TLEM, very high speed link emulation

Luigi Rizzo, Giuseppe Lettieri
Dipartimento di Ingegneria dell'Informazione
Università di Pisa
{rizzo,g.lettieri}@iet.unipi.it

Abstract

In this work we discuss the limitations of link emulators based on conventional network stacks, and present our alternative architecture called **TLEM**, which is designed to address current high speed links and be open to future speed improvements. **TLEM** is structured as a pipeline of stages, implemented with separate threads and with limited interactions with each other, so that high performance can be achieved. Our emulator can handle bidirectional traffic at speeds of over 18 Mpps (64 byte packets) and 30 Gbit/s (1500 byte packets) per direction even with large emulation delays. Even higher performance can be achieved with shorter delays, as the workload fits better into the L3 cache of the system.

TLEM runs on any system that supports netmap (this includes FreeBSD, Linux and now even Windows hosts). The code is available as open source under a BSD license at `github.com:luigirizzo/tlem` .

1 Introduction

Link and network emulators are hardware or software systems that manipulate network traffic in ways similar to the real devices they emulate: traffic is subject to queuing, bandwidth limitations, delay, and possibly classification and scheduling. Figure 1 shows the basic features (queue size, bandwidth and delay) that are implemented by link emulators. These can be constant or variable, both in deterministic and probabilistic ways.

Solutions for link emulation have been embedded in commodity operating systems for almost twenty years. The most relevant examples include `dummynet` [8, 1], which is available for all major operating systems (FreeBSD, Linux, OS/X and Windows), and `netem` [4], which is Linux-only. Having the emulator within the OS eases experiments, as they can be run with real traffic sources/sinks, and potentially even without having a real network. The negative aspect of this approach is that performance can be limited by the OS' network stack, which is often unable to deal with the extreme packet rates that are possible on 10 Gbit/s and faster networks.

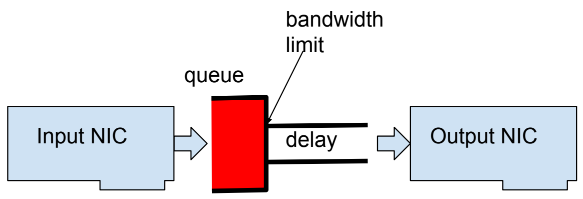

Figure 1: The basic operation implemented by a link emulator.

Having recently developed solutions for high speed packet I/O, we have been confronted with the problem of building high speed link emulators. In this paper we discuss why the problem cannot be simply approached by replacing the network I/O layer in existing emulators, and propose an alternative design that we developed to build a very high speed link emulator called **TLEM**.

Our system, available as open source under a BSD license, uses a pipeline of stages, each assigned to a separate core, to achieve high performance. The pipelined architecture addresses in an elegant way one of the key problems in scaling network appliances, namely the preservation of packet ordering. A mixed blocking/busy wait architecture inspired to solutions used in Virtual Machine communication permits keeping latency under control while achieving high packet rates and low energy consumption.

TLEM achieves bidirectional delay emulation at speeds of over 18 Mpps with short frames, and over 30 Gbit/s with 1500-byte frames. These speeds exceed the capacity of 10 Gbit/s interfaces, even with minimum packet sizes, and are almost 20 times higher than the throughput of a basic dummynet instance. While not including all features of dummynet, **TLEM** is actually easier to extend to implement more complex or custom features such as time-based transmissions, empirical or trace-driven delay emulation, packet mangling, classification, scheduling.

2 Background

In this Section we describe the features and performance of legacy link emulators embedded in commodity operating systems, discuss performance issues and report experimental results in accelerating our dummynet emulator with a faster network I/O framework.

2.1 Legacy link emulators

Almost twenty years ago, after some early experiment with custom solutions and dedicated hardware boxes, the quest for link emulators has been effectively addressed with our seminal work on dummynet [8], which showed how in-system link emulation can be achieved in simple and effective ways. Especially, its integration in FreeBSD [9] made the tool readily available to a large number of researchers, who used it in countless research projects, testbeds such as Emulab [12] and Planetlab [3, 2], as well as in ISPs and commercial deployments. On the Linux side, netem [4] and the traffic shaper tc [5] have become popular, also favoured by a much larger popularity of the operating system.

The advantage of an emulator embedded in the operating system are immense, in terms of ease to use, flexibility, even availability. Experiments do not require any special setup other than configuring, with OS commands, the desired features of the underlying network.

On the negative side, the very same location of the emulator raises the bar when it comes to modify or extend its features. Kernel components are generally fragile, and the environment offers limited support for features such as floating point computations, logging, and crash handling. We have tried to compensate these limitations for dummynet by doing periodic overhauls of the code, adding functionality such as support for programmable schedulers [1], enhanced link emulation [2], and working on improving performance. Part of our support effort was also dedicated to make dummynet available on other platforms, such as Linux and Windows (the OSX version came by itself when Apple decided to base its operating system upon FreeBSD). This work contributed largely to improving dummynet's availability and usefulness.

Nevertheless, we must acknowledge that the bar to build and include extensions to the emulator has been quite high, and very little third party code has made its way into dummynet. Similar considerations apply to netmem and tc.

2.2 Performance

On the performance side, dummynet and netem both suffer from the constraints of the environment in which they run. The use of the OS's network stack for packet representation and I/O makes their performance (and that of the rest of the network stack) unsuitable for the packet rates produced by the fastest NICs at any point in time. When 1 Gbit/s NICs became widespread, the PCI bus and single core, low speed CPUs of the time were preventing good performance. As CPU and bus speeds improved, and additional cores become available, NIC speeds bumped up to 10 Gbit/s and more, leaving the gap unchanged. This time, the bottleneck is less of a hardware problem (bus and CPU speeds, and core counts, are more than adequate) but a software one.

As a data point, our measurements on the 2010 version of dummynet [1, Sec.4.2] show how it can process about 0.5 Mpps. Other systems exhibit similar performance. Network I/O takes a significant part of the per-packet processing time, but it is not the only culprit, as we will discuss in the next Section.

2.3 Network I/O performance

As widely discussed elsewhere [10], network I/O subsystems for commodity operating were designed almost 30 years ago, under a set of constraints (CPU number and speed, memory size and speed, protocol features) very different from today. Retrofitting the software on modern architectures (which can exploit parallelism, have fast and large memories but with high latency) and supporting a variety of "hardware offloading" features on the network cards (such as checksums, TCP segmentation and reassembly, VLANs and encapsulation) only provided a small fraction of the speedup that one could expect from the difference in hardware speeds.

Acknowledging the I/O performance issue, a substantial amount of work has been performed in the past years to provide efficient mechanisms for network I/O, even though with somewhat specific goals such as software switching, and traffic capture and generation. High performance APIs such as netmap [11] and DPDK [6] have addressed in a successful way high packet rate applications, and opened the way to the application of certain performance improvement ideas (batching, streamlined processing, etc.) also to the ordinary network stack.

Link emulators, at least those that interconnect two network interfaces[1] as shown in Figure 1, are good candi-

[1] another use of link emulators such as dummynet is to have one of their ports connected to the host stack. In this case the need for

dates to use the fast I/O frameworks mentioned above. In fact, replacing the network I/O component with a faster one definitely removes one of the heaviest cost components from the emulator. How much this contributes to performance must be determined experimentally, which is what we do in the next Section.

2.4 Early experiment: netmap-ipfw

Our first attempt at high speed emulation, in 2012, was to run the dummynet code on top of netmap. For this project, called `netmap-ipfw` [7], we needed to port to user space the entire kernel code for `dummynet` and its associated packet classifier, `ipfw`.

The port was done by providing a set of functions to replicate some kernel functionality in user space (memory allocation, module management, timers, sysctl and ioctl/sockopt). Clearly, the network I/O path was replaced by calls to the netmap framework; other than that, we tried to make the port with as few as possible changes to the original code. This constraint was necessary to avoid introducing new bugs in the system, or having to redesign parts of what had over time become a relatively complex system (`dummynet` supports a variety of scheduling policies, and the traffic classifier itself has tens of traffic matching options).

2.4.1 netmap-ipfw performance

Packet processing costs in `netmap-ipfw` are made of three components: network I/O (the one we replaced by netmap), packet classification, and emulation. The latter two are essentially unchanged between the two environments (kernel and userspace), although there might be some differences due to the different locking requirements. Network I/O is expected to be much faster in the userspace version, as witnessed by our measurements.

Due to the additive impact of the three components, it is easier to report performance in terms of *time per packet*, which is also an additive figure. "time per packet" should be interpreted as the inverse of the throughput ("packet per second") computed over large ($\approx$ 1 second) intervals; we could not measure these numbers in other ways, as some operations are amortised over batches of packets, and individual packets are processed in multiple phases.

The numbers reported below have been measured on a variety of modern systems with single socket Intel i7 CPUs (including i7-4790 and i7-5930k) running at 3.5-3.7 GHz. Besides CPU type and speed, memory speed and L3 cache sizes also have a deep impact on some of the measurements. In our systems, memory runs at speeds between 1333 and 2133 MHz, and L3 cache sizes vary between 6 and 12 Mbytes.

high performance is much less pressing, being limited by the slow host stack.

2.4.2 netmap I/O costs

The netmap architecture lets us connect applications to a variety of "network ports", each with different features in terms of I/O costs. In particular we have the following cases (performance figures have been determined in previous experiments):

physical interfaces (NICs) For them, read and write costs depend on the number of register accesses and interrupts needed on each packet. By amortizing these operations over sufficiently large batches, we have a total of 15-20 ns on each side on the platform used for our tests (with Intel 10 G and 40 G NICs).

VALE switch In this case a read operation is extremely cheap (in the order of 15 ns), whereas writes are more expensive, requiring the sender to perform a data copy. For short packets, writes require 40-50 ns, up to 150-200 ns for 1500-byte packets. Note that it is difficult to report these times precisely, because they are dominated by memory latency and bandwidth, which in turn depends on how overall memory accesses in the system fit within caches.

netmap pipes In this case both read and write operations only require to manipulate ring pointers and minimal locking. We can account approximately 10 ns on each side for I/O.

In the case of NICs, the throughput may be further limited by the speed of the internal controller, or by congestion on the bus (PCI or PCIe) used to connect the NIC to the system. It follows that using a physical NIC to send and receive traffic may hide the effect of optimizations in the code and give misleading results in the experiment to determine the maximum speed of the system.

Given the above considerations, we ran our test on `netmap-ipfw` by connecting the application to netmap pipes, which give clear and repeatable overheads.

2.4.3 Cost breakdown

`netmap-ipfw` runs the following operations on each packet:

1. read packet, encapsulate in pseudo-mbufs;
2. apply `ipfw` rules for filtering;
3. for packets subject to emulation, allocate a buffer, copy the packet, and copy it back into a netmap buffer when it is time to release it;
4. transmit the packet.

In our first experiment we only ran the first and last step, disabling the intermediate operations. This accounts for the basic I/O costs, and requires only 26 ns per packet without touching the payload of the packet itself (pipes are fully zero copy, and `netmap-ipfw` exploits the feature if possible). The number quickly grows to 50 ns if we try to access even just one byte of the packet, as we pay the cache miss penalty for accessing data that had been never read before.

Enabling step #2, the application of traffic selection rules, increases the processing time by an amount that depends on the complexity of the ruleset. In our experiments, simply inserting one rule requires an additional 33 ns, and extra rules take at least 6-7 ns each. Thus, the combination of I/O and traffic selection brings us in the best case to about 83 ns per packet, which is already too slow for the peak rate on 10 Gbit/s NICs.

The largest component in the processing costs is however the one related to emulation. In this case the filters must use an `ipfw` rule that makes packets go to a dummynet `pipe`, which in turn implements the emulation. Packets subject to emulation must be delayed for some time before being sent out. Implementing this feature requires some data copies for the reasons discussed below.

Netmap I/O buffers are a limited resource and a space consuming one: they have a fixed size (default 2 Kbytes, or up to over 9 Kbytes in case the network uses jumbo frames), so they can use as much as 32..140 times the size of a short packet. Considering that the delays can be in the order of seconds, and data rates reach 40 Gbits/s, the inefficient use of space can easily consume tens of Gigabytes of memory.

To address this problem, `netmap-ipfw` trades space for time, and copies packets to a temporary, custom allocated buffer before the delay. Packets are then copied back into netmap buffers when it is time to release them. These two copies, together with the cost of running through the emulation engine and the packet scheduler (which is part of a dummynet pipe), consume a significant amount of time.

In our experiments, step #3 requires about 290 ns per packet even for 64-byte packets. Adding the I/O and filtering times, this translates into a maximum throughput of 2.7 Mpps when dealing with unidirectional traffic. Bidirectional traffic would reach a much lower value, because the same core that runs the entire `netmap-ipfw` process would have to deal also with traffic in the opposite direction.

2.4.4 Considerations

All the above numbers are both reason for excitement and for depression. On the one hand, we have achieved a 5-fold speedup over the existing in-kernel implementation. On the other hand, for emulation we are still six times below the top speed of a 10 Gbit/s interface.

There are multiple reasons why `netmap-ipfw` cannot exploit the raw speed of the netmap API. Remember that `ipfw` and `dummynet` were designed and developed for an environment where network I/O was by far the main bottleneck, so not much effort had gone in addressing other, less important bottlenecks. Among them, two are particularly relevant.
First, packets are represented by linked lists of dynamically allocated buffers, which not only adds complexity at runtime, but also produces very scattered memory accesses, which in turn result in a significant number of cache misses.
Second, `netmap-ipfw` is single threaded, and processes packets one at a time, in three phases: one when enqueueing the packet, one when the packet exits the emulated link, and finally when it exits the delay line and it is time to transmit it. Data structures between the three stages are completely shared, so the in-kernel version runs under a single lock, and `netmap-ipfw` is implemented by a single thread that alternates between the three tasks as needed. Having a single thread saves much of the locking overhead, which would be significant when dealing with one packet at a time. Trying to split `netmap-ipfw` into multiple threads while keeping the locking overhead low would require significant changes to the code.

Another limitation of `netmap-ipfw`, also due to the environment in which it was originally developed, is that it operates with a relatively coarse granularity (programmable, but for practical purposes in the order of 100 μs). At 10 Gbit/s, 100 μs correspond to one megabit of data, which means that traffic is released from the emulator in large bursts.

2.5 Multiqueue solutions

It is common, when dealing with high speed networking, to decompose applications in multiple instance of `netmap-ipfw`, each operating independently on a subset of the traffic. This approach leverages hardware support in the NICs; specifically, multiqueue adapters which can partition traffic into different queues according to criteria such as configuration of MAC or IP addresses and ports.

This solution works well when there are no timing relations among the traffic on different queues – a noticeable example being implementing network servers or firewalls. In fact, several people use exactly this approach in production firewalls. Link emulators, however, need to make global decisions on the traffic, such as enforcing bandwidth limitations, and preserving the ordering of packets across all queues. As a consequence, we cannot exploit multiqueue in a link emulator.

3 TLEM, a pipelined link emulator

From the experience with `netmap-ipfw` we have learned that a single core is unable to cope with traffic at line rate on even a 10 Gbit/s interface. We thus need an architecture that can make use of the many cores available in the system. We also need to reduce the impact of cache misses, which are not adequately amortised by `netmap-ipfw` due to the processing of one packet at a time.

Based on the above considerations, we redesigned the link emulator following two main principles.
First, for extremely high speed applications, simplicity is king, so we removed all features that exist in `netmap-ipfw` but are not required for link emulation. In particular, we do not need a programmable traffic filter that uses a domain specific language such as `ipfw` rules, nor we need a general purpose traffic scheduling framework as the one present in `dummynet`.
Second, in partitioning the system across multiple cores, we avoid solutions that introduce data races or dependencies that are expensive to verify; as an example, we do not use solutions that partition traffic because they would prevent us from enforcing bandwidth limitations or preserve global packet ordering.

3.1 Pipelined architecture

The solution we adopted for our design, called **TLEM**, is a pipelined software architecture, shown in Figure 2. Each stage performs a simple task and has minimal interactions with the other ones.

Each direction is managed by a different pipeline, so we can deal with bidirectional traffic with no loss of performance, of course subject to the availability of a sufficient number of cores in the system.

Each pipeline in **TLEM** is made of at least an *input stage*, which reads incoming traffic and copies it into a *shared buffer*, and an *output stage*, which polls the buffer and releases packets at their due time. Before packets are passed to the output stage, they are annotated with their fate (drop or keep it), the time at which they can be released, and possibly altered if the emulation includes packet modifications. Depending on the amount of computation to be performed on each packet, the computations to generate annotations and packet modifications can be run in the input stage, or in additional stages in the pipeline.

Decomposing operations in a pipeline, as opposed to running multiple parallel stages, has a significant advantage in that it is inherently safe from packet reordering. Of course, to achieve a sufficient performance, we should make sure that each stage of the pipeline can cope with the expected processing rate. TLEM can introduce more stages in the pipeline to perform the computation. In the current implementation, we have found that the input and output stages can perform all the computations and exceed the speed of a 10 Gbit/s interface.

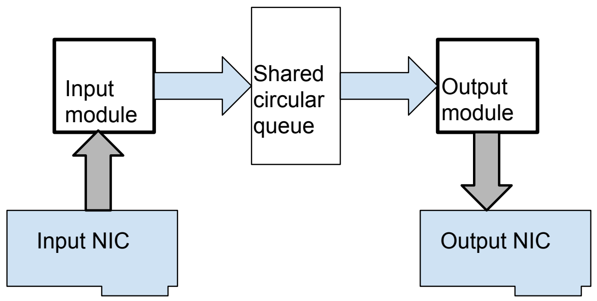

Figure 2: The architecture of **TLEM**

4 TLEM operation

In the rest of this Section we describe how **TLEM** implements its functions. Following Figure 1, we recall that the emulator must first replicate the effect of a queue attached to a link with predefined bandwidth, and then impose additional delay in delivering the packet, to model propagation delays and possibly additional effects such as further queueing in other parts of the network.

We call *link queue* the simulated queue, and *delay line* the part that emulates delay.

4.1 Network I/O

TLEM uses netmap for packet I/O, thus requiring only a small fraction of a core for communicating with the NIC, even at 10 Gbit/s and above. As mentioned in Section 2.4.1, the amortized cost per packet is between 10 and 20 ns when accessing network devices, excluding the cost of data touching operations (reads, which incur some latency on the first access; and copies, which consume CPU cycles and pollute caches). The above makes it possible for the input and output stages to perform a fair amount of useful computation.

A straightforward implementation of the emulator in Figure 1, such as the one in `dummynet`, would first put packets into a "link queue" (enforcing queue size limitations); drain the queue at a rate corresponding to the link's bandwidth, putting packets into a second queue implementing the delay line; and finally, extract packets from the delay line at their due transmission time. The above scheme is only appropriate for low speed operation. At our target speeds, enqueueing and dequeueing packets multiple times is a potential performance bottleneck that should be avoided.

The solution we use, instead, requires only a single enqueue and dequeue operation per packet. Packets coming from the input NIC are immediately[2] stored in a large circular buffer, described in Section 4.6. Each packet is preceded by a packet descriptor, which among other things contains the absolute time when the packet should be released to the output interface. This value is computed as described in Section 4.2 and 4.4, and is used by the output stage of the pipeline.

4.2 Queue and link bandwidth

Traffic shaping (emulating a link with predefined bandwidth) is a core function of any link emulator. This is normally implemented as follows. When the *i*-th packet arrives at time t_A^i, **TLEM** computes when it will exit the link, t_L^i, using the following formula:

$$t_L^i = \max(t_L^{i-1}, t_A^i) + \frac{l_i}{B}$$

where l_i is the packet's length (including framing overhead, such as preambles, inter-packet gaps, checksum) and B is the link's bandwidth. Minor modifications to the formula can be used when when the link's bandwidth is not constant. As an example, Time Division Multiplex channels let clients communicate only during periodic slots of time. Even on an idle channel, a packet arriving outside the slot must wait for the next slot to be ready, and t_L^i must be computed accordingly. Another example is that of wireless links where bandwidth may vary depending on channel conditions. In both cases **TLEM** can be configured, with user supplied C code, to emulate these and other variable bandwidth channels.

Knowledge of the exit time t_L^j for all previous packets makes it possible to determine the queue occupation (in bytes and packets) at the arrival of a new packet, and determine whether or not it should be dropped, without having to implement a separate link queue.

Link bandwidth and queue size to be used in the emulation are configured from the command line and independently for each direction.

4.3 Random packet drops

Congestion-induced drops, as those described in the previous Section, are a normal artifact of a communication network and one that **TLEM** emulates precisely. It is sometimes useful to analyse the behaviour of a system or application in presence of other types of packet drops or errors. These could be caused by noise in the communication channel, or even complex congestion situations in parts of the network that cannot be simply modeled with a queue and a link.

For this reason **TLEM**, same as many other emulators, supports random or deterministic packet dropping or errors. The actual distribution of drops (e.g., their frequency, burstiness, data dependencies) may affect the behaviour of the system under test, so it is important to provide high flexibility in defining drop patterns.

TLEM implements this feature allowing users to provide a C function that is invoked on each packet together with full information on the packet itself. The result – a yes/no answer – determines whether the packet should be dropped. The user supplied function can be stateful, thus supporting complex policies that simulate burst errors. **TLEM** also includes some predefined distributions that can be configured from the command line, and include constant packet- and bit- error rates.

4.4 Link delay

The time t_L^i computed by the input stage only indicates when the packet exits the link queue. Before being actually released by the emulator, a packet may incur further delay, normally to model the effect of its traversal of the physical link – copper, fiber, air, space. Sometimes the link delay is also used to model additional equipment downstream, including queueing delay and possibly multipath effects.

To support these features, **TLEM** must compute an additional value for each packet, t_D^i, which is added to t_L^i to determine when the packet can be released.

In the simplest case, t_D^i is constant and can be configured from the command line. Same as for random drop, however, **TLEM** lets the user provide arbitrary C code to compute the additional delay for each packet, and includes some predefined distributions. Among them we have constant delays, uniformly distributed delays within a range, or exponentially distributed delays with predefined minimum and average.

Values for non uniform distributions are computed by generating a uniformly distributed number in the range $[0,1]$ and using it as the argument to the Inverse of the Cumulative Density Function (CDF) for the distribution, to generate the desired values. When the CDF is too difficult to invert analytically, or it is empirically defined (derived by actual samples), the inverse is simply tabulated with a sufficient number of samples.

4.4.1 Reordering

TLEM imposes one restriction on link delay distributions, namely, link delays t_D^i must not cause packet re-

[2]as an optimization, **TLEM** does not store packets that must be dropped because of queue overflow or random drops.

ordering. This is motivated by practical considerations: reordering would require a sorting step when producing the output schedule, slowing down the emulator and requiring more complex code for managing storage as described in the next Section. The constraint is enforced by conditionally increasing packet release times so that they are monotonically increasing.

4.5 Output stage

The output stage has a very simple task: it only needs to look at the shared buffer, and transmit packets when their release time has elapsed. A simplified version of the code of the output stage is below:

```
// q is the shared buffer
q->now = <current time>
while (true) {
  pkt = head_pkt(q);
  if (<netmap buffer full> ||
      q->head == q->tail ||
      pkt->release_time > q->now) {
    <flush output interface>
    <sleep if next packet is far away>
    q->now = <current time>
    continue;
  }
  <copy pkt to netmap buffer>
  q->head = pkt->next;
}
```

The amount of computation on each packet minimal, and the most expensive operation is the data copy. Prefetch instructions (not shown above) help reduce a bit the effect of cache misses.

High speed and high efficiency requires doing I/O with large batches of packets; this however increases latency. In our case, given that a core is entirely dedicated to the output stage, efficiency is not a major concern as long as we can sustain the required output rate. With the above code, we make sure that overdue packets are promptly moved to the netmap buffer, and then sent without waiting for further packets that are not due yet. The system is self adjusting: if, for any reason, the output stage lags behind, it will build a larger batch before flushing it, thus becoming more efficient and hopefully catching up.

4.6 Shared buffer

Communication between the various stages of **TLEM** is implemented through a large, circular buffer shared by all stages of the pipeline. Packets are written contiguously into the buffer by the input stage, each preceded by a fixed size header containing the packet's release time, its length and a small amount of metadata. For performance reasons, packets are padded to multiples of the word size, and we make sure that individual packets are never split in two parts when the buffer wraps around.

The shared buffer is allocated when the emulator is started, and it is large enough to accommodate the data in the emulated link's queue, plus any packets that may be stored in the delay line. Note that at the speeds at which we operate (we are targeting 40 Gbit/s and higher) even a delay of 100 ms requires 500 Mbytes of memory for data, plus space for packet descriptors and padding.

It is common practice for high speed I/O frameworks to try as much as possible to use "zero-copy" solutions, saving the CPU cycles and memory bandwidth involved with the data copies. In our case, we had to abandon this idea because the of the potential waste of memory, and also because zero copy solutions tend to generate sparse memory accesses, resulting in frequent Translation Lookaside Buffer (TLB) misses which would defeat or greatly reduce the advantages of zero-copy.

Our solution of a contiguous buffer containing both descriptors and data packets is extremely cache friendly, and makes good use of the TLB entries due to high locality of accesses.

Each stage of the pipeline has its own set of pointers (indexes) into the shared buffer, to track which packets should be processed by the stage itself, and also communicate with the downstream stage about the availability of new packets. Contiguous stages in the pipeline act as a producer and a consumer. We need to deal with potential contention in accessing the head and tail pointers, and with the synchronization between the two entities when the buffer becomes empty or completely full.

To reduce access to shared variables, each stage of the pipeline keeps a private copy of the buffer indexes updated by the other stages, and refreshes the copy only when it has consumed all the data/space available. Since the buffer indexes only grow in one direction, this permits a correct access to the buffer while minimizing contention.

For the handling extreme situations (buffer full or empty), we decided not to implement a blocking scheme using semaphores or similar resources. Rather, stages spin on the buffer's indexes when they have no work to do and are waiting for updates. Spinning does not mean using a busy-wait loop: we implement an adaptive mechanism so that we start with busy wait, and then move to short sleeps after some amount of time. CPU utilization rapidly goes to a few percent even with modest sleep times (a few microseconds) and the additional jitter introduced in the processing is modest.

4.7 Bidirectional traffic

TLEM as described operates on a single direction of the traffic. Handling bidirectional traffic is as simple as running two instances of the pipeline in Figure 2, one per direction. Similarly, one can run multiple **TLEM** instances in a single host by starting pipelines on different pairs of interfaces.

5 Performance

As for the case of `netmap-ipfw`, we have run a number of tests with **TLEM** in various configurations (delay and queue sizes) and with different input traffic to evaluate its performance. The system used for evaluation uses a fast i7 CPU (i7-5930k at 3.5 GHz), and 2133 MHz memory, with a sufficient number of cores to run each stage of the pipeline, as well as traffic sources and sinks, on a separate core. We ran our measurements on netmap pipes, and using both FreeBSD and Linux hosts.

The main figures we are interested in are throughput, accuracy in the delay emulation, and stability of performance (both in terms of throughput and jitter in the output).

We briefly address the latter two as they are heavily affected by the various power management mechanism (C-states, frequency scaling) made available by the hardware and exploited more or less aggressively by the operating system.

5.1 Providing a sane test environment

Power management mechanisms on modern CPUs are aimed at reducing power consumption when the system is idle, which happens for 90-99.9% of the time on most systems.

Two mechanisms are called C-states and P-states. C-states (named C0, C1, ...) are states of operation which, when a core is HALTed, more and more parts of the system are shut down, saving energy. The higher the C state, the longer it takes to restart execution; these times are low for state C1 (100ns) and rapidly jump in the 50-100 μs range for higher C states, meaning that when a core goes to sleep with deep C-states enabled, it may take up to 100 μs to wake up. Which C-states can be used by the CPU can be set in the BIOS, or with run-time mechanisms such as setting sysctl variables (on FreeBSD) or keeping certain file descriptors open (on Linux). Once a certain C state is available, the CPU will automatically make use of it when a HALT or equivalent instruction is executed.

P-state, also known as dynamic frequency scaling, are a different mechanism which can be used to slow down active cores, throttling frequency (and reducing operating voltages), once again to reduce power consumption. Throttling is normally decided by software subsystems (called "governors") which monitor system load and manage the operating frequency accordingly.

First and foremost, stable performance demands that C-states are disabled with the exception of C1. A wake up latency of 10..100 μs would have horrible effect on the jitter of the system. Furthermore, 100 μs correspond to about 1300 minimum size packets on a 10 Gbit/s link, meaning that the system will run dangerously close to the total queue size of the input NIC, easily resulting in packet drops.

Dynamic frequency scaling is also a source of jitter. Many power governors dynamically adjust the CPU frequency based on observed load on relatively long windows. If an application is power aware and goes to sleep under light load, the governor may reduce the clock speed to 1/4 or less of the peak value, resulting in limited ability to handle spikes of load.

Finally we would like to mention another source of jitter, namely interrupt moderation. This is a mechanism designed to reduce the interrupt load on the system, which is necessary with conventional I/O architectures (this includes NAPI in Linux) where the interrupt handler performs a significant amount of work.

In netmap, the interrupt handler has relatively little work to do, so while it makes sense to use a bit of interrupt moderation, values should be limited to 10-20 μs to limit jitter.

5.2 Latency accuracy and jitter

We have run a limited number of tests on the system to determine the accuracy and jitter of latency emulation, not only across netmap pipes but also using intel 10 G NICs. Latency errors may come from multiple causes including from C- and P- states (which we disabled in tests, leaving only C1 and pinning CPUs to the maximum clock rate), thread migrations (which we disabled by pinning threads to a specific core, as migration may affect thread wakeup times), and the delays in waking up threads upon interrupts or timer notifications. The latter are, in normal situations, limited to 10 μs or less.

Another factor that affects latency is the load in the various stages of the system. Because we want to exploit batching, each stage may defer notifications or transmissions until a batch of available packets has been fully processed. **TLEM** has a command line parameter to set the maximum batch size, thus choosing a tradeoff between throughput and performance. We found that batch sizes of 64 packets give good results with short packets, although our tests with larger packets suggest that the batch size should also account for packet sizes.

With all the above considerations, we have measured that, on physical interfaces, latency emulation is accurate to 50 μs, which is in line with the expected effect of all the above mechanisms.

5.3 Throughput

We tested **TLEM** in a configuration similar to that presented in Section 2.4.1, using netmap pipes for I/O. Our configuration uses two cores per direction, one for the input stage and one for the output stage. The input stage has three main tasks to perform:

1. read packets;
2. compute timestamps;
3. copy to the shared buffer.

The output stage instead has two tasks:

1. copy from the shared buffer to the netmap buffer;
2. write to the netmap port.

The **TLEM** the pipeline has two stages, both extremely fast as we will see, and we need to run both in order to perform an experiment. Measurements can only indicate the maximum of the time spent in each of the stages of the pipeline, leaving some uncertainty on the location of the actual bottlenecks in the system. Nevertheless, the data reported below will give some hints on what are the operations that influence the performance of the system and suggest ways to improve it.

Our first experiment was done by disabling the copy of packets on both input and output. Note that the shared queue still needs to be updated with packet descriptors, which are scattered through the buffer itself. In this configuration, we achieved a time of 30 ns per packet with zero delay. The time grows to 35 ns when generating a 20 ms delay. The extra delay does not require additional computation, however it changes the memory access patterns, as it defines the distance between a packet is written to the queue and its extraction time. At 10 Gbit/s, 20 ms correspond to 200 Mbits or 25 Mbytes, exceeding the L3 cache size on our system. This means that accesses to the shared buffer must go to main memory, and this explains the extra time in this experiment.

Adding back memory copies (i.e. restoring full functionality of the emulator) brings the per-packet time to about 44 ns with zero delay, and 54 ns with 20 ms delay. The extra 15 ns can be charged to the memory copy, and specifically to the read latency in accessing the source.

We expect that memory copy costs are the dominant component in the operation of the emulator, so we ran some experiments with larger packets (1500 bytes). In this case the per-packet time jumps to very high values, around 250 ns with zero delay, and up to 400 ns for 20 ms delay. These rates correspond to 48 Gbit/s and 30 Gbit/s, respectively.

Once again memory access times are the main culprit for these values. With short delays (and correspondingly short buffers), the memory copies make a reasonable use of L3 cache, which is shared among the various cores used in the experiment. As the delay (and buffer size) increase, the cache is overflown and memory accesses become more expensive, with increased latency and reduced bandwidth.

6 Extensions and future work

We have presented **TLEM**, a fast, pipelined link emulator that can deal with speeds tens of gigabits per second in a scalable way. Being developed in userspace, **TLEM** is easy to extend with additional features such as more complex emulation functions and traffic selection mechanisms, and can be extended with additional pipeline stages to preserve the operating speed in the face of more expensive computations.

The main botteneck in **TLEM** operation is currently constituted by the cost of memory copies. We are studying mechanisms to reduce these costs, including better support for zero copy operation with large packets (the case where copy costs are more important), and options to run copies in parallel on separate cores (at the price of additional latency).

While our system has been designed specifically for link emulation, its components can be easily reused for other applications that fit the pipelined model of operation. One example that we have already implemented is that of a programmable traffic generator: we replace the input stage with one that generates a suitable schedule of packets to transmit, and use the output stage just as a fast data pump that goes through the schedule as desired. The traffic generator can read from a PCAP file, or generate packets programmatically. Other examples include fast packet processors such as firewalls, NAT or similar middleboxes.

Acknowledgements

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] CARBONE, M., AND RIZZO, L. Dummynet revisited. *ACM SIGCOMM Computer Communication Review 40*, 2 (2010), 12–20.

[2] CARBONE, M., AND RIZZO, L. An emulation tool for planetlab. *Computer communications 34*, 16 (2011), 1980–1990.

[3] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev. 33*, 3 (2003), 3–12.

[4] HEMMINGER, S., ET AL. Network emulation with netem. In *Linux conf au* (2005), Citeseer, pp. 18–23.

[5] HUBERT, B. Linux Advanced Routing and Traffic Control. In *Ottawa Linux Symposium 2002*.

[6] INTEL. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378* (2012).

[7] RIZZO, L. netmap-ipfw, a userspace version of ipfw+dummynet running on top of netmap. https://github.com/luigirizzo/netmap-ipfw.

[8] RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review 27*, 1 (1997), 31–41.

[9] RIZZO, L. Dummynet and forward error correction. In *USENIX Annual Technical Conference* (1998).

[10] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC'12* (2012), Boston, MA, USENIX Association.

[11] RIZZO, L. Revisiting network I/O apis: the netmap framework. *Commun. ACM 55*, 3 (Mar. 2012), 45–51.

[12] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 255–270.