Fast Userspace Packet Processing

Tom Barbette
University of Liege
Belgium
tom.barbette@ulg.ac.be

Cyril Soldani
University of Liege
Belgium
cyril.soldani@ulg.ac.be

Laurent Mathy
University of Liege
Belgium
laurent.mathy@ulg.ac.be

ABSTRACT

In recent years, we have witnessed the emergence of high speed packet I/O frameworks, bringing unprecedented network performance to userspace. Using the Click modular router, we first review and quantitatively compare several such packet I/O frameworks, showing their superiority to kernel-based forwarding.

We then reconsider the issue of software packet processing, in the context of modern commodity hardware with hardware multi-queues, multi-core processors and non-uniform memory access. Through a combination of existing techniques and improvements of our own, we derive modern general principles for the design of software packet processors.

Our implementation of a fast packet processor framework, integrating a faster Click with both Netmap and DPDK, exhibits up-to about 2.3x speed-up compared to other software implementations, when used as an IP router.

1. INTRODUCTION

Recent years have seen a renewed interest for software network processing. However, as will be shown in section 3, a standard general-purpose kernel stack is too slow for linerate processing of multiple 10-Gbps interfaces. To address this issue, several userspace I/O frameworks have been proposed. Those allow to bypass the kernel and obtain efficiently a batch of raw packets with a single syscall, while adding other capabilites of modern Network Interface Cards (NIC), such as support for multi-queueing.

This paper first reviews the technical aspects of some existing userspace I/O frameworks, such as Netmap [19], Intel's DPDK [10], OpenOnload [22], PF_RING [16], PacketShader I/O [8] and Packet_MMAP [14]. Some other works go further than a "simple" Linux module bypassing the kernel, like IX [5] and Arrakis [17]. We won't consider them as, for our purpose, they only offer fast access to raw packets, but in a more protected way than the others I/O frameworks, using virtualization techniques, and they were not fully available at the time this paper was written.

To explore their performance inside a general purpose environment, we then compare the existing off-the-shelf integrations of some of these frameworks in the Click Modular Router [12]. Click allows to build routers by composing graphs of *elements*, each having a single simple function (e.g. decrementing a packet TTL). Packets then flow through the graph from input elements to output elements. Click offers a nice abstraction, includes a wealth of usual network processing elements, and already has been extended for use with some of the studied I/O frameworks. Moreover, we think its abstraction may lend itself well to network stack specialization (even if it's mostly router-oriented for now).

Multiple authors proposed enhancements to the Click Modular Router. RouteBricks [6] focuses on exploiting parallelism and was one of the first to use the multi-queue support of recent NICs for that purpose. However, it only supports the in-kernel version of Click. DoubleClick [11] focuses on batching to improve overall performances of Click with PacketShader I/O [8]. SNAP [23] also proposed a general framework to build GPU-accelerated software network applications around Click. Their approach is not limited to linear paths and is complementary to the others, all providing mostly batching and multi-queuing. All these works provide useful tips and improvements for enhancing Click, and more generally building an application on top of a "raw packets" interface.

The first part of our contribution is the critical analysis of those enhancements, and discuss how they interact with each other and with userspace I/O frameworks, both from a performance and a configuration ease points of view.

While all those I/O frameworks and Click enhancements were compared to some others in isolation, we are the first, to our knowledge, to conduct a global survey of their performance and, more importantly, interactions between the features they provides. Our contribution include new discoveries resulting from this in-depth factor analysis, such as the fact that the major part of performance improvement often attributed to batching is more due to the usage of a run-to-completion model, or the fact that locking can be faster than using multi-queue in some configurations.

Finally, based on this analysis, and new ideas of our own such as a graph analysis to discover the path that each thread can take to minimize the use of memory and multi-queue, we propose a new userspace I/O integration in Click (including a reworked implementation for Netmap, and a novel one for Intel's DPDK). Our approach offers both simpler configuration and faster performance. The network operator using Click does not need to handle low-level hardware-related configuration anymore. Multi-queue, core affinity and batching are all handled automatically but can still be tweaked. On the contrary to previous work which broke the compatibility by requiring a special handling of batches, our system is retro-compatible with existing Click elements. The Click developer is only required to add code where batching would improve performance, but it's never mandatory.

This new implementation is available at [4] and will conclude our contribution with the fastest version of Click

978-1-4673-6633-5/15/$31.00

we could achieve on our system, more flexible and easy-to-use with regard to modern techniques such as NUMA-assignement, multi-queueing, core and interrupt affinity or configuration of the underlying framework.

Section 2 reviews a set of existing userspace I/O frameworks. Section 3 then evaluates their forwarding performance. Section 4 discusses how some of these frameworks were integrated into the Click modular router. Section 5 analyzes those integrations, and various improvements to Click, giving insights for the design of fast userspace packet processors. We then propose FastClick, based on those insights. Finally, section 6 evaluates the performance of our implementation and section 7 concludes the paper.

2. I/O FRAMEWORKS

In this section, we review various features exhibited by most high performance userspace packet I/O frameworks. We then briefly review a representative sample of such frameworks.

2.1 Features

Zero-copy. The standard scheme for receiving and transmitting data to and from a NIC is to stage the data in kernelspace buffers, as one end of a Direct Memory Access (DMA) transfer. On the other end, the application issues read/write system calls, passing userspace buffers, which copy the data, across the protection domains, as a memory-to-memory copy.

Most of the frameworks we review aim to avoid this memory-to-memory copy by arranging for a buffer pool to reside in a shared region of memory visible to both NICs and userspace software. If that buffer pool is dynamic (i.e. the number of buffers an application can hold at any one time is not fixed), then true zero-copy can be achieved: an application which, for whatever reasons must hold a large number of buffers, can acquire more buffers. On the other hand, an application reaching its limit in terms of held buffers would then have to resort to copying buffers in order not to stall its input loop (and induce packet drops).

Note however that some frameworks, designed for end-point applications, as opposed to a middlebox context, use separate buffer pools for input and output, thus requiring a memory-to-memory copy in forwarding scenarios.

Kernel bypass. Modern operating system kernels provide a wide range of networking functionalities (routing, filtering, flow reconstruction, etc.). This generality does, however, comes at a performance cost which prevents to sustain linerate speed in high-speed networking scenarios (either multiple 10-Gbps NICs, or rates over 10 Gbps). To boost performance, some frameworks bypass the kernel altogether, and deliver raw packet buffers straight into userspace. The main drawback of this approach is that this kernel bypass also bypasses the native networking stack; the main advantage is that the needed userspace network stack can be optimized for the specific scenario [13].

In pure packet processing applications, such as a router fast plane, a networking stack is not even needed. Note also that most frameworks provide an interface to inject packets "back" into the kernel, at an obvious performance cost, for processing by the native networking stack.

I/O batching. Batching is used universally in all fast userspace packet frameworks. This is because batching amortizes, over several packets, the overhead associated with accessing the NIC (e.g. lock acquisition, system call cost, etc.).

Hardware multi-queue support. Modern NICs can receive packets in multiple hardware queues. This feature was mostly developed to improve virtualization support, but also proves very useful for load balancing and dispatching in multi-core systems. Indeed, for instance, Receiver-Side Scaling (RSS) hashes some pre-determined packet fields to select a queue, while queues can be associated with different cores.

Some NICs (such as the Intel 82599) also allow, to some extent, the explicit control of the queue assignment via the specification of flow-based filtering rules.

2.2 Frameworks

Packet_mmap [14] is a feature added to the standard UNIX sockets in the Linux kernel[1], using packet buffers in a memory region shared (mmaped, hence its name) between the kernel and the userspace. As such, the data does not need to be copied between protection domains. However, because Packet_mmap was designed as an extension to the socket facility, it uses separate buffer pools for reception and transmission, and thus zero-copy is not possible in a forwarding context. Also, packets are still processed by the whole kernel stack and need an in-kernel copy between the DMA buffer and the sk_buff, only the kernel to user-space is avoided and vice versa.

PacketShader [8] is a software router using the GPU as an accelerator. For the need of their work, the authors implemented PacketShader I/O, a modification of the Intel IXGBE driver and some libraries to yield higher throughput. It uses pre-allocated buffers, and supports batching of RX/TX packets. While the kernel is bypassed, packets are nevertheless copied into a contiguous memory region in userspace, for easier and faster GPU operations.

Netmap [19] provides zero-copy, kernel bypass, batched I/O and support for multi-queueing. However, the buffer pool allocated to an application is not dynamic, which could prevent true zero-copy in some scenarios where the application must buffer a lot of packets. Recently, support for pipes between applications has also been added. Netmap supports multiple device families (IGB, IXGBE, r8169, forcedeth, e1000 and e1000e) but can emulate its API over any driver at the price of reduced performance.

PF_RING ZC (ZeroCopy) [16] is the combination of ntop's PF_RING and ntop's DNA/LibZero. It is much like Netmap [15], with both frameworks evolving in the same way, adding support for virtualization and inter-process communication. PF_RING ZC has also backward compatibility for non-modified drivers, but provides modified drivers for a few devices. The user can choose to detach an interface from the normal kernel stack or not. As opposed to Netmap, PF_RING ZC supports huge pages and per-NUMA

[1]When not mentioned explicitly, *the kernel* refers to Linux.

Framework	Packet_mmap	PacketShader I/O	Netmap	PF_RING ZC	DPDK	OpenOnload
Zero-copy	~	N	Y	Y	Y	Y
Kernel bypass	N	Y	Y	Y	Y	Y
I/O Batching	Y	Y	Y	Y	Y	Y
Hardware multi-queue support	N	Y	Y	Y	Y	Y
Devices family supported	ALL	1	8 ZC / ALL (non-ZC)	4 ZC / ALL (non-ZC)	11	All SolarFlare
Pcap library	Y	N	Y	Y	Y	Y
License	GPLv2	GPLv2	BSD	Proprietary	BSD	Proprietary
IXGBE version	Last	2.6.28	Last	Last	Last	N/A

Table 1: I/O Frameworks features summary.

node buffer regions, allowing to use buffers allocated in the same NUMA node than the NIC.

A major difference is that PF_RING ZC is not free while Netmap is under a BSD-style license. The library allows 5 minutes of free use for testing purpose, allowing us to do the throughput comparison of section 4 but no further testing. Anyway, the results of this paper should be applicable to PF_RING DNA/ZC.

DPDK. The Intel Data Plane Development Kit [10] is somehow comparable to Netmap and PF_RING ZC, but provides more userlevel functionalities such as a multi-core framework with enhanced NUMA-awareness, and libraries for packet manipulation across cores. It also provides two execution model: a *pipeline* model where typically one core takes the packets from a device and give them to another core for processing, and a *run-to-completion* model where packets are distributed among cores using RSS, and processed on the core which also transmits them.

DPDK can be considered more than just an I/O framework as it includes a packet scheduling and execution model. However, DPDK exhibits fewer features than the Click modular router.

DPDK originally targeted, and is thus optimized for, the Intel platform (NICs, chipset, and CPUs), although its field of applicability is now widening.

OpenOnload [22] is comparable to DPDK but made by SolarFlare, only for their products. It also includes a user-level network stack, allowing to quickly accelerate existing applications.

We do not consider OpenOnload further in this paper, because we do not have access to a SolarFlare NIC.

Table 1 summarize the features of the I/O frameworks we consider.

3. PURE I/O FORWARDING EVALUATION

For testing the I/O system we used a computer running Debian GNU\Linux using a 3.16 kernel on an Intel Core i7-4930K CPU with 6 physical cores at 3.40 GHz, with hyper-threading enabled [9]. The motherboard is an Asus P9X79-E WS [2] with 4×4 GB of RAM at 1.6 GHz in Quad-Channel mode. We use 2 Intel (dual port) X520 DA cards for our performances tests. Previous experiments showed that those Intel 82599-based cards cannot receive small packets at linerate, even with the tools from framework's author [8, 19]. Experiments done for figure 1 lead to the same conclusion, our system seems to be capped to 33 Gbps with 64-byte packets.

To be sure that differences in performances is due to changes in the tested platform, a Tilera TileENCORE Gx36 card fully able to reach linerate in both receive and transmit

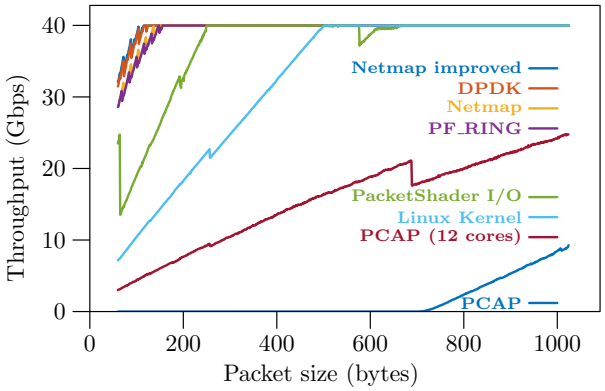

Figure 1: Forwarding throughput for some I/O frameworks using 4 cores and no multi-queueing.

side was used. We used a little software of our own available at [4] to generate packets on the Tilera at linerate towards the computer running the tested framework connected with 4 SFP+ DirectAttach cables, and count the number of packets received back. All throughput measurements later in this paper indicate the amount of data received back in the Tilera, 40 Gbps meaning that no loss occurred, and a value approaching 0 that almost all packets were lost. We start counting after 3 seconds to let the server reach stable state and compute the speed for 10 seconds. The packets generated have different source and destination IP addresses, to enable the use of RSS when applicable.

For these tests there is no processing on the packets: packets turning up on a specific input device are all forwarded onto a pre-defined, hardwired output device. Each framework has been tuned for its best working configuration including batch size, IRQ affinity, number of cores, thread-pinning and multi-queue. All forwarding tests were run for packet sizes varying from 60 to 1024 bytes, excluding the 4 bytes of the Frame Check Sequence appended at the end of each Ethernet frame by the NIC.

We realized that our NICs have a feature whereby the status of a transmit packet ring can be mirrored in memory at very low performance cost. We modified Netmap to exploit this feature and also limited the release rate of packets consumed by the NIC to only recover buffer for sent packets once every interrupt, which released the PCIe bus of useless informations and brought Netmap performances above DPDK as we can see in figure 1. For 64-byte packets, theses improvements boost the throughput by 14%, 1.5% over DPDK. However, except for the line labeled "Netmap Improved" in figure 1, only the final evaluation in section 6 use this improved Netmap version.

As expected (because they share many similarities), PF_RING has performance very similar to (standard) Netmap as shown in figure 1.

The wiggles seen in DPDK, Netmap and PF_RING are due to having two NICs on the same Intel card and produce a spike every 16 bytes. When using one NIC per card with 4 PCIe cards the wiggles disappears and performance is a little better. The wiggles have a constant offset of 4 bytes with PF_RING, but we couldn't explain it, mainly because PF_RING sources are unavailable.

PacketShader I/O is slower because of its userspace copy, while the other frameworks that use zero-copy do not even touch the packet content in theses tests and do not pay the price of a memory fetching.

We also made a little Linux module available at [4] which transmits all received packets on an interface back to the same interface directly within the interrupt context. It should show the better performances that the kernel is able to provide in the same 4 cores and no multi-queue conditions. The relative slowness of the module regarding the other frameworks can be explained by the fact the receive path involves the building of the heavy skbuff and the whole path involves multiple locking, even if in our case no device is shared between multiple cores.

PCAP shows very poor performances because it does not bypass the kernel like the other frameworks and, in addition to the processing explained for the Linux module, the packet needs to go through through the kernel forward information base (FIB) to find its path to the PCAP application. But the bigger problem is that it relies on the kernel to get packets, and each IRQ cause too much processing by the kernel, which is overwhelming a single core and does not let enough time for any thread actually consuming the packets to run, even with NAPI[2] enabled.

This is known as a receive livelock and [20] shows that beyond a certain packet rate the performance drops. The solution is either to use a polling system like in FreeBSD [18] or DPDK, or reduce the "per-packet" cost in the IRQ routines like in Netmap where it does very little processing (e.g. flags and ring buffer counter updates), or to distribute the load using techniques like multi-queue and RSS to ensure that each core receives fewer packets than the critical livelock threshold. Our kernel module is not subject to the receive livelock problem because the forwarding of the packet is handled in the IRQ routine which does not interrupt itself.

To circumvent the livelock problem seen with PCAP, we used the 12 hyper-threads available on our platform and 2 hardware queue per device to dispatch interrupt requests on the first 8 logical cores, while 4 PCAP threads forwards packets on their own last 4 logical cores. The line labeled "PCAP (12 cores)" shows the results of that configuration which gave the best results we could achieve out of many possible configurations exchanging the number of cores to serve the IRQ and the PCAP threads.

However this setup is using all cores at 100% and still provides performances way below the previous frameworks which achieve greater throughput even when using only one core.

[2]NAPI, which stands for "New API" is an interface to network device drivers designed for high-speed networking via interrupt mitigation and packet throttling [21].

4. I/O INTEGRATIONS IN CLICK

We chose the Click modular router to build a fast userspace packet processor. We first compare several systems integrating various I/O frameworks with either a vanilla Click, or an already modified Click for improved performance.

In Click, packet processors are built as a set of interconnected processing elements. More precisely, a Click task is a schedulable chain of elements that can be assigned to a Click thread (which, in turn, can be pinned to a CPU core). A task always runs to completion, which means that only a single packet can be in a given task at any time. A Click forwarding path is the set of elements that a packet traverses from input to output interfaces, and consists of a pipeline of tasks interconnected by queues.

While Click itself can support I/O batching if the I/O framework exposes batching (a FromDevice element can pull a batch of packets from an input queue), the vanilla Click task model forces packet to be processed individually by each task, with parallelism resulting from the chaining of several tasks.

Also, vanilla Click uses its own packet pool, copying each packet to and from the buffers used to communicate with the interface (or hardware queue). As such, even if the I/O framework supports zero-copy, vanilla Click imposes two memory-to-memory copies.

For our tests, we use the following combinations of I/O framework-Click integration:

- Vanilla Click + Linux Socket: this is our off the shelf baseline configuration. The Linux socket does not expose batching, so I/O batching is not available to Click.

- Vanilla Click + PCAP: while PCAP eliminates the kernel to userspace copy by using packet_mmap, Click still copies the packets from the PCAP userspace buffer into its own buffers. However, PCAP uses I/O batching internally, only some of the library calls from Click produce a syscall.

- Vanilla Click + Netmap: as netmap exposes hardware multi-queues, these can appear as distinct NICs to Click. Therefore multi-queue configuration can be achieved by using one Click element per hardware queue. Netmap exposes I/O batching, so Click uses it.

- DoubleClick [11]: integrates PacketShader I/O into a modified Click. The main modification of Click is the introduction of compute batching, where batches of packets (instead of individual packets) are processed by an element of a task, before passing the whole batch to the next element. PacketShader I/O exposes I/O batching and supports multi-queueing.

- Kernel Click: To demonstrate the case for userspace network processing, we also run the kernel-mode Click. We only modified it to support the reception of interrupts to multiple cores. Interrupt processing (creating a skbuff for each incoming packets) is very costly and using multiple hardware queues pinned to different cores allows to spread the work. Our modification has been merged in the mainline Click [3].

Kernel Click had a patch for polling mode, but it's only for e1000 driver and only supports very old kernels which prevent our system from running correctly.

Table 2 summarizes the features of these I/O framework integrations into Click.

We ran tests for pure packet forwarding, similar to those in section 2, but through Click. Each packet is taken from the input and transmitted to the output of the same device. The configuration is always a simple FromDevice pushing packets to a ToDevice. These two elements must be connected by a queue, except in DoubleClick where the queue is omitted (and thus the FromDevice and ToDevice elements run in the same task) because PacketShader I/O does not support the select operation. As a result, the ToDevice in DoubleClick cannot easily check availability of space in the output ring buffer, while the FromDevice continuously polls the input ring buffer for packets. As soon as the FromDevice gets packets, these are thus completely processed in a single task.

While this scenario is somewhat artificial, it does provide baseline ideal (maximum) performance.

In all scenarios, corresponding FromDevice and ToDevice are pinned to the same core. The results are shown in figure 2. The top two lines, labelled FastClick, should be ignored for the moment. For this simple forwarding case, compute batching does not help much as the Click path consists of a very small pipeline and the Netmap integration already takes advantage of I/O Batching. Therefore Netmap closely follows DoubleClick.

The in-kernel Click, the integration of Click with PCAP and the one using Linux Socket all showed the same receive livelock behaviour than explained in section 3. The same configurations where interrupt requests (IRQ) are dispatched to 8 logical cores and keep 4 logical cores to run Click lead to the best performances for those 3 frameworks.

The in-kernel Click is running using kernel threads and is therefore likely to receive livelock for the same reason than the PCAP configuration in section 3. The interrupts do less processing than for sockets because they do not pass through the forward information base (FIB) of the kernel but still create heavy skbuff for each packet and call the "packet handler" function of Click with a higher priority than Click's thread themselves, causing all packet to be dropped in front of Click's queue while nearly never servicing the queue consumer.

We also tested a router configuration similar to the standard router from the original Click paper, changing the ARP encapsulation element into a static encapsulation one. Each interface represents a different subnetwork, and traffic is generated so that each interface receives packets destined equally to the 3 others interfaces, to ensure we can reach linerate on output links. As the routing may take advantage of flows of packets, having routing destination identical for multiple packets, our generator produces flows, that is packets bearing an identical destination, of 1 to 128 packets. The probability of a flow size is such that small flows are much more likely than large flows (fig. 3).

Results are shown in figure 4. We omit the PCAP and socket modes as their performance is very low in the forwarding test. Additionaly, we show the Linux kernel routing functionality as a reference point. Again, ignore the lines labelled FastClick for now.

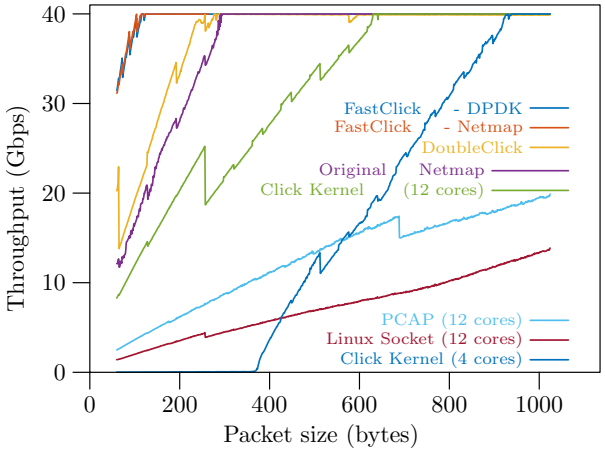

Figure 2: Forwarding throughput for some Click I/O implementations with multiples I/O systems using 4 cores except for integration heavily subject to receive livelock.

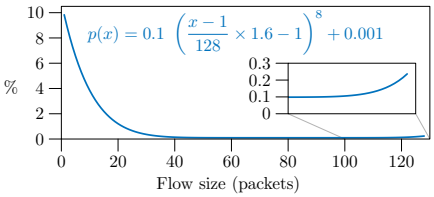

Figure 3: Probability of having flow of 1 to 128 packets for the router packet generator.

DoubleClick is faster than the Netmap integration in Click, owing to its compute batching mode and its single task model. They are both faster than the Linux native router as it does much more processing to build the skbuffs and go through the FIB than Click which does only the minimal amount of work for routing. The Kernel-Click is still subject to receive livelock and is slower than the native kernel router when routing is done on only 4 cores. Even when using 12 cores, Kernel-Click is slower than DoubleClick and the Netmap integration.

5. ANALYSIS TOWARDS FASTCLICK

We now present an in-depth analysis of the Click integrations, discussing their pros and cons. This will ultimately lead to general recommendations for the design and implementation of fast userspace packet processors. As we implement these recommendations into Click, we refer to them as FastClick for convenience.

In fact, we integrated FastClick with both DPDK and Netmap. DPDK seems to be the fastest in term of I/O throughput, while Netmap affords more fine-grained control of the RX and TX rings, and already has multiple implementations on which we can build upon and improve.

The following section starts from vanilla Click as is. Features will be reviewed and added one by one. Starting from here, FastClick is the same than vanilla Click, without compute batching, or proper support for multi-queuing.

IO Framework	Netmap Netmap	PCAP PCAP	UNIX Sockets Linux socket	DoubleClick PSIO	Kernel Linux Kernel	FastClick Netmap Netmap	FastClick DPDK DPDK
IO Batching	Y	N	N	Y	N	Y	Y
Computation Batching	N	N	N	Y	N	Y	Y
Multi-queue support	Y	N	N	Y	N	Y	Y
No copy inside Click	N	N	N	Y	N	Y	Y

Table 2: Click integrations of I/O frameworks.

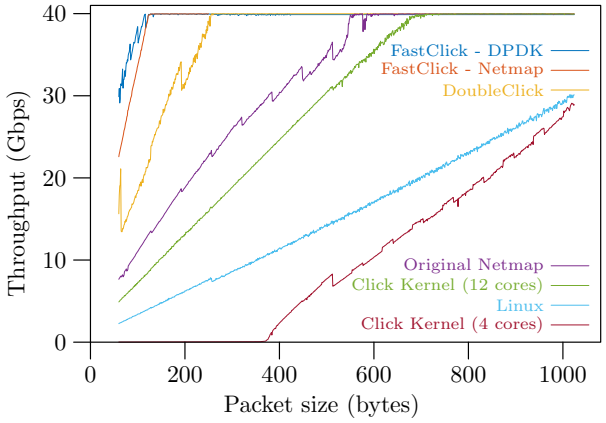

Figure 4: Throughput in router configuration using 4 cores except for in-kernel Click.

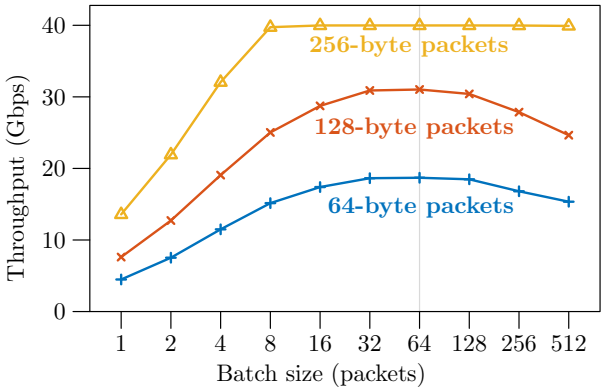

Figure 5: I/O Batching – Vanilla Click using Netmap with 4 cores using the forwarding test case (queue size = 512). A synchronization is forced after each "batch size".

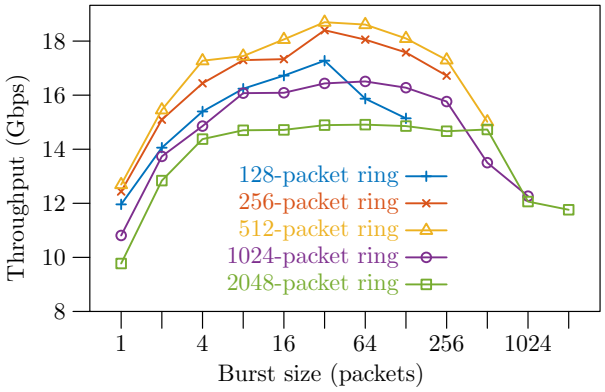

Figure 6: Influence of ring size - FastClick using Netmap with 4 cores using the forwarding test case and packets of 64 bytes.

5.1 I/O batching

Both DPDK and Netmap can receive and send batches of packets, without having to do a system call after each packet read or written to the device. Figure 5 assess the impact of I/O batching using Click in a modified version to force the synchronization call after multiple batch size in both input and output.

Vanilla Click always process all available packets before calling again Netmap's poll method – the poll method indicates how many packets are in the input queue. It reads available packets in batches and transmits it as a burst which is a series of transmission one packet at a time through a sequence of Click elements. The corresponding tasks only relinquishes the CPU at the end of the burst. Vanilla Click will reschedule the task if any packet could be received. On the other hand FastClick will only reschedule the task if a full I/O burst is available. This strategy tends to forces the use of bigger batches and thus preserves the advantages of I/O batching.

5.2 Ring size

The burst limit is there for insuring that synchronization is not done after too few packets. As such it should not be related to the ring size. To convince ourselves, we ran the same test using FastClick with multiple ring sizes and found that the better burst choice is more or less independent to the ring size as shown in firgure 6.

What was surprising though is the influence of the ring size on the performances.

With bigger ring size, the amount of CPU time spent in memcpy function to copy the packet's content goes from 4% to 20%, indicating that the working set is too big for the CPU's cache.

5.3 Execution model

In the standard Click, all packets are taken from an input device and stored in a FIFO queue. This is called a "push" path as packets are created in the "FromDevice" elements and pushed until they reach a queue. When an output "ToDevice" element is ready (and has space for packets in the output packet ring it feeds), it traverses the pipeline backwards asking each upstream element for packets. This is called a "pull" path as the ToDevice element pulls packets from upstream elements. Packets are taken from the queue, traverse the elements between the queue and the ToDevice and are then sent for transmission as shown in figure 7 (a).

One advantage of the queue is that it divides the work between multiple threads, as one thread can take care of the part between the FromDevice and the queue, and another

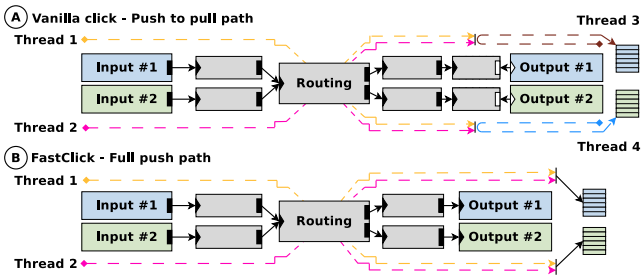

Figure 7: Push to Pull and Full Push path in Click.

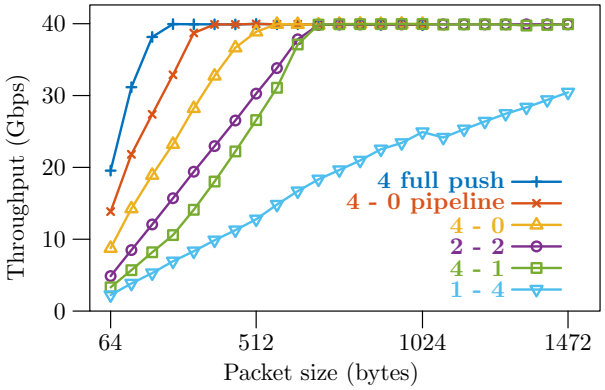

Figure 8: Comparison between some execution models. Netmap implementation with 4 or 5 cores using the router test case.

thread can handle the path from the queue to the ToDevice. Another advantage is that the queue allows the ToDevice to receive packets only when it really has space to receive packets. It will only call the pull path when it has some space and, when I/O batching is supported, for the amount of available space.

But there are two drawbacks. First, if multiple threads can write to the queue, some form of synchronization must be used between these threads, resulting in some overhead. Second, if the pushing thread and the pulling thread run on different cores, misses can occur at various levels of the cache hierarchy, resulting in a performance hit as the packets are transfered between cores.

Therefore, we adopt a model without queue: the full-push model where packets traverse the whole forwarding path, from FromDevice to ToDevice, without interruption, driven by a single thread.

NICs now possess receive and transmit rings with enough space to accommodate up to 4096 packets for the Intel 82599-based cards (not without some cost as seen in section 5.2), so these are sufficient to absorb transient traffic and processing variations, substituting advantageously for the Click queues.

Packets are taken from the NIC in the FromDevice Element and packets are pushed one by one into the Click pipe, even if I/O batching is supported. It does so until it reaches a ToDevice Element and adds it in the transmit buffer as shown in figure 7 (b). If there is no empty space in the transmit ring, the thread will either block or discard the packets.

This method is introducing 3 problems.

- All threads can end up in the same output elements. So locking must be used in the output element before adding a packet to the output ring.

- Depending on packet availability at the receive side, packets are added to the output rings. But the output ring must be synchronized sometime, to tell the NIC that it can send some packets. Syncing too often cancels the gain of batching, but syncing too sporadically introduces latency.

  DPDK forces a synch every maximum 32 packets, while Netmap does it for every chosen I/O burst size.

- When the transmit ring is full, two strategies are possible: the output has a blocking mode, doing the synchronization explained above until there is space in the output ring; in non blocking mode, the remaining packets are stored in a per-thread internal queue inside the ToDevice, dropping packets when the internal queue is longer than some threshold.

Figure 8 shows a performance comparison using the Netmap implementation with I/O batching, and a varying number of cores running FromDevice and ToDevice (the label i-j, represents i cores running the 4 FromDevice and j cores running the 4 ToDevice). The full push, where we have a FromDevice and all the ToDevice in a single thread on each core, performs best. The second best configuration corresponds to also a FromDevice and all the ToDevice running on the same core, but this time as independent Click tasks with a Click queue in between (label 4-0). Even when using 5 cores, having one core taking care of the input or the output expectedly results in a CPU constrained configuration.

While full-push mode seems best for our usage, one could need the "pipeline" mode anyway, having one thread doing only one part of the job, by using Queue element. But even in this case the full push execution model prove to be faster as shown in figure 8 by the line labeled "4 - 0 pipeline". Using our new Pipeliner element which can be inserted between any two Click elements there is no more "pull" path in the Click sense. Packets are enqueued into a per-thread queue inside the pipeliner element and it is the thread assigned to empty all the internal queues of the pipeliner element which drives the packet through the rest of the pipeline.

Full-push path is already possible in DoubleClick but only as a PacketShader I/O limitation and we wanted to study further its impact and why it proves to be so much faster by comparing the Netmap implementation with and without full push, decoupling it from the introduction of compute batching.

In vanilla Click, a considerable amount of time is spent in the notification process between the Queue element and the ToDevice. It reaches up to 60% of CPU time with 64-byte packets for the forwarding test case. With full push path, when a packet is received, it is processed through the pipeline and queued in the Netmap output ring. When the amount of packets added reaches the I/O batch size or when the ring is full, the synchronization operation explained above is called. The synchronization takes the form of an ioctl with Netmap, which updates the status of the transmit ring so that available space can be computed. This

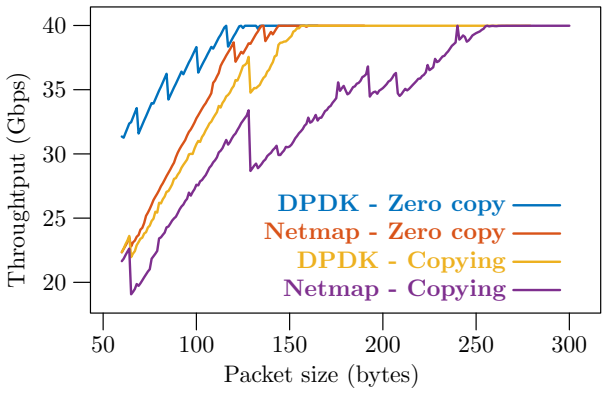

Figure 9: Zero copy influence - DPDK and Netmap implementations with 2 cores using the forwarding test case.

also allows a second improvement as the slower select/poll mechanism isn't used anymore for the transmit side, not having to constantly remove and add the Netmap file descriptor to Click's polling list anymore.

Click allows to clone packets by keeping a reference to another packet as the one containing the real data, using a reference counter to know when a data packet can be freed. In the vanilla Click, the packet can be cloned and freed by multiple cores, therefore an atomic operation has to be used to increment and decrement the reference counter. In full push mode, we know that it is always the same core which will handle the packets, therefore we can use normal increment and decrement operations instead of the atomic ones. That modification showed a 3% improvement with the forwarding test case and a 1% improvement with the router test case. FastClick automatically detect a full push configuration using the technique described in the section 5.6.

Because DPDK does not support interrupts (and it must therefore poll continuously its input), our DPDK integration only supports full-push mode.

5.4 Zero Copy

Packets can be seen as two parts: one is the packet metadata which in Click is the Packet object and is 164-byte as of today. The other part is the packet data itself, called the buffer which is 2048 bytes both for Click and Netmap. The packet metadata contains the length of the actual data in the buffer, the timestamp and some annotations to the packet used in the diverse processing functions.

In the vanilla Click, a buffer space is needed to write the packet's content, but allocating a buffer for each freshly received packets with malloc() would be very slow. For this reason, packets are pre-allocated in "pools". There is one pool per thread to avoid contention problems. Pools contain pre-allocated packet objects, and pre-allocated buffers. When a packet is received, the data is copied in the first buffer from the pool, and the metadata is written in the first packet object. The pointer to the buffer is set in the packet object and then it can be sent to the first element for processing.

Netmap has the ability to swap buffer from the receive and transmit rings. Packets are received by the NIC and written to a buffer in the receive ring. We can then swap that buffer

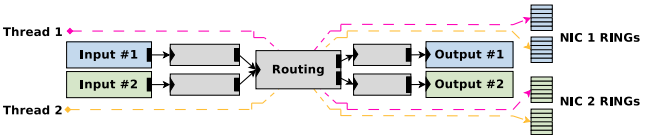

Figure 10: Full push path using multi-queue.

with another one to keep the ring empty and do what we want with the filled buffer. This is useful as some tasks such as flow reconstruction may need to buffer packets while waiting for further packet to arrive. By allocating a number of free buffers and swapping a freshly received packet with a free buffer, we can delay processing of the packet while not keeping occupied slots in the receive ring. This also allows to swap buffers with the transmit ring, allowing "zero-copy" forwarding, as a buffer is never copied.

We implemented an ioctl using the technique introduced in SNAP [23] to allocate a number of free buffers from the Netmap buffer space. A pool of Netmap buffers is initialized, substituted for the Click buffer pool. When a packet is received, the Netmap buffer from the receive ring is swapped for one of the buffers from the Click buffer pool.

DPDK directly provide a swapped buffer, as such we can directly give it to the transmit ring instead of copying its content.

With Netmap, the update of the transmit ring is so fast that output ring is nearly always full, and the ioctl to sync the output ring is called too often, especially its part to reclaim the buffers from packets sent by the NIC which is very slow and is forced to be done when using the ioctl. Instead, we changed two lines in Netmap to call the reclaim part of the ioctl only if a transmit side interrupt has triggered. We set Netmap's flag "NS_REPORT" one packet every 32 packets to ask for an interrupt when that packet has been sent.

We used the forwarding test case as our first experiment, with only two cores to serve the 4 NICs to ensure that the results are CPU-bound, and that better performance is indeed due to a better packet system. The results are shown in figure 9. The test case clearly benefits from zero-copy, while copy mode produces more important drops in the graph as one byte after the cache line size forces further memory accesses.

5.5 Multi-queueing

Multi-queueing can be exploited to avoid locking in the full-push paths. Instead of providing one ring for receive and one ring for transmit, the newer NICs provide multiple rings per side, which can be used by different threads without any locking as they are independent as shown in figure 10.

We compared the full push mode using locking and using multiple hardware queues both with DPDK and Netmap, still with 4 cores. Netmap cannot have a different number of queues in RX and TX, enabling 4 TX queues forces us to look for incoming packets across the 4 RX queues. The results are shown in figure 11.

The evaluation shows that using multiple queues is slower than locking using Netmap, but provides a little improvement with DPDK. With Netmap, augmenting the number of queues produces the same results than augmenting the number of descriptors per rings, as seen in section 5.2. Both ends up multiplying the total number of descriptors, aug-

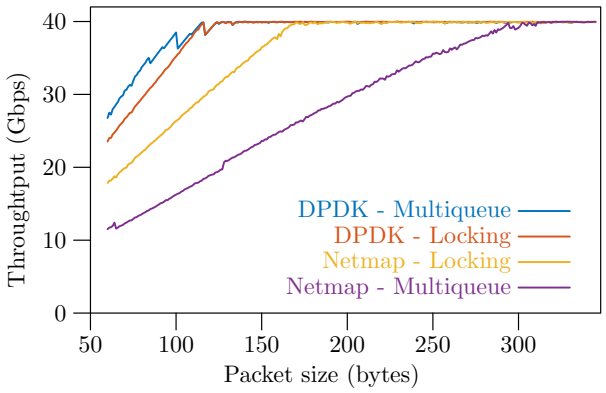

Figure 11: Full push path using multi-queue compared to locking - DPDK and Netmap implementations with 4 cores using the router test case.

menting the size of Click's working set to the point where it starts to be bigger than our CPU's cache. As we use zero-copy, we see that the cost of reading and writing from and to Netmap's descriptors goes up with the number of queues.

5.6 Handling mutable data

In the vanilla Click one FromDevice element takes packets from one device. As show in section 3, handling 10 Gbps of minimal-size packets on one core is only possible with a fast framework to deliver quickly packets to userspace and a fast processor. And even in this configuration, any processing must be delegated to another core as the core running the FromDevice is nearly submerged by the reception. A solution is to use multi-queueing, not only to avoid locking like for the full push mode, but to exploit functionality such as Receive Side Scaling (RSS) which partitions the received packets among multiple queues, and therefore enables the use of multiple cores to receive packets from the same device. However, packets received in different hardware queues may follow the same Click path. The question is thus how to duplicate the paths for each core and how to handle mutable state, that is, per-element data which can change according to the packets flowing through it, like caches, statistics, counters, etc. In figure 12, the little dots in Routing elements represent per-thread duplicable meta-data, like the cache of a last seen IP route, and the black big dot is the data which should not be duplicated because either it is too big, or it needs to be shared between all packets (like an ARP Table, a TCP flow table needing both direction of the flow, etc).

In a multi-queue configuration, there will be one FromDevice element attached to one input queue. Each queue of each input device is handled by different cores (if possible), otherwise some queues are assigned to the same thread. The problem is that in most cases, the Click paths cross at one element that could have mutable data.

A first solution is to use thread-safe elements on the part of the path that multiple threads can follow as in figure 12 (a). Only mutable data is duplicated, such as the cache of recently seen routes per cores but not the other non-mutable fields of the elements such as the Forward Information Base (FIB). The advantage of this method is that in some cases memory duplication is too costly, for example in the case of

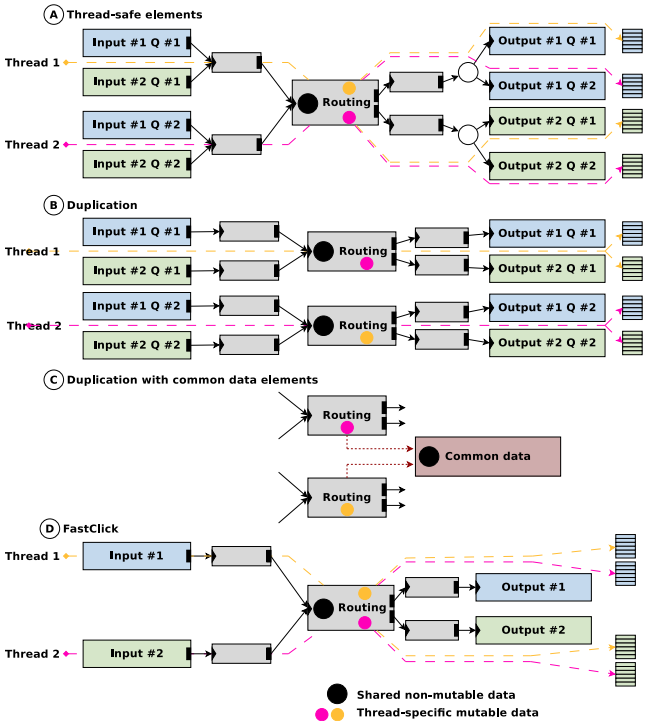

Figure 12: Three ways to handle data contention problem with multi-queue and our solution.

a big FIB, although the corresponding data structure must become thread safe. Moreover, the operator must use a special element to either separate the path per-thread to use one output queue for each thread, or use a thread-safe queue and no multi-queue.

This is in contrast to the way SNAP and DoubleClick approach the issue: the whole path is completely duplicated, as in figure 12 (b).

A third solution would be to duplicate the element for each thread path with a shared pointer to a common unmutable data like in figure 12 (c). But that would complicate the Click configuration as we would need to instantiate one element (let's say an IP router) per thread-path, each one having their own cache, but pointing to a common element (the IP routing table).

We prefer to go back to the vanilla Click model of having one Element per input device and one per output device. In that way the user only cares about the logical path. Our FastClick implementation takes care of allocating queues and threads in a NUMA-aware way by creating a Click task for each core allocated for the input device, without multiplying elements. It transparently manage the multi-queueing by assigning one hardware queue per thread as in figure 12 (d) so the operator do not need to separate path according to threads or join all threads using a queue as in figure 12 (a).

Of course, the user can still specify parameter to the FromDevice to change the number of assigned threads per input device. He can also force each thread to take care of one queue of each device to allow the same scheme than in figure 12 (a) but still with one input element and one output element per device. The difference between the two

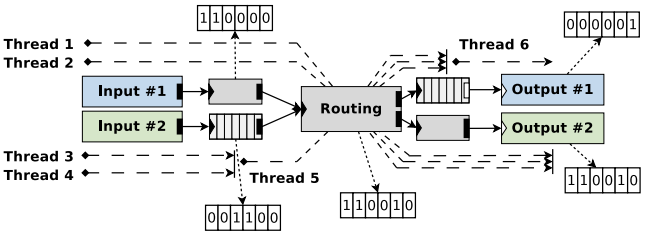

Figure 13: Thread bit vectors used to know which thread can pass through which elements.

configurations depends mostly on the use case. Having each core handling one queue of each device allows more load-balancing if some NICs have less traffic than others, but if the execution paths depend strongly on the type of traffic, it could be better to have one core doing always the same kind of traffic and avoid instruction cache misses.

To assign the threads to the input device we do as follows: We use only one thread per core. For each device, we identify its NUMA node and count the number of devices per NUMA node. We then divide the number of available CPU cores per NUMA node by the number of devices on that NUMA node, which gives the number of cores (and thus threads) we can assign to each device. With DPDK, we use one queue per device per thread, but with Netmap we cannot change the number of receive queues(which must be equal to the number of send queues), and have to look across multiple queues with the same thread if there are too many queues.

For the output devices, we have to know which threads will eventually end up in the output element corresponding to one device, and assign the available hardware queues of that device to those threads. To do so, we added the function getThreads() to Click elements. That function will return a vector of bits, where each bit is equal to 1 if the corresponding thread can traverse this element, that is called the thread vector.

FastClick performs a router traversal at initialization time, so hardware output queues are assigned to threads.

To do so, the input elements have a special implementation of getThreads() to return a vector with the bits corresponding to their assigned threads set to 1. For most of the other elements, the vector returned is the logical OR of the vector of all their input elements, because if two threads can push packets to a same element, this element will be executed by either of these threads. Hence, this is the default behavior for an elements without specific getThreads() function.

An example is shown in figure 13. In that example, some path contains a queue where multiple threads will push packets. As only one thread takes packet from the top right queue, the output #1 does not need to be thread-safe as only one thread will push packets into it. The output #2 will only need 3 hardware queues and not 6 (which is the number of threads used on this sample system) as the thread vector shows that only 3 threads can push packets in this element. If not enough queues are available, the thread vector allows to automatically find if the output element need to share one queue among some threads and therefore need to lock before accessing the output ring.

Additionally to returning a vector, the function getThreads() can stop the router traversal if the element does not care of its input threads, such as for the queues elements. If that happens, we know that we are not in the full push mode and we'll have to use atomic operations to update the reference counter of the packets as explained in section 5.3.

We also provide a per_thread template using the thread vector to duplicate any structure for the thread which can traverse an element to make the implementation of thread-safe elements much more easier. Our model has the advantages of figure 12 (a) and (c) while hiding the multi-queue and thread management and the simplicity of solution (b).

5.7 Compute Batching

While compute batching is a well-known concept [11, 23] we revisit it in the context of its association with other mechanisms. Moreover, its implementations can differ in some ways.

With compute batching, batches of packets are passed between Click elements instead of only one packet at a time, and the element's job is done on all the packets before transmitting the batch further. SNAP and DoubleClick both use fixed-size batches, using techniques like tagging to discard packets which need to be dropped, and allocating an array for each output of a routing element as big as the input one would, leading to partially-filled batches. We prefer to use a simply linked lists, for which support is already inside Click, and accommodate better with splitting and variable size batches. Packets can have some annotations, and there is an available annotation for a "next" Packet and a "previous" Packet used by the Click packet pool and the queuing elements to store the packets without the need of another data structure. As such, we introduce no new Packet class in Click.

For efficiency we added a "count" annotation which is set on the first packet of the batch to remember the batch size. The "previous" annotation of the first packet is set to the last packet of the batch, allowing to merge batches very efficiently.

The size of the batch is determined by the number of packets available in the receive ring. The batch which comes out of the FromDevice element is composed of all the available packets in the queue, thus only limited by the chosen I/O batching limit.

Compute batching simplifies the output element. The problem with full-push was that a certain number of packets had to be queued before calling the ioctl to flush the output in the Netmap case, or DPDK's function to transmit multiple packets that we'll both refer as the output operation. With compute batching, a batch of packets is received in the output element and the output operation is always done at the end of the sent batch. If the input rate goes down, the batch size will be smaller and the output operation will be done more often, reducing latency as packets don't need to be accumulated.

Without batching, a rate-limit mechanism had to be implemented when the ring is full to limit the call to the output operation. Indeed, in this case, the output operation tends to be called for every packet to be sent, in an attempt to recover space in the output ring. These calls of the output operation can create congestion on the PCIe, a situation to be avoided when many packets need to be sent. This problem naturally disappears when compute batching is used as the time to process the batch gives time to empty part of the output ring.

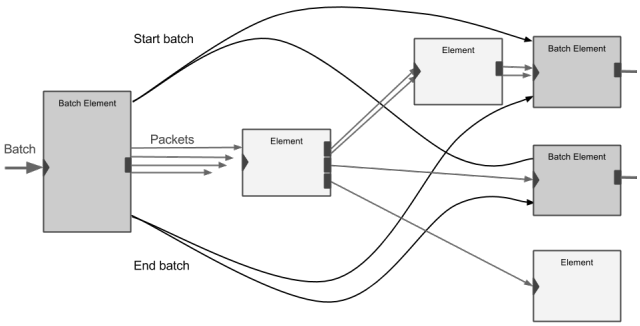

Figure 14: Un-batching and re-batching of packets when downstream elements have multiple paths.

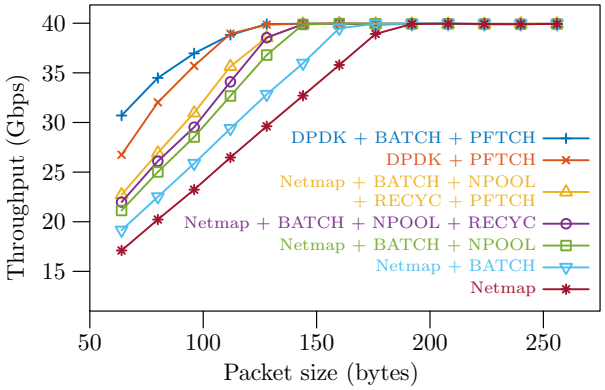

Figure 15: Batching evaluation - DPDK and Netmap implementations with 4 cores using the router test case. See section 5.7 for more information about acronyms in legend.

In both SNAP and DoubleClick, the batches are totally incompatible with the existing Click elements, and you need to use either a kind of Batcher/Debatcher element (SNAP) or implement new compatible elements. In our implementation, elements which can take advantage of batching inherit from the class BatchElement (which inherit from the common ancestor of all elements "Element"). Before starting the router, Click makes a router traversal, and find BatchElements interleaved by simple Elements. In that case the port between the last BatchElement and the Element will unbatch the packets and the port after the last Element will re-batch them. As the port after the Element cannot know when a batch is finished, the first port calls start_batch() on the downstream port and calls end_batch() when it has finished unbatching all packets. When the downstream port receives the end_batch() call, it passes the reconstructed Batch to their output BatchElement. Note that the downstream port use a per-thread structure to remember the current batch, as multiple batches could traverse the same element at the same time but on different threads.

When a simple (non-batching) element has multiple output, we apply the same scheme but we have to call the start_batch() and end_batch() on all possible directly reachable BatchElement as shown in figure 14. This list is also found on configuration time.

For elements in the fast path, it's important to implement a support for batching as the cost of un-batching and re-batching would be too big. But for elements in rarely used path such as ICMPError or ARP elements, the cost is generally acceptable and preferable over development time of elements taking full advantage of compute batching.

Click keeps two pools of objects: one with Packet objects, and one with packet buffers. The previous version of Click and SNAP way of handling a freshly available Netmap packet is to take a packet from the Packet object pool and attach to the filled Netmap buffer from the receive ring. A Netmap buffer from a third pool of free Netmap buffers is then placed back in the receive ring for further reception of a new packet. SNAP does the buffer switching only if the ring is starting to fill up, maintaining multiple type of packets in the pipeline which can return to its original ring or to the third pool; introducing some management cost. We found that it was quicker to have only Netmap buffers in the buffer pool and completely get rid of Click buffer if Click is compiled with Netmap support as the allocate/free of Netmap buffer is done very often. It is very likely that if Netmap is used, packets will be either received or sent from/to a Netmap device. This is called the Netmap pool and is labelled "NPOOL" in figure 15.

Even if Netmap devices are not used, if there is not enough buffers in the pool, we can expand the pool by calling the same ioctl than in section 5.4 to receive a batch of new Netmap buffers and allocate the corresponding amount of Packets. Moreover, our pool is compatible with batching and using the linked list of the batches, we can put a whole batch in the pool at once as we have only one kind of packets, this is called per-batch recycling and is labelled "RECYC" in figure 15.

We do not provide the same functionality in our DPDK implementation. As DPDK always swaps the buffer with a free buffer from its mbuf pool when it receives a packet, we do not need to do it ourself. We simply use the Click pool to take Packet objects and assign them a DPDK buffer.

The results of the router experiment with and without batching for both DPDK and Netmap implementations are shown in figure 15. The "BATCH" label means that the corresponding line uses batching. The "PFTCH" label means that the first cacheline of the packet is prefetched into the CPU's cache directly when it is received in the input elements. When a packet reaches the "Classifier" element which determine the packet type by reading its Ethernet type field and dispatch packets to different Click paths, the data is already in the cache thanks to prefetching, allowing another small improvement. We omit the forwarding test case because the batching couldn't improve the performance as it doesn't do any processing.

6. FASTCLICK EVALUATION

We repeated the experiments in section 4 with our FastClick implementation. The results of the forwarding experiments are shown in figure 2, and those of the routing experiment in figure 4.

Both Netmap and DPDK FastClick implementations remove the important overhead for small packets, mostly by removing the Click wiring cost by using batches and reducing the cost of the packet pool, using I/O batching and the cost of the packet copy compared to vanilla Click.

The figures do not show an important part of the novelty which is also in the configuration, which becomes much simpler (from 1500 words to 500, without any copy-paste), autoconfigured according to NUMA nodes and available CPUs, and the compatibility of pipeline elements with batches.

7. CONCLUSION

We have carried out an extensive study of the integration of packet processing mechanisms and userspace packet I/O frameworks into the Click modular router.

The deeper insights gained through this study allowed us to modify Click to enhance its performance. The resulting FastClick system is backward compatible with vanilla Click elements and was shown to be fit for purpose as a high speed userspace packet processor. It integrates two new sets of I/O elements supporting Netmap and Intel's DPDK.

The context of our work is network middleboxes: packets are received, maybe dropped or modified and then sent through another interface. It may be less suited to scenarios where data is created from scratch in userspace and sent out (as in the case of a server scheme where requests are received as a small number of packets and generate a much bigger number of packets at the output).

Beyond improved performance, FastClick also boasts improved abstractions for packet processing, as well as improved automated instantiation capabilities on modern commodity systems, which greatly simplifies the configuration of efficient Click packet processors.

As many middleboxes operate on the notion of microflows, we will extend, as future work, the flow capabilities of FastClick to facilitate the development of such middleboxes.

8. REFERENCES

[1] K. Argyraki, S. Baset, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, E. Kohler, M. Manesh, S. Nedevschi, and S. Ratnasamy. Can software routers scale? In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 21–26, New York, NY, USA, 2008. ACM.

[2] ASUS. P9x79-e ws. `http://www.asus.com/Motherboards/P9X79E_WS/`.

[3] T. Barbette. Click pull request #162 to enable multi-producer single-consumer mode in linuxmodule fromdevice. `https://github.com/kohler/click/pull/162`.

[4] T. Barbette. Tom barbette's research part. `http://www.tombarbette.be/research/`.

[5] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association.

[6] 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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[7] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, page 20. ACM, 2008.

[8] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[9] Intel. Core^{TM} i7-4930k processor (12m cache, up to 3.90 ghz). `http://ark.intel.com/products/77780`.

[10] Intel. Data plane development kit. `http://www.dpdk.org`.

[11] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon. The power of batching in the click modular router. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 14:1–14:6, New York, NY, USA, 2012. ACM.

[12] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[13] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, page 9. ACM, 2013.

[14] Linux Kernel Contributors. Packet_mmap. `https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt`.

[15] ntop. Dna vs netmap. `http://www.ntop.org/pf\_ring/dna-vs-netmap/`.

[16] ntop. Pf_ring. `http://www.ntop.org/products/pf\_ring/`.

[17] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[18] L. Rizzo. Device polling support for freebsd. In *BSDConEurope Conference*, 2001.

[19] L. Rizzo. netmap: A novel framework for fast packet i/o. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[20] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet i/o in virtual machines. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 47–58. IEEE Press, 2013.

[21] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet.

[22] Solarflare. Openonload. `http://www.openonload.org/`.

[23] W. Sun and R. Ricci. Fast and flexible: Parallel packet processing with GPUs and Click. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Oct. 2013.