Aaron Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, ISBN:097178616X.

[17] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 237–248, New York, NY, USA, 2005. ACM Press.

[18] Paul E. McKenney. Using RCU in the Linux 2.5 kernel. *Linux Journal*, 1(114):18–26, October 2003.

[19] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev. An Intel IXP1200-based network interface, 2003.

[20] Kees van Reeuwijk and Herbert Bos. Ruler: a high-level language for traffic classification and rewriting using tagged DFA. Technical Report IR-CS-027, Vrije Universiteit Amsterdam, The Netherlands, 2006.

[21] Willem de Bruijn, Asia Slowinska, Kees van Reeuwijk, Tomas Hruby, Li Xu, and Herbert Bos. SafeCard: a Gigabit IPS on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, Hamburg, Germany, September 2006.

[22] Snort - the defacto standard for intrusion detection/prevention. http://www.snort.org/.

[23] Jason Gait. Hash table methods for case statements. In *ACM-SE 20: Proceedings of the 20th annual Southeast regional conference*, pages 211–216, New York, NY, USA, 1982. ACM Press.

[24] Ulfar Erlingsson, Mukkai S. Krishnamoorthy, and T. V. Raman. Efficient Multiway Radix Search Trees. *Information Processing Letters*, 60(3):115–120, 1996.

[25] Iperf - The TCP/UDP Bandwidth Measurement Tool. http://dast.nlanr.net/Projects/Iperf/.

[26] SCAMPI - A Scaleable Monitoring Platform for the Internet. http://www.ist-scampi.org/.

[27] Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, and Per Gunningberg. SILK: Scout paths in the Linux kernel. Technical Report 2002–009, Dept. of Information Technology, Uppsala University, 2002.

[28] David Mosberger-Tang. Scout: A path-based operating system. Technical report, Tucson, AZ, USA, 1997.

[29] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 217–231, New York, NY, USA, 1999. ACM Press.

[30] Aki Nakao. Extensible routers for active networks. In *DANCE '02: Proceedings of the 2002 DARPA Active Networks Conference and Exposition*, page 92, Washington, DC, USA, 2002. IEEE Computer Society.

[31] Mihai-Lucian Cristea, Li Xu, Leon Gommans, and Herbert Bos. Efficiency and performance issues in token based switch systems. In *Twelfth Annual Conference of Advanced School for Computing and Imaging ASCI*, Lommel, Belgium, June 2006.

[32] Ralf Lehmann and Alexander Schill. Linux tcp network stack analysis and partitioning for network processors. In *WISICT '05: Proceedings of the 4th international symposium on Information and communication technologies*, pages 69–74. Trinity College Dublin, 2005.

[33] Li Zhao, Yan Luo, Laxmi Bhuyan, and Ravi Iyer. SpliceNP: a TCP splicer using a network processor. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, pages 135–143, New York, NY, USA, 2005. ACM Press.

[34] LOBSTER - Large-scale Monitoring of Broadband Internet Infrastructures. www.ist-lobster.org.