Very high speed link emulation with TLEM

Luigi Rizzo, Giuseppe Lettieri, Vincenzo Maffione

Dipartimento di Ingegneria dell'Informazione

Università di Pisa, Italy

Email: {rizzo,g.lettieri}@iet.unipi.it, v.maffione@gmail.com

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 40 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 is distributed as BSD-licensed opensource as part of the netmap distributions, and runs on any system that supports netmap (this includes FreeBSD, Linux and now even Windows).

I. Introduction

Link and network emulators are hardware or software systems that manipulate network traffic in ways similar to the devices they emulate: traffic is subject to queuing, bandwidth limitations, delay, and possibly classification and scheduling. Figure 1 shows some of the configurable features.

Link emulators have been embedded in commodity Operating Systems (OSes) 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] and `tc` [5], which are Linux-only. Placing the emulator within the OS eases experiments, as they can be run with real traffic sources/sinks, and potentially even without an actual network. On the negative side, performance can be limited by the OS' network stack, which is often unable to deal with the extreme packet rates of 10+ Gbit/s networks.

Having recently developed solutions for high speed network I/O, we have been confronted with building high speed link emulators. In this paper we discuss why the existing, aforementioned emulators are not up to the task, and present an alternative design that we developed to build a very high speed link emulator called **TLEM**.
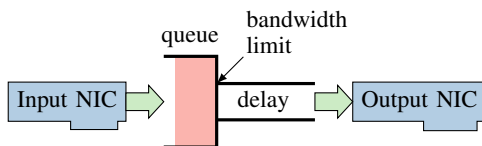

Fig. 1. The basic features supported by a link emulator.

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 keeps 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 40 Gbit/s with 1500-byte frames. These figures exceed the capacity of 10 Gbit/s links, even with minimum packet sizes, and are almost 20 times faster than a basic `dummynet` or `tc` instance. While not including all features of `dummynet`, **TLEM** is very easy to extend, so more complex or custom features such as time-based transmissions, empirical or trace-driven delay emulation, packet mangling, classification, scheduling etc. can be added with little effort.

II. Background

A. Legacy link emulators

After early experiments with custom solutions and dedicated hardware, the quest for link emulators has been effectively addressed in 1997 with our seminal work on `dummynet` [8], which showed how in-system link emulation can be achieved in simple and effective ways. 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 [13] 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 diffusion of the OS.

An emulator embedded in the OS has huge advantages in terms of ease to use, flexibility, and 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. In `dummynet`, we have tried to compensate these limitations with periodic overhauls of the code, adding functionality such as support for loadable schedulers [1], enhanced link emulation [2], and improving performance. Part of our effort was also dedicated

to make `dummynet` available on other platforms, such as Linux and Windows (the OS/X version came by itself when Apple decided to base its OS upon FreeBSD).

Nevertheless, extending an in-kernel emulator remains hard, and very little third party code has made its way into `dummynet`. Similar considerations apply to `netmem` and `tc`.

B. Performance

On the performance side, `dummynet` and `netem` both suffer from the constraints of the environment in which they run. At any time over the past 20 years, the network stack in all OSes has generally been unable to cope with the packet rates produced by the fastest NICs of the time. When 1 Gbit/s NICs became widespread, the bottlenecks were the PCI bus and single core, low speed CPUs. As CPU and bus speeds improved, and additional cores become available, NIC speeds also bumped up to 10+ Gbit/s, leaving the gap unchanged. On 2010 hardware, `dummynet` [1, Sec.4.2] could handle about 0.5 Mpps. Recent versions of `tc` peak at about 1.1 Mpps.

Current hardware is however more balanced between I/O and computation speeds, so the time is ripe for building link emulators that can cope with NIC speeds.

C. Network I/O performance

As discussed elsewhere [10], network I/O subsystems for commodity OSes were designed almost 30 years ago, with constraints (CPU number and speed, memory size and speed, NIC and protocol features) very different from today. Retrofitting the software on modern architectures (which can exploit parallelism, and 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) provided much smaller gains than what one could expect.

Acknowledging the I/O performance issue, much work in recent years has targeted efficient mechanisms for network I/O, particularly for applications such as software switching, traffic capture and generation. High performance APIs such as netmap [11] and DPDK [6] have given excellent results and pushed several improvements (batching, streamlined processing, etc.) also to the standard network stack.

Our netmap architecture is particularly convenient to use, as applications can transparently connect to different "network ports" talking to physical interfaces (NICs), Virtual switches such as VALE [12] or point to point pipes. The amortised I/O costs vary from 10 to 50 ns per packet for short packets, up to a worst case of 150-200 ns for sending a 1500 byte packets through a virtual switch.

Link emulators are good candidates to use the fast I/O frameworks mentioned above. In fact, replacing the network I/O component with a faster one 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 few Sections.

D. 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 ported to user space the entire kernel code for `dummynet` and its associated packet classifier, `ipfw`. This was done by building a library to replicate some kernel functionality in user space (memory allocation, module management, timers, sysctl and ioctl/sockopt), while otherwise keeping the existing (kernel) code almost unmodified. The network I/O path was replaced by calls to the netmap framework.

E. 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. Network I/O is much faster in the userspace version, while the other two are essentially unchanged between the two environments (kernel and userspace), although there might be some differences due to the different locking requirements. To measure performance we ran `netmap-ipfw` connected to netmap pipes on a single-socket system with i7 CPUs at 3.5 GHz, 12 Mbytes of cache and RAM running at 1333 MHz.

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 (≈ 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.

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.

Just doing steps #1 and #4, which account for I/O costs, requires only 26 ns per packet without touching the payload of the packet (pipes are fully zero copy, and `netmap-ipfw` exploits the feature if possible). The value grows to 50 ns if we access even just one byte of the packet, as we pay the cache miss penalty for accessing untouched data.

Enabling step #2 (traffic filtering), increases the processing time by an amount that depends on the complexity of the ruleset. The first rule requires 33 ns, 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 cost component is however #3, 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, and this feature requires two data copies in and out of a temporary buffer. 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 rate, because the same core that runs the entire `netmap-ipfw` process would also have to deal with traffic in the opposite direction.

Discussion. 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. While we have beaten the I/O bottleneck, the design of `ipfw` and `dummynet`, as well as that of other in-kernel emulators, still has significant overheads in packet representation and management, that result in too frequent memory allocations and copies. Shared data structures also force the use of a single thread or coarse grained locking.

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.

III. 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 redesigned the link emulator to further trim unnecessary functions (such as the filtering options that exist in `ipfw`, or the general purpose traffic scheduler in `dummynet`), and use an architecture that can make use of the many cores available in the system, but without introducing expensive locking or checks for data dependencies (such as the ones that typically occur when partitioning traffic and operating in parallel on the subsets).

A. Pipelined structure

The solution we adopted for our design, called **TLEM**, is a pipelined structure, shown in Figure 2. Each stage performs a simple task and has minimal interactions with the others. 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*, that reads incoming traffic and copies it into a *shared buffer*, and an *output stage*, that polls the buffer and releases packets at their due time. Before going to the output stage, packets are annotated with their fate ("drop" or "keep"), the time at which they can be released, and possibly altered
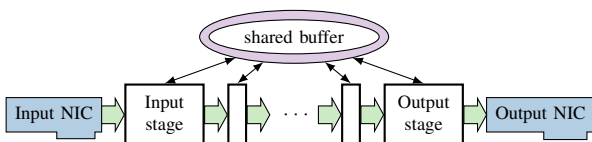

Fig. 2. The architecture of **TLEM**

if the emulation requires packet modifications. Depending on their cost, computations such as annotations and packet modifications can be run in the input stage, or in additional stages in the pipeline.

Decomposing operations in a pipeline 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, and for efficiency, we should make sure that the workload on each stage is well balanced. TLEM can introduce more stages in the pipeline to perform the computations. In the current implementation, we have found that the input and output stages suffice to exceed the speed of a 10 Gbit/s interface.

IV. 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.

A. Packet 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. 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 is the one used in `dummynet`: there, we 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 performance bottleneck that should be avoided.

TLEM, instead, uses only a single enqueue and dequeue operation per packet. Packets coming from the input NIC are immediately[1] stored in a large circular buffer, described in Section IV-F. 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 IV-B and IV-D, and is used by the output stage of the pipeline.

---

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

B. Queue and link bandwidth

Traffic shaping (emulating a link with predefined bandwidth) is a core function of any link emulator. 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. **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.

C. Random packet drops

Congestion-induced drops, as described earlier, are a normal artifact of a communication network and one that **TLEM** emulates precisely. It is sometimes useful to study the behaviour of an application in presence of other types of packet drops or errors. These could be caused by channel noise, or complex congestion situations elsewhere in 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. This feature is implemented by calling a user-supplied C function on each packet, which returns a yes/no answer to determine 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.

D. Link delay

The time t_L^i computed above 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. The link delay may also be 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 drops, users can call a library function or provide arbitrary C code to compute the t_D^i for each packet. Library functions include uniformly or exponentially distributed delays within a range.

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 derived from actual samples, the inverse is simply tabulated with a sufficient number of samples.

TLEM imposes one restriction on link delay distributions: the t_D^i must not cause packet reordering. This is motivated by practical considerations: reordering would require a sorting step when producing the output schedule, slowing down the emulator and complicating storage management. The constraint is enforced by conditionally increasing packet release times so that they are monotonically increasing.

E. Output stage

The output stage has a very simple task: it only needs to look at the shared buffer, and copy packets to the output buffers (provided by netmap) and transmit them when their release time has elapsed. The computation on each packet is minimal, and the most expensive operation is the data copy. Prefetch instructions help reduce a bit the effect of cache misses. The output stage can transmit multiple packets at once when it lags behind, thus highly increasing I/O speed and self-adjusting its efficiency to the demands of the input load.

F. Shared buffer

The stages of **TLEM** communicate through a 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 a cache line, and are never split in two parts when the buffer wraps around.

The buffer is allocated when the emulator is started, and is large enough to store all packets in the queue and the delay line of the emulated link. At the speeds of interest (40 Gbit/s and higher), 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 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 choice 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 pointers into the shared buffer, to track which packet it should process next, and tell the downstream stage about the availability of new packets. Contiguous stages in the pipeline act as a producer and a consumer, and we handle communication between the two with mechanisms similar to those used for lock-free queues.

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

For the handling of extreme situations (buffer full or empty), we decided not to implement a blocking scheme using semaphores or similar mechanisms. Instead, stages spin on the buffer pointer when they have no work to do and are waiting for updates. Spinning does not mean busy-wait: since stages know the release time of the packets, they can switch between sleep and busy wait depending on when they should act next. CPU utilization rapidly goes to a few percent even with modest sleep times (a few microseconds) and the additional jitter introduced by going to sleep is modest.

G. 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.

V. PERFORMANCE

As for `netmap-ipfw`, we studied **TLEM**'s behaviour with various configurations (delay and queue sizes) and input traffic patterns. The test platform uses a fast i7 CPU (i7-5930k at 3.5 GHz), and 2133 MHz memory, with enough cores to run on a separate core each stage of the pipeline, as well as traffic sources and sinks. We ran our measurements on 10-Gbit/s NICs and on netmap pipes (with I/O costs similar to those of 10- and 40-Gbit/s NICs), and using both FreeBSD and Linux.

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 discuss how the latter two are affected by the power management mechanism (C-states, frequency scaling) available on modern hardware and exploited more or less aggressively by the operating system.

A. Providing a sane test environment

Modern CPUs have power management mechanisms, such as C-states and P-states, aimed at reducing power consumption when the system is idle.

With increasing C-states (named C0, C1, ...), more and more parts of the CPU are shut down when a core is HALTed, but this causes longer delays to restart operation. When a core

goes to sleep with deep C-states enabled, it may take up to 100 μs to wake up, compared to less than 1 μs in C1. 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-states, also known as dynamic frequency scaling, are a different mechanism used to slow down active cores, throttling frequency (and reducing operating voltages) to reduce power consumption. Throttling is normally controlled by software subsystems (called "governors") that monitor system load and set the operating frequency accordingly.

Stable performance demands that C-states other than C1 are disabled, as a wake up latency of 10..100 μs would induce a huge jitter on the delay. Also, 100 μs correspond to about 1300 minimum size packets on a 10 Gbit/s link, meaning that the system would 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 intervals. If an application is power aware and goes to sleep under light load, the governor may reduce the clock speed to 1/3 or less of the peak value, resulting in limited ability to handle spikes of load.

Another source of jitter is 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 very little work to do, so while it makes sense to use interrupt moderation, values should be limited to 10-20 μs to keep jitter small.

B. Latency accuracy and jitter

We have run a small 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 Gbit/s NICs. We avoided the effect of C- and P-states by using only C1 and setting CPUs to the maximum clock rate, and prevented thread migrations by pinning threads to a specific core, to reduce delay and jitter in waking up threads upon interrupts or timer notifications. These delays 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.

C. Throughput

We tested **TLEM** in a configuration similar to that presented in Section II-E, 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: i) read packets; ii) compute timestamps; iii) copy to the shared buffer. The output stage instead has two tasks: i) copy from the shared buffer to the netmap buffer; ii) write to the netmap port. The **TLEM** 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, growing to 35 ns when generating a 20 ms delay. The extra delay does not require additional computation, but 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.

VI. 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 bottleneck 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. **TLEM** and the traffic generator described above are part of netmap distributions at `github.com/luigirizzo/netmap/` .

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] RIZZO, L., AND LETTIERI, G. VALE, a switched ethernet for virtual machines. In *CoNEXT '12* (New York, NY, USA, 2012), ACM, pp. 61–72.

[13] 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.