Transparent acceleration of software packet forwarding using netmap

Luigi Rizzo, Marta Carbone, Gaetano Catalli
Dip. di Ingegneria dell'Informazione
Università di Pisa, Italy
Email: rizzo@iet.unipi.it

Abstract—**Software packet forwarding has been used for a long time in general purpose operating systems. While interesting for prototyping or on slow links, it is not considered a viable solution at very high packet rates, where various sources of overhead (particularly, the packet I/O mechanisms) get in the way of achieving good performance.**

Having recently developed a novel framework for packet I/O on general purpose operating systems, we have investigated how much the removal of this bottlenecks can improve the performance of software packet processing. The problem is of interest because software switches/routers are widely used, and they are becoming inadequate with the increasing use of 1..10 Gbit/s links.

The two case studies (OpenvSwitch and Click userspace) that we report in this paper give very interesting answers and insights. First of all, improving the I/O layer has the potential for huge benefits: we managed to bring the performance of OpenvSwitch from 780 Kpps to almost 3 Mpps, and that of Click userspace from 490 Kpps to 3.95 Mpps, by simply replacing the I/O library (libpcap) with our accelerated version.

On the other hand, reaching these speedups was not purely mechanical. The original versions of the two systems had other limitations, partly hidden by the slow packet I/O library, which prevented or limited the exploitation of these speed gains.

In the paper we make the following contributions: i) present an accelerated version of libpcap which has the potential of significant speedups for many existing packet processing applications; ii) show how we modified two representative packet processing applications (in particular, Click userspace), achieving huge performance improvements; iii) prove that existing software packet processing systems can be made adequate for high speed links, provided we are careful in removing other bottlenecks not related to packet I/O.

I. Introduction

From their inception, computer networks have used software packet forwarding nodes to extend their reach and to experiment with new protocols and mechanisms. Many of these prototypes have been developed on general purpose Operating Systems, which provide a comfortable environment for experimentation. At times, the level of performance and maturity reached by such software has promoted their use in production systems.

In absolute terms, however, the performance of software packet processing, especially on general purpose OS, is often one or two orders of magnitude below that of hardware equivalents. Not rarely, there same goes for equipment cost.

Packet I/O (i.e., the task of moving data packets between the network and the process/thread in charge of their handling) tends to be one of the most expensive steps in software packet processing. This is true whether the processing thread is in user space or within the kernel.

There are however many hints that software packet forwarding is not *inherently* slow: many appliances on the market overcome performance issues bypassing the operating system and taking direct control of the hardware. Something similar is done in several research projects [14], [8], [12].

We have recently developed a framework, called `netmap` [22], which significantly reduces the cost of packet I/O in general purpose OS. Obviously, we wondered whether existing packet processing applications (software routers, firewalls, NAT, etc.) could be made faster by just replacing the underlying I/O routines with our more efficient version. Secondly, we were interested in how much effort, in terms of software modifications, this would require.

In this paper we report our experience in on the subject, which is extremely encouraging. Apart from trivial cases (traffic sources and sinks, simple traffic analyzers, for which no changes are required and speedups of 10 times are immediate), , we present two other applications as our main case studies: a software forwarding system called OpenvSwitch [20], and the userspace version of the Click Modular Router [14]. Both applications used `libpcap` [1] as their underlying packet I/O mechanism. In both cases, we managed to achieve huge performance improvements by just replacing the standard `libpcap` with our enhanced version (see Section III) running on top of `netmap`. We went from 780 Kpps to 3.0 Mpps for OpenvSwitch, and from 490 Kpps to 3.95 Mpps for Click[1].

However, reaching these impressive speedups required some investigation and some (limited) amount of programming effort. In fact, while packet I/O is often a significant bottleneck in a system, it not always the only one, and sometimes its presence hides other equally important performance impairments.

As an example, in the OpenvSwitch case, the performance of the original application was only 65 Kpps, and some modifications, as described in Section IV-C, were needed to exploit the benefits of the faster I/O library. The case of userspace Click is similar: just replacing the I/O library in the original application gave "only" 1.3 Mpps, which would be a great improvement in other contexts, but was not so

[1]Clearly, the speedup depends a lot on the type of task. CPU intensive tasks might see much lower gains. This does not disqualify our or similar work, just means that the specific application has totally different constraints.

exciting in our case, in light of the performance achieved with OpenvSwitch. In this case, as documented in Section V-B, we were able to find other bottlenecks that, once removed, enabled a further speedup by a factor of 3.

The fact that in both cases it was necessary to make modifications to the original applications does not contradict the title of this paper. The changes required were relatively small in terms of code size, and they addressed issues that existed independently from the use of a faster I/O library. In fact, the OpenvSwitch code, as modified, is over ten times faster than the original even without any faster I/O library. Once these changes were in place, we achieved exactly the result we wanted: running the same packet forwarding applications on top of an accelerated I/O library achieves speedups of 4..8 times.

The Click result is extremely important because of the widespread use of this tool. However, we expect the same gains for a number of other software packet processing applications, including firewalls and traffic monitors. The implications are quite large, from extending the life of existing equipment and applications to the next round of network speed increases, to enabling the use of software switching solutions in environments where it was not possible earlier due to cost or power constraints.

The code that we used for this work is publicly available [22] and ready for use in production systems.

The paper is structured as follows. In Section II we discuss the motivation and goals of our work, and give some background information on mechanism for packet I/O on general purpose operating systems.

Section III presents the first contribution of the paper, namely the accelerated `libpcap` library that is used in subsequent tests Section IV and Section V present the two case studies, illustrate how we accelerated the performance of two significant packet processing applications, and emphasize the issues that may arise in other similar systems. Finally, Section VI presents some related work.

II. Motivations and background

As mentioned earlier, software packet forwarding is commonly used in many experimental and production systems, and not infrequently these applications run on top of general purpose operating systems. Examples include native, in-kernel solutions for bridging and routing, routing daemons such as Zebra and Xorp [13], NAT boxes, firewalls and transparent proxies. More recently, experimental proposals [19], [2] for data center networks have emerged which use some non-standard functions in the forwarding layer, and could make use of a software implementation if sufficient performance were available.

Software packet forwarding requires at least three tasks:

- moving traffic between the network to the application;
- per-packet processing (ranging from simple Ethernet forwarding to more complex activities such as NAT and encryption);
- high-level processing such as running routing algorithms, reservation systems and traffic management.

The first two tasks have a large impact on the maximum forwarding performance, which is usually measured as the number of packets per second (pps) that the application can deliver because the dominant cost factors are only a weak dependency on packet sizes.

The performance of higher level processing is also important but it mostly affects the ability of the system to handle reconfigurations (such as route and interface state changes). We will not address this aspect in this paper.

Low speed operation (in the 10..100 Kpps range) is normally within reach even of the less expensive systems. As the speed of network links grows in the 1..10 Gbit/s range, pps rates increase by two orders of magnitude and go outside the capabilities of software packet processing systems. The fastest "standard" systems (i.e. those using ordinary kernel APIs) barely reach 1 Mpps (see e.g. [3]), and only prototypes using custom solutions [10], [8], [12] manage to go faster.

The motivation of this work is thus to investigate how the performance of existing packet forwarding software can be improved making use of faster I/O libraries such as `netmap`.

A. Packet I/O options

The first step in any software packet forwarding system is to move traffic between the network and the process/thread in charge of its manipulation. The network interface (NIC) is normally able to manage circular lists (called *NIC rings*) of memory buffers, and move packets between the physical links and these buffers without CPU intervention. So the actual issue, in terms of performance, is how to make the content of these memory buffers available to applications, and how to exchange signals (such as requests to transmit, or notification of incoming traffic) between the NICs and applications. There are three main approaches in use:

1) in-kernel operation: NICs are normally controlled by in-kernel device drivers, which encapsulate packets in OS-specific structures (mbufs, skbufs, NdisPacket) that carry all the metadata related to the packet themselves. Accessing packets through this mechanism from within the kernel saves some expensive operations such as system calls and data copies. This is the reason why several high performance packet processing applications are implemented within the kernel itself, either as an integral part of the system, or as a special module (such as in-kernel Click).

On the other hand, in-kernel operation poses a lot of restrictions on what a thread can do, and makes the system a lot harder to develop, debug and modify. For this reason, in-kernel operation is normally used only when the functions to be implemented are very well defined (e.g. IP routing, Layer 2 switching) or performance is a major concern.

2) user-space operation: Access to data packets is possible from user space as well, through specific APIs such as special sockets (the AF_SOCKET family), or mechanisms such as PF_RING [5], BPF [17] and libpcap [1].

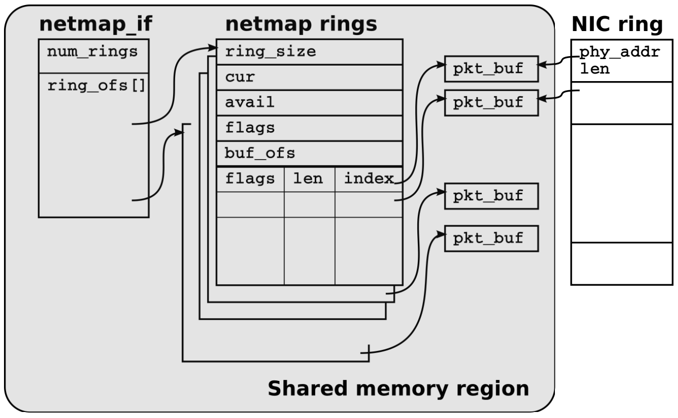

Fig. 1. The memory areas exported by netmap and used to represent rings and packet buffers

These mechanisms are more expensive than the in-kernel API, because they need system calls both to exchange synchronization information and to move actual data. On the other hand, they allows applications to run in a much more comfortable (for the developer) and safer (for the system) environment. For relatively low performance applications, the additional overhead is not a major concern and this is why there are cases where it makes sense to use this approach.

3) memory mapped I/O and direct NIC access: One approach to get the same (or possibly better) speed of in-kernel access and the convenience of use of user-space operation is to bypass the standard APIs used within the operating system. The two mainstream options here are memory-mapping of packet buffers (so that applications can access them without copies or system calls) and giving the application direct access to the NIC registers, so that even synchronization does not need to go through system calls. This is done in various proposals [6], [7], [15], [16], [11], and with different tradeoffs, which may make the application tied to a specific hardware, or make the operating system vulnerable to programming errors causing memory corruptions (the NIC has DMA engines that can write to arbitrary memory addresses, unless one uses protection mechanisms available on modern hardware).

B. The netmap framework

We have recently developed a novel framework to support packet I/O at speeds equal or exceeding those available by in-kernel APIs, and yet with much of the safety and convenience of userspace solutions. Our framework, called `netmap` [22], builds on a number of existing proposals for fast network access, reducing the cost for moving packets between the network and applications to about 90 clock cycles, almost one order of magnitude lower than the standard OS mechanisms. A full description of `netmap` is given in [22]. In this paper we only report a summary description and the features that are relevant to this work.

The basic goals of `netmap` are to make packet I/O efficient by removing all unnecessary run-time costs and system calls, and make the framework easy to use through a tight integration with existing OS mechanisms. To this end, the `netmap` API exposes to userspace processes shadow copies of the NIC rings, and uses them to support packet I/O and synchronization. These shadow copies, called *netmap rings*, are shown in Figure 1. The memory region containing the netmap rings and the packet buffers is mapped in the process' address space through the `mmap()` system call. This permits to transfer data with no copy overhead.

Each ring has a device-independent format, and includes the index (`cur`) of the first buffer to send or receive packets, and the number (`avail`) of buffers available for transmit or receive. The state of a netmap ring is updated by the kernel (to reflect the state of the corresponding NIC ring) only when executing `poll()`[2] or certain `ioctl()` system calls. Hence no locking is needed to access the netmap ring, and a system call carries information on a potentially large number of packets.

Synchronization is done exclusively with the `poll()` system call. In order to use netmap, a process opens a special device (`/dev/netmap`) and issues an `ioctl()` to set a specific NIC in netmap mode. In this mode the NIC is still visible and controlled by the operating system, but the transfer of packets from/to the host stack is blocked. Passing the file descriptor to `poll()` lets the application know when there are newly received packets, or buffers available for transmission. So applications do not need busy-waiting or other custom mechanisms to synchronize with the hardware.

If a network card has multiple NIC rings (this is the case on modern high speed NICs) `netmap` exports a corresponding number of netmap rings. Multiple file descriptors can be opened for each card, and the mapping of file descriptors to rings is fully programmable, so one can easily dedicate multiple threads (possibly associated to different cores using the standard `setaffinity()` system call) to a single NIC. At the same time, the same thread can access multiple cards, and implement zero-copy packet forwarding between NICs thanks to the fact that all the rings and buffers (Figure 1) for all NICs are in the same memory region.

III. LIBPCAP, AND LIBPCAP OVER NETMAP

`netmap` has a very simple API, but being very new, it is not widely used by applications. Many packet processing systems, including our two case studies, use the `libpcap` API to send and receive packets, so it makes sense to build an adaptation layer that maps one API into another. Usually these adapters cause some performance penalty but, as we will see, in our case this is reasonably low.

Despite its large number of functions, `libpcap` is relatively straightforward to use. The library supplies functions to attach to a NIC, and return a file descriptor which can be used, among other things, as an argument to a `poll()` system call. The function `pcap_inject()` is one of the mechanisms that `libpcap` clients can use to send packets to a device. For

[2]The `select()` system call is equivalent to `poll()`. Throughout the paper we will only mention the former for brevity.

reception, the method that suits best our needs is the function `pcap_dispatch()`, which applies a function to a batch of packets received by the NIC. This mechanism is interesting because it can amortize the function call over the entire batch, and because it can operate on an externally supplied packet buffer instead of requiring data to be in a specific location.

```
int pcap_dispatch(pcap_t *p, int cnt,
    pcap_handler callback, u_char *user)
{
  int i, idx, si, ret = 0;
  struct netmap_ring *ring;

  for (si =p->begin; si < p->end; si++) {
    ring = NETMAP_RXRING(p->nifp, si);
    while ((cnt == -1 || ret != cnt) && ring->avail > 0) {
      i = ring->cur;
      idx = ring->slot[i].buf_idx;
      p->hdr.len = p->hdr.caplen = ring->slot[i].len;
      callback(user, &p->hdr, NETMAP_BUF(ring, idx));
      ring->cur = NETMAP_RING_NEXT(ring, i);
      ring->avail--;
      ret++;
    }
  }
  return ret;
}


int pcap_inject(pcap_t *p, void *buf, size_t size)
{
  int si, idx;
  struct netmap_ring *ring;

  for (si = p->begin; si < p->end; si++) {
    ring = NETMAP_TXRING(p->nifp, si);
    if (ring->avail == 0)
      continue;
    idx = ring->slot[ring->cur].buf_idx;
    bcopy(buf, NETMAP_BUF(ring, idx), size);
    ring->cur = NETMAP_RING_NEXT(ring, ring->cur);
    ring->avail--;
    return size;
  }
  return -1;
}
```

Fig. 2. The complete code for `pcap_dispatch()` and `pcap_inject()` used in our adaptation library.

Our version of `libpcap` that runs on top of `netmap` is called `netmapcap`. It implements the basic `libpcap` functions to access a NIC, and in particular the two functions that are used to send and receive packets, whose complete code is shown in Figure 2. The code is also an example of the use of the native `netmap` APIs and how multiple NIC rings are used. Due to space limitations we only point out a few details: i) the receive function, `pcap_dispatch()`, operates directly on the memory mapped buffer, avoiding memory copies; ii) the timestamp in the `hdr` field, needed by some `libpcap` clients, is pre-set by `netmap` so it does not need an additional system call to be computed; iii) both the send and receive functions rely on a subsequent `poll()` to synchronize the state of the netmap and NIC rings.

The loss of performance using `netmapcap` instead of the native `netmap` API is not too high. As an example, a simple application that passes packets between two 10 Gbit/interfaces using a single CPU core reaches about 7.50 Mpps (unidirectional) with `netmapcap`, and 9.42 Mpps with the native `netmap` API under the same conditions[3]. The throughput with the original `libpcap` is less than 1 Mpps.

IV. Case study: OpenvSwitch

We now move to the evaluation of the performance of the two processing systems that we use as case studies, and the discussion of the modifications that were needed to achieve good performance.

Our first example is OpenvSwitch [20], a software implementation of a layer-2 switch designed to scale to large configurations. Same as many modern switching/routing solutions, OpenvSwitch (Figure 3) separates the *datapath* (in charge of the actual switching) and the *controller*, which is in charge of management and controlling the forwarding tables for the data plane. This particular software is interesting because packet processing on a switch is particularly simple, hence this kind of device suffers the most from operating system overheads, and can benefit a lot from improvements in that area. Also, OpenvSwitch also supports a userspace datapath so we can evaluate its performance with all forms of packet I/O described in Section II-A.

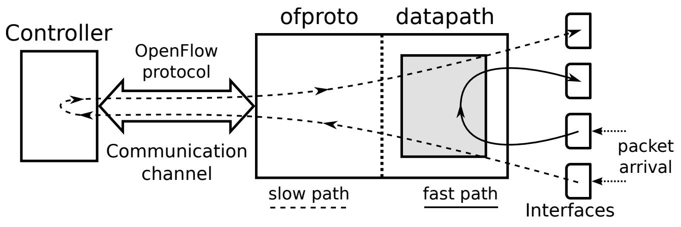

Fig. 3. OpenvSwitch stucture, representing the flow of traffic in the system.

A. Architecture

The datapath implements the basic Ethernet switching functions, with the help of a local flow table that decides the fate of incoming packets. If an incoming packet has a matching entry in the flow table, it is processed directly in the datapath (on the “fast path” indicated in the figure) and sent to the destination interface. Packets without matching entries in the flow table, or requiring more complex operations, are send towards the controller with the help of another module (called “ofproto” in the figure). Processing these packets is more expensive, and this is called the “slow path” of the packet processing chain. As an example, the slow path is used for packets with not-yet-seen source or destination addresses: they are sent to the controller which is in charge of running whatever algorithm is involved to determine the location of the station, and update the flow table accordingly.

Figure 4 details the interaction between the datapath and ofproto blocks. The former is in charge of pure traffic switching, whereas the latter manages the slow path, communication with the controller, and interface-related events.

The original distribution of OpenvSwitch provides two different implementations of the datapath: one as a Linux

[3]Removing the timestamp would make the pure `netmap` version run at 10.66 Mpps.

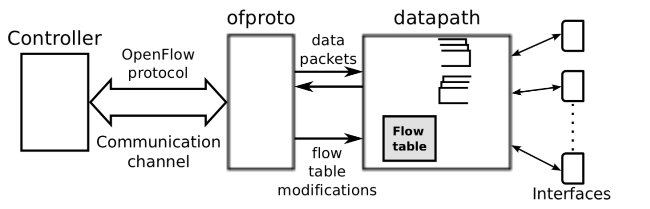

Fig. 4. Communication between the datapath and ofproto blocks in OpenvSwitch.

Initial OpenvSwitch performance	
Configuration	Kpps
Linux userspace	50
FreeBSD userspace	65
Linux Kernel	300
Other systems	
Configuration	Kpps
original Click userspace	400
native FreeBSD bridging	690
netmapcap forwarding	7,500
netmap forwarding	9,660

Fig. 5. Forwarding performance of the original OpenvSwitchimplementation, and comparison with other solutions.

kernel module, and one as a userspace module, still for Linux, and using the PF_PACKET sockets API to exchange packets. For our experiments, we have implemented a third datapath module, working under FreeBSD and using `libpcap` as a packet I/O library.

B. Native Performance

Determining the baseline performance of the system is fundamental to evaluate the improvements that can be achieved using a faster packet I/O mechanism.

For this and all tests in this paper we use two systems equipped with an Intel i7-870 CPU (four cores, but only one used in the experiments) running at frequencies up to 2.93 GHz. Each system has multiple Ethernet interfaces including a dual-port 10 Gbit Ethernet mounted on a PCI-e bus with 8 lanes, which gives sufficient I/O bandwidth to sustain the traffic generated by the interfaces[4]. This is the same platform used in [22] and for which, in Section III, we have measured a forwarding rate in excess of 7 Mpps using `netmapcap`.

In terms of OpenvSwitch configuration, in the experiments reported here we have used a simple case with only two interfaces and unidirectional traffic of 64-byte packets to a single destination. The performance of the system degrades with the number of interfaces but investigation of the phenomenon is beyond the scope of this paper. Similarly, the behaviour of the system in presence of multiple flows and high turnaround (when a significant fraction of the traffic goes through the controller) is studied in [20] but is not of interest here.

Initial measurements were quite disappointing, as shown by the Table in Figure 5. While we could expect a low performance of the userspace versions, the 50-65 Kpps we measured are almost one order of magnitude below our expectations. Even the in-kernel version is somewhat unsatisfactory, achieving only about 300 Kpps, compared to a native performance with FreeBSD bridging of about 690 Kpps. Nevertheless, we are not interested in the in-kernel performance so we have not investigated this behaviour further.

The existing literature on OpenvSwitch performance does not help much. Pfaff and others [20, Section 4.2] compare OpenvSwitch and Linux Ethernet Bridging under various conditions. At large flow sizes, when the controller's overhead is negligible, they reports about 450 Mbit/s, about half the speed of the link used in the test. The packet sizes used in those tests are unknown, though we suspect TCP flows with 1500-byte packets, which would result in about 37.5 Kpps.

[4]Sufficient bus bandwidth does not mean that the NIC is actually able to run at wire rate in all conditions. As shown in [22], there are several cases where the NIC is unable to deal with packets at line rate, despite the CPU and the bus would be able to support that.

C. Hunting the performance bottleneck

Investigations on the poor userspace performance quickly revealed the source of the problem. In this implementation, declared experimental by the OpenvSwitch authors, there is a single event loop handling all events for both the ofproto and datapath blocks in Figure 4. This means that the event loop has to deal with a much larger number of file descriptors than just those for the interfaces, and this makes each iteration extremely expensive. On top of this, the event loop processes at most one packet per interface per iteration, and fires all handlers even if the corresponding file descriptor is not marked as ready.

We have then addressed the problem by using a separate thread to implement the datapath, and allowing the processing multiple packets per interface at each iteration of the loop. Though conceptually a major one, the modification is relatively simple in terms of code. The main eventloop remains the same, except that it ignores the file descriptors for packet I/O which are now handled by the datapath thread. Notifications between the two threads are exchanged using a unix pipe (whose endpoints can be used as arguments to the `poll()` system calls in each event loop), whereas data packets that need to go through the slow path and other messages are exchanged through a couple of queues.

With this modification, performance reaches reasonable levels: using the standard `libpcap`, the throughput of is now between 383 and 783 Kpps (see Figure 6, center column) depending on the maximum number of packets per NIC (batch size) that we process at each iteration of the eventloop. We will take this as a reference to measure improvements with the faster I/O library.

D. OpenvSwitch over netmapcap

We are now ready to evaluate the performance improvements that can be achieved running the application on top of a faster I/O library. For this tests, all we needed was to replace the system's libpcap with our own version described in Section III. The results are shown in the right column of

Batch size	Throughput (Kpps)	
	libpcap	netmapcap
1	373	250
2	510	466
3	579	619
5	638	882
10	709	1430
20	744	1970
50	764	2500
100	775	2730
500	781	2960

Fig. 6. Forwarding performance of the optimized OpenvSwitch code with different batch sizes and I/O libraries.

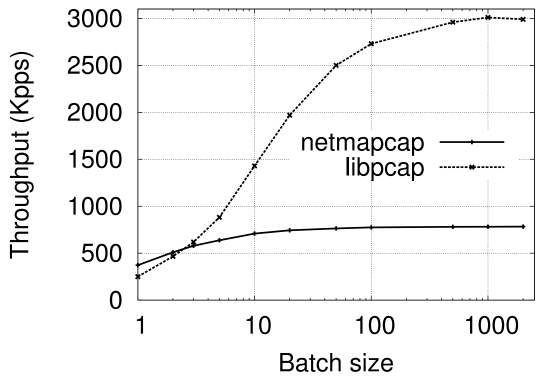

Fig. 7. Forwarding performance of the optimized OpenvSwitch code (data from Figure 6).

the Table in Figure 6, and plotted in Figure 7. As can be seen from the data, the system is between three and four times faster even with relatively small batch sizes (50 packets).

We notice that the batch size has a large impact on the performance, so there would be an incentive in using large values for this parameter. The only concern in using large sizes is that packets could be moved in large bursts from one interface to another, possibly leading to overflows in case of short queues or rate mismatches. In practice, however, at high speeds the NIC rings are quite large (256 or more, 2048 in our case) and a batch size of 50 provides a reasonable compromise between performance and safety of operation.

Even though it is not a realistic operating point, it is interesting to understand why `netmapcap`'s performance with small batch sizes (1 or 2) is significantly slower than `libpcap`. The reason is that in our tests we are handling unidirectional traffic, so one of the descriptors passed to `poll()` corresponds to an empty ring, and the low level handler called by `poll()` when a ring is empty is more expensive in `netmap` (hence in netmapcap) than it is in `libpcap`. This explains the difference in performance. Running the same test with a batch size of 1 and bidirectional traffic brings the performance to 900 Kpps per direction, which is perfectly in line with our expectations.

Finally, we should point out that the huge speedup achieved with `netmapcap` is the result of a redesign of the features of `netmap` so that it could emulate efficiently the `libpcap` API. With the original version of `netmap`, the eventloop required one additional system call per iteration to fetch the timestamp, and one additional system call per interface/loop to push out any packets that were queued on a ring. Investigation on the common usage patterns of the `netmap` API has suggested the addition of a number of programmable features so that, for instance, `poll()` on a netmap file descriptor also returns an up-to-date timestamp which is used by many libpcap clients, and pushes out any packet that are queued on the transmit ring associated to the file descriptor.

E. Efficiency, not just speed

The performance of applications is often measured in terms of peak speed, but we should mention that our work achieves speed not by brute force or throwing hardware at the problem, but by making the entire system more efficient. The following graphs illustrate the issue from two points of view.

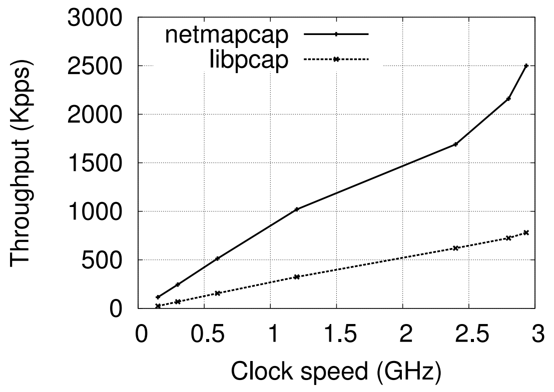

Fig. 8. Throughput versus CPU frequency for the optimized OpenvSwitch code (batch size=50).

In Figure 8 we plot the forwarding rates for different CPU speeds, using a batch size of 50 and an offered load of 14.88 Mpps. As the graph shows, the large performance difference using the two libraries exists at all clock speeds, meaning that a given packet rate can be processed at a much lower clock frequency (and significantly reduced power consumption)[5] using `netmapcap`.

Figure 9 plots the CPU usage (computed with `top`) with different offered loads. In general this kind of data must be collected and read with extreme care, because they may be affected by large errors. As an example, our plot is noisy and deviates from the expected linear shape, for these reasons: our packet source is bursty; the device driver uses interrupt mitigation [18], which helps reducing the CPU load but might cause some synchronization phenomena with the source itself; CPU ticks are charged using sampling, and so depending on the timing they might be assigned to the wrong process.

[5]The superlinear behaviour between the last two points (2.8 and 2.93 GHz) is because at the highest speed the CPU uses the “Turbo Boost” mode, where it autonomously increases the clock speed up to 3.2 GHz if thermal constraints are satisfied. Hence the nominal rate of 2.93 GHz does not match the actual clock speed.

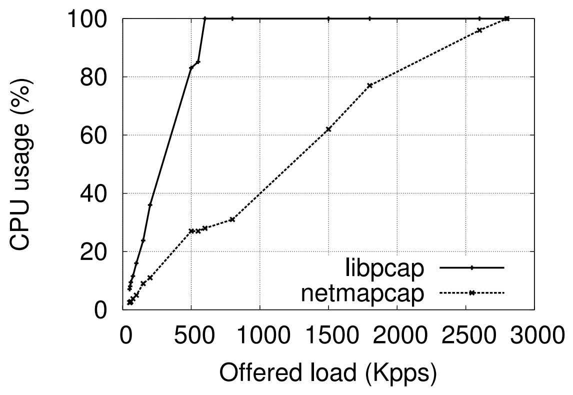

Fig. 9. OpenvSwitch: CPU usage versus offered load at maximum frequency.

With all the above in mind, the information (at least qualitative) that we get from Figure 9 is that running over `netmapcap` uses about 1/3 of the CPU resources used with the standard `libpcap`. Also, it should be remembered that even when the load reaches 100%, the system has generally not reached saturation. At that stage there is in fact a self-adaptation phenomenon so that userspace processing takes longer, causing fewer system calls returning each a longer batch of packets, hence resulting in lower per-packet overhead. The phenomenon is more evident without interrupt mitigation (not our case), where usually 100% usage is reached much before the peak forwarding rate.

V. Case study: Click

Our second case study is represented the userspace version of the Click modular router. Click [14] is an extremely popular platform to build packet processing systems, widely used in both research and actual deployments. Click lets the user create packet processing graphs (Figure 10 by connecting standard or user-defined processing elements using a script-like configuration language. Input and output ports are used to transfer data packets, using *push* or *pull* mechanism. A push port results in a direct call of the processing function in the element downstream; a pull port means that the element downstream is in charge of polling the upstream port to extract any available packet. Among the large set of standard Click elements there are some directly connected to hardware interfaces, so that actual traffic can be injected and extracted from a Click graph.

Click can be run in two different environments: the *userspace* version is simply one process (or a set of threads in recent versions) running in userspace, and accessing devices using `libpcap` or equivalent libraries. The *in-kernel* version is by far the most popular, and it has no significant difference from the other one except from the ability to access the interfaces using custom elements (PollDevice and ToDevice) with much better performance. The literature reports send and receive rates, for in-kernel Click configurations, of up to 3-4 Mpps per core. Actual values are difficult to compare because the various papers use slightly different hardware and software configurations, but in general in-kernel Click performance exceeds the native Linux performance by at least 2-3 times. Given the convenience of userspace operation, bringing the performance of userspace Click to that of its kernel counterpart would be extremely valuable.

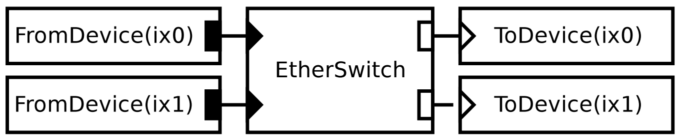

Fig. 10. Click configurations are created by connecting elements in charge of different processing functions.

A. Usermode Click - native performance

Same as for our other case study, we need to determine the forwarding performance of an unmodified version of Click running in userspace. In our tests we use the same hardware used for the OpenvSwitch tests, and Click 1.8.0 running in userspace, with some minimal FreeBSD modifications to make use of the `libpcap` API. Only one thread is used in the experiments. Our click configuration is very simple[6]:

```
FromDevice(ix0) -> Queue -> ToDevice(ix1)
FromDevice(ix1) -> Queue -> ToDevice(ix0)
```

and replicates the setup of Section III where we do direct forwarding of traffic between interfaces.

In this configuration, unidirectional forwarding reaches about 400 Kpps, which is reasonably close to the throughput of other userspace applications using libpcap. Looking at this number we have no reason to suspect the existence of other significant bottlenecks, although the following issue might need further investigation as we move to faster speeds.

Click has an internal scheduler which decides which element to run next. Elements use a form of cooperative multitasking in which they run once, performing only non blocking activities, and then return to the scheduler with the indication of whether they are ready for an extra iteration or not. The amount of work done by each element of course depends on the element itself, but in many cases it corresponds to the processing of just one packet from input to output (and in the case of push element, this may cause the direct call of a function on the downstream element, and so on). Some elements have a BURST parameter which decides how many packets should be processed before yielding the CPU. This way of operation means that the execution of the elements is interleaved with that of the scheduler, and so the latter constitutes an extra cost that can be reduced (amortized) using larger BURST values. The benefit clearly depends on the cost of the scheduling operation compared to that of running the element itself.

The default ToDevice element does not have a BURST parameter so we cannot experiment with it. However, 400 Kpps

[6]We omit details of the configuration, such as setting promiscuous mode on the interfaces, changing queue and burst sizes, and the scripts to print the throughput every second.

Original Click		
version	Burst	Kpps
libpcap	-	400
netmapcap	-	1300
kernel (est.)	-	3-4000
Modified Click		
version	Burst	Kpps
libpcap	1	490
libpcap	50	495
netmapcap	1	3300
netmapcap	50	3950

Fig. 11. Performance of userspace Click in various configurations. The in-kernel speed is derived from the literature

corresponds to 2500 ns per packet, presumably much higher than the cost of a scheduling decision. Hence we expect that, at least in this particular configuration, changing the burst size should not improve performance too much.

When no elements are ready, or at least every so often, the scheduler call `poll()` to wait for timeouts or devices to become ready. Unlike the OpenvSwitch case, changing the BURST size does not affect the frequency with which `poll()` is invoked, as the latter only happens when no elements are ready.

B. Click over netmapcap

A first test replacing `libpcap` with `netmapcap` gives a sudden increase of the performance to 1.3 Mpps, about three times faster. Without further information one would be led to think of this as a significant and satisfactory speedup. However, considering that our Click configuration runs a much simpler task than the one implemented by OpenvSwitch, we would at least expect a similar result, in the 2.5-3 Mpps range.

At first, the attention went to the scheduler, and we modified the ToDevice element adding a BURST parameter to process multiple packets at each invocation. This gave only a modest speed improvement, suggesting some other time-consuming operation on each packet. The usual suspects for this type of problems are either system calls, or locks or similar operations which have significant latency.

Diagnosing this problem took a bit more time than for the OpenvSwitch case, because we could find no obvious problem in the C++ code that implements the elements or the Click scheduler. A significant help to the investigation came from the performance analysis of a much simpler configuration which replaced one or both of FromDevice and ToDevice with an InfiniteSource and a Sink. These elements perform no device I/O whatsoever, and even after removing the system call (`gettimeofday()`) that usually pollutes network code applications, speed was well below the expected values. This shifted the attention to the only remaining operation, namely the memory allocator, and this turned out to be the actual issue.

In Click, each Packet (a user defined C++ object) requires the allocation of two blocks of memory, one for the descriptor and one for the payload. These allocations are done through the standard C++ allocator, which is probably acquiring some lock around every allocation, and is also designed for variable size objects. Click's Packets can be easily implemented with fixed-size objects, and memory recycled within each Click thread without recurring to the centralized C++ allocator.

Applying this change to the generic Click code increased the throughput to 490 Kpps when running over `libpcap`, and 3.3 Mpps running over `netmapcap`, still with the default BURST size of 1. At this level of performance, even a modest scheduler cost is significant, and in fact increasing the burst size to 50 gives a further significant speedup in the `netmapcap` case, reaching almost 4 Mpps. This level of performance, in userspace and using a single core, is similar or possibly exceeding that of in-kernel Click configurations.

Figure 11 summarizes the results of original and modified Click in various operating modes. As we can see, we achieved an improvement of almost 10 times over the original performance, and even in this case a significant help came from the removal of an underestimated performance bottleneck unrelated to packet I/O. The modifications that led to this improvement have been contributed back to the Click project and are now included in the recent Click 2.0 distribution.

VI. Related work

The two areas of work related to this paper are on mechanisms for doing packet I/O in user space, and on the performance of software packet processing.

A. Packet I/O mechanisms

Most conventional mechanisms for doing packet I/O from userspace suffer from the problems listed in Section II: the device driver operation is expensive, and often the mechanism requires per-packet system calls with further compromise the performance. The Berkeley Packet Filter, or BPF [17], is one of the most popular systems to support direct access to raw packet data. It works by tapping into the data path of a network device driver, and dispatching a copy of each received or transmitted packet to a special device, from which userspace processes can read or write. Linux has a similar mechanism through the AF_PACKET socket family. `libpcap` [1] is a popular high-level mechanism to access raw packets. Although the original versions of this library were based on BPF, current versions support a number of low-level mechanisms depending on the available hardware and operating system.

More efficient solutions, similar in spirit to `netmap`, are those used by netslice [16] and PacketShader [12]. Both work by removing the in-kernel packet management described in Section II-A, and binding the NIC rings to device descriptors, from which userspace applications can read/write batches of buffers with a single system call. The I/O channel implemented by PacketShader has a reported performance similar to that of `netmap`. We expect netslice to be in the same range.

Even though `netmap` and PacketShader have demonstrated that the system call overhead is not a significant obstacle to performance, system calls are used for synchronization and they introduce some latency in the notification of events, especially if interrupt mitigation is used (but mitigation can be removed or its delay be increased in time-critical systems).

It is the opinion of these authors that an additional latency of a few microseconds falls in the noise for network systems subject to unpredictable load. Nevertheless, some systems try to run without system calls (and with instant notification of events, by continuously polling the NIC's memory regions) by giving the user space process direct access to the hardware. Examples include UIO-IXGBE [15], and a recent variant of PF_RING [5] called DNA [6].

B. Software router performance

The performance of software packet processing has been the subject of investigation for a long time. As discussed in this paper, the cost of packet I/O is only one of the terms of the equation, and we have pointed out that software features may significantly compromise the speed of operation of a system. There are however other issues related to the platform or to the type of processing that must be performed, that are equally important. In particular, historically we have seen a number of hardware aspects that turned out to be the dominant bottleneck in such systems. As an example, the I/O bus used to (and still does, especially on low-end systems) have insufficient bandwidth to sustain the full link speed. The peak performance in the original Click prototype [14] as well as in other systems [21] was indeed limited by the PCI bus rather than the CPU speed. Memory latency and cache sizes are also an important bottleneck, especially in systems that perform a lot of data manipulation [9], [10].

Apart from these low level issues, there are a number of papers which report on the performance of software routing systems, showing how the native throughput of general purpose OS is relatively low [4], while in-kernel Click-based solutions perform much better [3], [10], both on a single CPU and in terms of scalability. The two most recent proposals in the literature further address the scalability aspects by looking at how to achieve large port counts [8] and how to make use of Graphic Processing Units as network coprocessors [12].

VII. Conclusions

In this paper we have shown how the performance of userspace packet processing systems can be largely improved by replacing the underlying packet I/O libraries with more efficient systems, such as one developed by the authors. We hoped to achieve our goal without any modification to the existing software, and the two case studies presented here even suggest that this was a realistic option. In practice, however, the same examples have shown that it is very likely that certain performance bottlenecks remain hidden because of dominant ones (in our case, a slow packet I/O mechanism), and are only exposed when the system runs in a more performant environment.

The results presented here are nevertheless extremely interesting, not only because we achieved a huge (4..10 times) speedup of the original applications with relatively limited effort, but also because we expect that the same results, and the same methodology to investigate and improve performance, applies to many existing similar systems.

References

[1] Tcpdump and libpcap web site. *http://www.tcpdump.org/*.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38:63–74, August 2008.

[3] A. Bianco, R. Birke, D. Bolognesi, J.M. Finochietto, G. Galante, M. Mellia, M.L.N.P.P. Prashant, and F. Neri. Click vs. linux: two efficient open-source ip network stacks for software routers. In *High Performance Switching and Routing, 2005. HPSR. 2005 Workshop on*, pages 18 – 23, may 2005.

[4] Raffaele Bolla and Roberto Bruschi. Pc-based software routers: high performance and application service support. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '08, pages 27–32, New York, NY, USA, 2008. ACM.

[5] L. Deri. Improving passive packet capture:beyond device polling. In *SANE 2004, Amsterdam*.

[6] L. Deri. ncap: Wire-speed packet capture and transmission. In *Workshop on End-to-End Monitoring Techniques and Services*, pages 47–55. IEEE, 2005.

[7] L. Deri, J. Gasparakis, P. Waskiewicz, and F. Fusco. Wire-speed hardware-assisted traffic filtering with mainstream network adapters. *Advances in Network-Embedded Management and Applications*, pages 71–86, 2011.

[8] M. Dobrescu, N. Egi, K. Argyraki, B.G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *ACM SOSP*, pages 15–28, 2009.

[9] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley. Evaluating xen for router virtualization. In *Computer Communications and Networks, 2007. ICCCN 2007. Proceedings of 16th International Conference on*, pages 1256 –1261, aug. 2007.

[10] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 20:1–20:12, New York, NY, USA, 2008. ACM.

[11] Alexander Fiveg. Ringmap capturing stack for high performance packt capturing in freebsd. *http://ringmap.googlecode.com/ files/ringmap_slides.pdf*, 2010.

[12] S. Han, K. Jang, K.S. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[13] M. Handley, O. Hodson, and E. Kohler. Xorp: An open platform for network research. *ACM SIGCOMM Computer Communication Review*, 33(1):53–57, 2003.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[15] Max Krasnyansky. Uio-ixgbe. *https://opensource.qualcomm.com/wiki/UIO-IXGBE*.

[16] Tudor Marian. Operating systems abstractions for software packet processing in datacenters. *PhD Dissertation, Cornell University*, 2010.

[17] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX Winter Conference*. USENIX Association, 1993.

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

[19] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[20] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenkerz. Extending networking into the virtualization layer. In *Proceedings of the ACM SIGCOMM HotNets, October 2009*. ACM, 2009.

[21] L. Rizzo. Polling versus interrupts in network device drivers. *BSD-ConEurope 2001*, 2001.

[22] Luigi Rizzo. netmap: fast and safe access to network adapters for user programs. *Tech. Report, Univ. di Pisa, June 2011, http://info.iet.unipi.it/∼luigi/netmap/*, 2011.