A Wire-speed Packet Classification and Capture Module for NetFPGA

Malcolm Scott
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge CB3 0FD, UK
Malcolm.Scott@cl.cam.ac.uk

ABSTRACT
Hardware-based packet classification and capture can be a useful feature for a high-speed networked device, or a useful debugging aid for NetFPGA projects. This paper presents the design and implementation details of a drop-in module for the NetFPGA framework which provides a simple but nevertheless highly flexible system for matching patterns in one or more packet headers and/or payloads and diverting such packets to the host system via DMA for inspection or recording. The module is implemented in such a way as to never act as a bottleneck to the NetFPGA pipeline, and can classify packets at wire speed. We also present an extensible software framework which allows filters to be specified and implemented by the user in a simple manner according to built-in knowledge of common protocols, provides display of captured packets via the Wireshark protocol analyser and optionally further distributes captured packets to custom processes via a publish/subscribe system for analysis and/or storage. The hardware module and associated software will be available to the NetFPGA community under a free license.

Categories and Subject Descriptors
C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network Monitoring*

General Terms
Measurement, Performance

1. INTRODUCTION
It is frequently required for a network operator to be able to inspect a subset of the packets passing through a network. This can be needed for a wide variety of different reasons, such as

- fault diagnosis,
- fault detection,
- investigation of network abuse,
- capacity planning and monitoring,
- usage logging

and many others. Traditionally, this used to be achieved by configuring a switch or router to mirror all traffic to a monitoring server, which performed packet classification of the complete feed of all packets in software against criteria configured by the network operator—but with exponentially-increasing transmission speeds it is increasingly necessary to perform packet classification in hardware so that only the packets of interest are handled in software.

There are a number of commercial, high-speed, hardware-based packet classification and capture systems on the market such as those manufactured by Endace [1]. The aim of this paper and the module it describes is not to compete with such systems, but rather to design and implement a small yet flexible self-contained module for performing packet classification and capture as part of a larger NetFPGA project—for example a router or switch. This could be in order to add an extra feature for one of the reasons described above, or it could be as a debugging aid during the development of a NetFPGA-based project.

Our module can be inserted into the user data path of a project which follows the recommended NetFPGA architecture, likely with no or minimal modification required, and provides a register interface for configuration of per-port filters. Captured packets are sent via DMA to the software, specially marked such that they can easily be separated from other exception or control packets, and can be displayed by standard tools or processed by simple scripts.

In addition, we have implemented a software configuration tool to aid the user in compiling filters from simple, human-readable specifications broadly similar to those used by the familiar software-based packet capture tool TCPdump [3].

2. DESIGN DECISIONS
There is considerable variation in what might be required or expected in a packet classifier. In the simplest case, a classifier might allow only header fields of a particular set of predefined protocols to be inspected when considering whether a given packet should match the classifier, and may only allow one classification filter to be active at a time. More

First European NetFPGA Developers' Workshop
2010, Cambridge, UK

advanced classifiers may allow multiple filters to be installed, and/or allow matching of user-defined protocols or arbitrary packet payload data.

The approach chosen in this case was a compromise between hardware complexity and feature breadth. Keeping the hardware simple was an important requirement as this module is intended to be used as part of a larger design and must avoid taking up too much space on the FPGA. The following requirements were drawn up:

- **Optional per-physical-port filters.** As standard, each MAC interface can have filters configured independently; packet classification is applied on packet ingress according to the filters configured on the ingress port. Filtering can also be added to DMA interfaces at the cost of FPGA area. Alternatively, if required in order to save more space, one filter can be shared between ports thus excluding the per-port filter capability.
- **Multiple filters can be combined such that all must match.** Multiple header patterns can be combined into a single filter such that the packet must match every specified pattern in order to satisfy the filter as a whole. No complex boolean expressions involving multiple filters are permitted—in particular there is no boolean 'OR' supported, only 'AND'—as this would increase hardware complexity considerably.
- **Matching of arbitrary protocol headers.** The set of protocols supported is not hardwired into the hardware and can be updated from software. Matching of headers of any protocol whatsoever is supported, provided that the headers are contained within the first 64 bytes of the packet (counting from the start of the Ethernet header) and the headers are at fixed positions within the packet (variable-length headers are not supported; see below). If 64 bytes proves to be insufficient, the packet capture range can be increased without too much difficulty, although this would take up a greater amount of the available FPGA area.
- **Matching of data at the start of protocol payload.** As a generalisation of the previous point, no distinction is made between protocol headers and payload, and as a result filters can apply string matching to the first few bytes of packet payloads in order, for example, to identify HTTP GET requests and similar simple protocols. However, strings matched must appear at a predictable byte offset into the packet, which once again precludes use of this feature where variable-length headers are present. This is discussed in more detail below.
- **Marking of captured packets by inserting a special Ethernet address.** Software can inspect the Ethernet header of the packet in order to distinguish captured packets from other exception packets which may be sent to the DMA interfaces by other modules. This does mean that the original Ethernet destination of the packet is lost, but we considered that in general higher-layer protocol headers such as IP are of more interest and this was the least intrusive method of marking a packet.

Furthermore, was considered important that this module would not adversely affect performance of the overall hardware system. The module is intended to sit in a simple pipeline in series with other NetFPGA user data path modules and as with any such module is in a position where it could cause a performance bottleneck if it behaved in certain ways. As a result we implemented the module in such a way as to never cause such a bottleneck. In particular, the packet classification and capture module is:

- **Fully pipelined.** The module by necessity buffers packet data for a short while, in a FIFO, in order to allow inspection of up to 64 bytes of packet data before determining the packet's destiny but is capable of having multiple packets in motion at the same time—i.e. the tail end of one packet and the start of the next—without interruption.
- **Low latency.** Packet data passes through the pipeline in eight-byte words, preceded by a control header at the start of the packet, so nine clock cycles elapse during the arrival of the required 64 bytes of packet data. One further clock cycle after the required data has arrived, packet data can be passed on to the next module, and so 80 nanoseconds of latency is introduced by this module.
- **Wire-speed.** The module is not capable of initiating a pipeline stall, and thus will not cause a throughput bottleneck. Data entering the FIFO buffer is matched asynchronously within one clock cycle, and the packet header rewriting stage at the output of the module also completes within one clock cycle. Flow through the FIFO is controlled entirely by inputs from neighbouring modules.

2.1 Variable-length Headers and Payload

Extending the packet classification hardware design to support variable-length protocol headers would introduce a considerable extra complexity and cause a corresponding FPGA area penalty at the very least—and would likely also require a relaxation of the latency or throughput requirements given above due to the difficulty of performing more complex matches within a single 8-nanosecond clock period.

Unfortunately, however, some of the most common protocols such as IP and TCP include the facility for optional additional headers. Of these, IP options would be the most problematic as they would introduce an unpredictable offset to all higher-layer header fields, but thankfully they are seen rarely in the wild due to the prevalence of routers which do not permit them [2].

TCP options are seen in the wild frequently, however; Linux for example will usually include the optional TCP timestamp header on every packet unless this has been specifically disabled. Since options are placed after mandatory header fields, this does not impede in matching other fields, but it does cause the start of the payload to become unpredictable, or in some cases it can extend the headers beyond the 64-byte matching window supported by this packet classifier in its current form. TCP timestamps can usually be disabled—on Linux, for example, one can run (as root) the command "`sysctl net/ipv4/tcp_timestamps=0`"—and this may prove necessary if hardware-based payload matching is absolutely required. However due to the impracticality of disabling TCP timestamps and any other options on every endpoint it is likely that the payload matching feature cannot be relied upon in the general case in its current form.

Nevertheless this does not preclude the utility of the packet classifier for TCP traffic. Everything but the payload can be reliably matched in hardware, and any further payload matching could be done with software cooperation. For example, in order to match HTTP GET requests, the hardware could capture TCP packets destined for port 80—presumably a tiny subset of the total traffic through the hardware—and the software could inspect the payloads for the GET query taking into account the length of the TCP header.

3. HARDWARE DESIGN

The self-contained packet classification and capture hardware module, named "`packet_capture`", is intended to sit in the user data path provided by the NetFPGA framework. A single instance of the module implements a single set of filters (either for a single header pattern or a combination of header patterns which must all match for a packet to be captured). It is primarily intended to be instantiated once per physical port, between and in series with the MAC receive queue and the input arbiter, in order to achieve per-port filtering. However if FPGA space is at a premium the `packet_capture` module can be instantiated a single time, after the input arbiter module (thus reducing the footprint by 75%) in order to forego per-port filters.

Since the `packet_capture` module implements both the filter management and packet classification logic it also requires participation in the PCI register bus for filter configuration purposes.

The filter itself can be considered to be stored in two 8×64 bit, dual-port RAMs per module instantiation, called `filter_data` and `filter_mask`. In practice, although these are behaviourally specified as RAMs, due to their small size they are invariably automatically optimised during synthesis into LUTs for increased performance. `filter_data` represents the data to search for in each of the first eight data words (i.e. the first 64 bytes of packet data after the standard module header prepended onto the packet by the NetFPGA framework); the relevant word from `filter_mask` is first ANDed with the packet data before comparison with the relevant `filter_data` word.

This is best illustrated through an example. Consider a filter which aims just to match an Ethertype of 0x0800, ignoring all other header fields. The Ethertype field appears in bits 16 through 31 of the second word. Thus this match is specified by the two variables

```
filter_data[1] = 64'h0000_0000_ffff_0000;
filter_mask[1] = 64'h0000_0000_0800_0000;
```

(with the remainder of `filter_data` and `filter_mask` initialised to all-zeros).

The task of computing suitable values for `filter_data` and `filter_mask` is left to the software. This at first glance may appear to be a trade-off, but actually this approach has several benefits and few disadvantages. Leaving protocol-specific knowledge out of the hardware allows the packet classification engine to be futureproof: adding support for a new protocol need only be done in software, and does not require taking the hardware offline. It also keeps the hardware simple, and thus more reliable as well as smaller in terms of FPGA footprint. Complex filters can be compiled by the software and uploaded to the hardware as a simple set of 64-bit values.

3.1 Configuration Interface

The hardware receives raw `filter_data` and `filter_mask` values from the software via PCI register writes.

An XML register map is provided, suitable for use with the register system introduced in version 2 of the NetFPGA framework. The register interface provided is simple; a filter is uploaded via a table interface which will be familiar to users of the standard NetFPGA framework modules' register interfaces. The register map compiles to the following set of registers:

```
FILTER_TABLE_ENTRY_DATA_HI
FILTER_TABLE_ENTRY_DATA_LO
FILTER_TABLE_ENTRY_MASK_HI
FILTER_TABLE_ENTRY_MASK_LO
FILTER_TABLE_WR_ADDR
FILTER_TABLE_RD_ADDR
```

In order to upload a filter (which, after compilation, is likely represented in the software by a 512-bit data string and a further 512-bit mask), the software must divide the data into 64-bit chunks indexed $i = 0$ through 7, then for some index i, write the top 32 bits of `filter_data[`i`]` to the `FILTER_TABLE_ENTRY_DATA_HI` register, the bottom 32 bits to `FILTER_TABLE_ENTRY_DATA_LO`. The same procedure is used for `filter_mask[`i`]` and `FILTER_TABLE_ENTRY_MASK_HI` and `_LO`. Having performed these four writes, the index i must then be written to the table's write address register, `FILTER_TABLE_WR_ADDR`. This completes the write of a filter word, and the whole procedure of five writes can be repeated for each other value of i.

Reading the filter back into software requires a similar process in reverse: first the address to be read is written to `FILTER_TABLE_RD_ADDR` and then the data and mask at that address can be read back from the appropriate other register.

On power-up, the filter table on the FPGA may contain uninitialised data. It is not cleared during startup, but instead a separate set of flags—`filter_valid[`i`]`—tracks when each table entry is written. When matching packet data, if this flag is not set for a particular filter word, that word is assumed to have a mask of zero such that it will never be considered.

One additional register is provided, separately from the filter table interface: `PORT_NUM_HITS`. This is simply a counter

```
clk ----------------------+
                          V
                  +----------------+
in_wr ----------> |       V        |---------------------------------> out_wr
                  |                |
in_rdy <----------|     FIFO       | <--------------------------------- out_rdy
                  |                |        +------------------+
in_ctrl ---8----> |     10x72      |---8--> |                  |---8--> out_ctrl
                  |                |        |  Header rewrite  |
in_data --64--+-> |                |--64--> |                  |--64--> out_data
              |   +----------------+        |      Enable      |
              |                             +------------------+
              V                                       ^
     +------------------+                             |
     |                  |                             |
     |   Filter check   |------ exception_pkt ------+
     |                  |
     +------------------+
```

Figure 1: Simplified data flow through the packet classifier

which is incremented every time a packet matches the filter and is provided to allow the software to check the number of packets it has received against this counter for verification purposes. It can be reset by writing a value (most usefully 0) to this register.

As with the module itself, the register map defines enough registers for a single filter only. If using per-port filters, the register map must be instantiated four times. The register system provides a facility to do this simply in `project.xml`:

```
<nf:instance name="packet_capture" count="4" />
```

3.2 Data Flow

The flow of data through the module centres around a small FIFO buffer—an instantiation of the NetFPGA library module "`small_fifo`", parameterised to be 72 bits wide (in order to accommodate a 64-bit data word and an 8-bit control word) and 10 stages deep—as summarised in Figure 1. The input of this FIFO is connected directly up to the external interface of `packet_capture`; the next available data and control words from the preceding stage of the user data path (e.g. the MAC) is written directly into the FIFO in each clock cycle so long as the FIFO is not full.

Filter matching also takes place at this stage, by monitoring data as it arrives in the module and is written into the FIFO. A small state machine detects the arrival of a NetFPGA module header and thus the start of a packet, and in a register—`in_word_num`—keeps a count of the number of packet data words so far. The currently-appropriate word of the filter is thus contained in `filter_data[in_word_num]` and `filter_mask[in_word_num]`, and similarly for the current `filter_valid` flag.

Using this data, we can determine whether the filter matches (a "hit") or mismatches (a "miss") at the current word. This problem has by this point been reduced to the pair of combinatorial logic expressions given in Figure 2. Crucially though it is *not* the case that `filter_miss` is the negation of `filter_hit`. A miss means that a filter pattern explicitly did not match; in the case where the mask is zero (or `filter_valid` is unset) there is simply no pattern to match and the filter is ignored giving neither a hit nor a miss.

If when the eighth word is reached there has been at least one filter hit and no filter miss, the packet is considered to have matched the filter and a flag—`exception_pkt`—is set to indicate that the current packet is to have its header and destination port rewritten so that it reaches the software. Soon afterwards (by one clock cycle if the downstream pipeline is flowing freely) the first word of the packet will emerge from the other end of the FIFO and reach the header rewrite state machine. If the `exception_pkt` flag is set, the aforementioned rewriting happens inline here as the data passes from the FIFO to the module outputs (`out_ctrl` and `out_data`).

The exact nature of the rewriting depends on parameters passed to the module, detailed below.

3.3 Module Parameterisation

The header rewriting behaviour of the packet capture module is configurable to an extent. As previously mentioned, captured packets are tagged with a special reserved destination Ethernet address; this defaults to `FF:FF:FF:FF:FF:FE` but can be changed via a (compile-time) module parameter; it is not expected that there would be any need to change this however as this address is not likely to be found in real network traffic.

The destination DMA interface for packets matching the filter can be set during operation: the module provides an input called `exception_port` for this purpose (a 16-bit one-hot-encoded field in the same format as the destination port field in the NetFPGA module header). Additionally a per-port default can be specified via a module parameter, such that if `exception_port` is tied to zero the default is used instead; the intended purpose of this is to have captured packets from each of the four MAC ports sent to the corresponding one of the four DMA interfaces by configuring the default for each module appropriately.

```
assign filter_hit  = in_wr &
                     filter_valid[in_word_num] & |filter_mask[in_word_num] &
                     ((in_data & filter_mask[in_word_num]) == filter_data[in_word_num]);

assign filter_miss = in_wr &
                     filter_valid[in_word_num] & |filter_mask[in_word_num] &
                     ((in_data & filter_mask[in_word_num]) != filter_data[in_word_num]);
```

Figure 2: Combinatorial expressions for determining filter hits and misses at a given word

Additionally, in a per-port filtering setup, the module must be parameterised according to the port it is associated with: in particular the block address of the PCI register block (as generated from the XML register map by the register system) to use must be provided, and must be different for each instance of the `packet_capture` module so that filters remain separate.

4. SOFTWARE DESIGN

The software component contains all of the protocol-specific knowledge needed to compile filters suitable for sending to the hardware. It also contains components to aid the distribution of captured packets to any interested party via a publish/subscribe mechanism.

4.1 Filter Compilation and Installation

A utility called **pacap_filter** is responsible for compiling a human-readable packet filter specification into the filter data and filter mask values required by the hardware. As a demonstration of the flexibility of this application, a sample invocation could look like

```
pacap_filter 2,3                          \
    ip.dest=131.111.179.82                \
    tcp.dport=80 tcp.data=GET
```

meaning that ports 2 and 3 will be monitored for TCP packets directed to 131.111.179.82 port 80, whose payloads begin with the string "GET". (All clauses must match simultaneously, i.e. are ANDed together.) This filter specification, in fact, implies a few other criteria which are applied automatically—for example that the Ethertype must be 0x0800, so that non-IP packets are not matched accidentally. The process which pacap_filter undertakes in order to apply this filter can be seen in the sample output from the application shown in Figure 3. This process completes in a fraction of a second.

The pacap_filter utility is written in Python, and mostly consists of a table of protocols and header fields, along with metadata on the way in which protocols stack (so that, for example, referring to TCP in the filter causes checks to be added on various IP header fields, which in turn causes an Ethertype check, as expanded above). At the time of writing, pacap_filter understands Ethernet, IPv4, ICMP, UDP, TCP and OSPF; more protocols are expected to be added as they are required.

```
== Expanded filter =====================
ip:
        dest: 0x836FB352
        ver: 0x4
        hdrlen: 0x5
        protocol: 0x6
ether:
        type: 0x800
tcp:
        dport: 0x50
        data: 0x474554

== Compiled filter =====================
DATA 0000000000000000 0000000008004500 0000000000000006
     000000000000836F B352000000500000 0000000000000000
     0000000000004745 5400000000000000
MASK 0000000000000000 00000000FFFFFF00 00000000000000FF
     000000000000FFFF FFFF0000FFFF0000 0000000000000000
     000000000000FFFF FF00000000000000

Erasing packet capture filters on port 0
Erasing packet capture filters on port 1
Uploading filters to packet capture hardware on port 2
Uploading filters to packet capture hardware on port 3
Done.

== Running capture =====================
```

Figure 3: Sample output from pacap_filter

4.2 Captured Packet Distribution

control andThe existing software component of the project which integrates this `packet_capture` module—for example, in the case of the basic router we used to prototype the module, the control and exception-packet-handling software—must be modified slightly in order to take account of the specially-tagged captured packets which will be received on the DMA interfaces. The required change is minimal, however: all it must do is check the destination MAC address on every received packet, and if it matches the tag address (`FF:FF:FF:FF:FF:FE` or as configured in the hardware) it should be forwarded to a UNIX socket.

The UNIX socket is provided by **pacap_daemon**, a publish/subscribe moderator which distributes packets to any connected client. Clients can exist to process and/or store packets in any way; this stage is left to the user. Writing a client is a simple process in almost any UNIX scripting or programming language, as the required UNIX socket API is almost ubiquitous.

A sample client is provided with the packet capture system as it stands. This is built into pacap_filter, and copies received packets into the Wireshark network protocol analyser [4] for user-friendly display and manipulation (via a dummy network interface, as provided by the Linux kernel 'dummy' module, since Wireshark is unable to read from a UNIX socket). As it stands, therefore, pacap_filter is suitable for use by an end-user who does not need to know the detail of how packets are being captured; the software operates in a similar manner to TCPdump or Wireshark but with the benefit of hardware acceleration.

5. CONCLUSION

We have implemented a capable yet compact packet capture and classification module suitable for insertion into a NetFPGA project, and able to cooperate with the other modules that make up the project, allowing for the addition of packet capture capabilities to almost any networking hardware implementable on the NetFPGA—or simply as a handy debugging mechanism for NetFPGA projects which allows the developer to see in real time the packets passing through the hardware.

The module works effectively in practice, despite a few design trade-offs having been made in order to keep the FPGA area used small. Filters are compiled and deployed from a simple, human-readable filter specification language using an easily-extensible software framework, and packets can be easily distributed in software for custom analysis.

The Verilog and XML source of the hardware module, together with the Python source of the software components, will be made available under a free license to the NetFPGA community.

6. ACKNOWLEDGEMENTS

I am very grateful to my colleagues Sarah Fortune and Sebastian Kaldonek who worked tirelessly to produce the software components of the router on which this packet capture system was developed.

7. REFERENCES

[1] Endace Limited. 100% Packet Capture: Hardware. Online; available at `http://www.endace.com/high-speed-packet-capture-hardware.html`.

[2] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. Ip options are not an option. Technical report, 2005.

[3] LBNL Network Research Group and TCPdump Maintainers. TCPdump and LibPcap. Online; available at `http://www.tcpdump.org/`.

[4] The Wireshark Foundation. Wireshark – go deep. Online; available at `http://www.wireshark.org/`.