VALE, a Switched Ethernet for Virtual Machines

Luigi Rizzo
Dip. di Ingegneria dell'Informazione
Università di Pisa, Italy
rizzo@iet.unipi.it

Giuseppe Lettieri
Dip. di Ingegneria dell'Informazione
Università di Pisa, Italy
g.lettieri@iet.unipi.it

ABSTRACT

The growing popularity of virtual machines is pushing the demand for high performance communication between them. Past solutions have seen the use of hardware assistance, in the form of "PCI passthrough" (dedicating parts of physical NICs to each virtual machine) and even bouncing traffic through physical switches to handle data forwarding and replication.

In this paper we show that, with a proper design, very high speed communication between virtual machines can be achieved completely in software. Our architecture, called VALE, implements a Virtual Local Ethernet that can be used by virtual machines, such as QEMU, KVM and others, as well as by regular processes. VALE achieves a throughput of over 17 million packets per second (Mpps) between host processes, and over 2 Mpps between QEMU instances, without any hardware assistance.

VALE is available for both FreeBSD and Linux hosts, and is implemented as a kernel module that extends our recently proposed netmap framework, and uses similar techniques to achieve high packet rates.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Experimentation, Performance

Keywords

Virtual Machines, Software switches, netmap

1. INTRODUCTION

A large amount of computing services nowadays are migrating to virtualized environments, which offer significant advantages in terms of resource sharing and cost reduction. Virtual machines need to communicate and access peripherals, which for systems used as servers mostly means disks and network interfaces. The latter are extremely challenging to deal with even in non-virtualized environments, due to the high data and packet rates involved, and the fact that, unlike disks, traffic generation is initiated by external entities on which the receiver has no control. It is then not a surprise that virtual machines may have a tough time in operating network interfaces at wire speed in all possible conditions.

As it is often the case, hardware assistance comes handy to improve performance. As shown in Section 2.2, some proposals rely on multiqueue network cards exporting resources to virtual machines through PCI passthrough, and/or on external switches to copy data between interfaces. However this solution is expensive and not necessarily scalable. On the other hand, software-only solutions proposed to date tend to have relatively low performance, especially for small packet sizes.

We then wondered if there was an inherent performance problem in doing network switching in software. The result we found is that high speed forwarding between virtual machines is achievable even without hardware support, and at a rate that exceeds that of physical 10 Gbit/s interfaces, even with minimum-size packets.

Our contribution: The main result we present in this paper is a system called VALE, which implements a Virtual Local Ethernet that can be used to interconnect virtual machines, or as a generic high speed bus for communicating processes. VALE is accessed through the netmap API, an extremely efficient communication mechanism that we recently introduced [17]. The same API can be trivially used to connect VALE to hardware devices, thus also providing communications with external systems at line rate [16].

VALE is 10..20 times faster than other software solutions based on general purpose OSes (such as in-kernel bridging using TAP devices or various kinds of sockets). It also outperforms NIC-assisted bridging, being capable to deliver well over 17 Mpps with short frames, and over 6 Mpps with 1514-byte frames (corresponding to more than 70 Gbit/s).

The use of the netmap API is an enabling factor for such performance, but the packet rates reported in this paper could have not been achieved without engineering the forwarding code in a way that exploits batched processing. Section 3.3 discussed the solutions we use.

To prove that VALE's performance can be exploited by virtual machines, we then added VALE support to

CoNEXT'12, December 10–13, 2012, Nice, France.

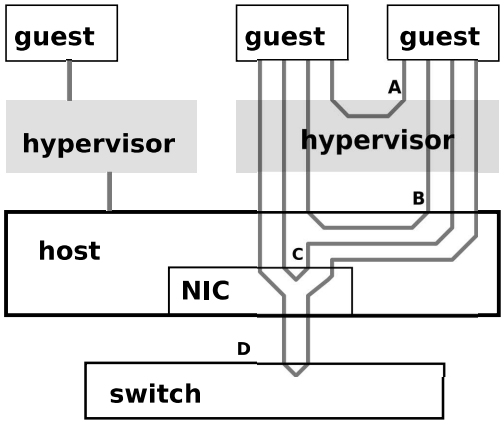

Figure 1: Possible paths in the communication between guests: A) through the hypervisor; B) through the host, e.g. via native bridging; C) through virtual queues the NIC; D) through an external network switch.

QEMU [4] and KVM [9], and measured speedups between 2 and 6 times for applications running on the guest OS[1], reaching over 2.5 Mpps with short frames, and about 2 Mpps with 1514-byte frames, corresponding to 25 Gbit/s. We are confident that we will be able to reach this level of performance also with other hypervisors and guest NIC drivers.

The paper is structured as follows. In Section 2 we define the problem we are addressing in this paper, and present some related work that is also relevant to describe the solutions we adopted. Section 3 details the architecture of our Virtual Local Ethernet, discussing and motivating our design choices. Section 4 comments on some implementation details, including the hypervisor modifications needed to make use of the VALE infrastructure, and device emulation issues. We then move to a detailed performance measurement of our system, comparing it with alternatives proposed in the literature or implemented in other products. We first look at the raw switch performance in Section 5, and then study the interaction with hypervisors and guest in Section 6. Finally, Section 7 discusses how our work can be used by virtualization solutions to achieve large speedups in the communication between virtual machines, and indicates some directions for future work.

2. PROBLEM DEFINITION

The problem we address in this paper is how to implement a high speed Virtual Local Ethernet that can be used as a generic communication mechanism between virtual machines, OS processes, and physical network interfaces.

Solutions to this problem may involve several components. Virtual machine instances in fact use the services supplied by a *hypervisor* (also called Virtual Machine Monitor, VMM) to access any system resource (from CPU to memory to communication devices), and consequently to communicate with each other. Depending on the architecture of the system, resource access from the guest can be mediated by the hypervisor, or physical resources may be partially or completely allocated to the guest, which then uses them with no interference from the hypervisor (except when triggering protection mechanisms).

Figure 1 illustrates implementations of the above concepts in the case of network communication between virtual machines. In case A, the hypervisor does a full emulation of the network interfaces (NICs), and intercepts outgoing traffic so that any communication between the virtual machine instances goes through it; in QEMU, this is implemented with the `-net user` option. In case B, the hypervisor still does NIC emulation, but traffic forwarding is implemented by the host, e.g. through an in-kernel bridge (`-net-tap`) or a module such as the one we present in this paper.

Other solutions give the virtual machine direct access to the NIC (or some of its queues). In this case, the NIC itself can implement packet forwarding between different guests (see the path labeled C), or traffic is forwarded to an external switch which in turn can bounce it back to the appropriate destination (path D).

NIC emulation, as used in A and B, gives the hypervisor a lot of control on the operations done by the guest, and works as a nice adaptation layer to run the guest on top of hardware that the guest OS would not know how to control.

Common wisdom suggests that NIC emulation is slow, and direct NIC access (as in C and D) is generally necessary for good virtual network performance, even though it requires some form of hardware support to make sure that the guest does not interfere with other critical system resources.

However, the belief that NIC emulation is inherently slow is wrong, and mostly the result of errors in the emulation code [15] or missing emulation of features (such as interrupt moderation, see [14]) that are fundamental for high rate packet processing. While direct hardware access may help in communication between the guest and external nodes, virtio [20], proprietary virtual device emulators [22,23], and as we show in this paper, even e1000 emulation, done properly, provide excellent performance, comparable or even exceeding that of real hardware in VM-to-VM communication.

2.1 Organization of the work

In summary, the overall network performance in a virtual machine depends on three components:

- the guest/hypervisor communication mechanism;
- the hypervisor/host communication mechanism;
- the host infrastructure that exposes multiple physical or virtual NIC ports to its clients.

In this paper we first present an efficient architecture for the last component, showing how to design and implement an extremely fast Virtual Local Ethernet (VALE) that can be used by the hypervisors (and possibly exported to the guest OS), or even used directly by regular host processes. We then extend some popular hypervisors so that their communication with the host can be made efficient and exploit the speed of VALE. Finally, we discuss mechanisms that we implemented in previous work and that can be extremely effective to improve performance in a virtualized OS.

2.2 Related work

There are three main approaches to virtualized I/O, which rely on different choices in the communication between guest and hypervisor, and between hypervisor and the host/bridging infrastructure. We examine them in turn.

[1]especially when we can overcome the limitations of the emulated device driver, see Section 6

2.2.1 Full virtualization

The simplest approach (in terms of requirements for the guest) is to expose a virtual interface of the type known to the guest operating system. This typically involves intercepting all accesses to critical resources (NIC registers, sometimes memory regions) and use them to update a state machine in the hypervisor that replicates the behaviour of the hardware. Historically, this is the first solution used by most emulators, starting from VMware to QEMU [4] and other recent systems.

With this solution the hypervisor can be a simple process that accesses the network using standard facilities offered by the host: TCP/UDP sockets (usually to build tunnels or emulate ethernet segments), or BPF/libpcap [10] or virtual interfaces (TAP) to inject raw packets into the network.

TAP is a software device with two endpoints: one is managed as a network interface (NIC) by the operating system, the other one is a file descriptor (FD) driven by a user process. Data blocks sent to the FD endpoint appear as ethernet packets on the NIC endpoint. Conversely, ethernet packets that the operating system sends to the NIC endpoint can be read by the user process from the FD endpoint. A hypervisor can then pass the guest's traffic to the FD endpoint of a TAP device, whose NIC endpoint can be connected to other NICs (TAPs or physical devices) using the software bridges available in the Linux and FreeBSD kernels, or other software bridges such as Open vSwitch [12].

There also exist solutions that run entirely in user space, such as VDE [8], providing configurable tunnels between virtual machines. In general, this kind of solution offers the greatest ease of use, at the expense of performance. Another possibility is offered by MACVTAPs [1], which are a special kind of TAP devices that can be put in a "bridge" mode, so that they can send packets to each other. Their main purpose is to simplify networking setup, by removing the need to configure a separate bridge.

2.2.2 Paravirtualization

The second approach goes under the name of paravirtualization [3] and requires modifications in the guest. The guest becomes aware of the presence of the hypervisor and cooperates with it, instead of being intercepted by it. As far as I/O is concerned, the modifications in the guest generally come in the form of new drivers for special, "paravirtual" devices. VMware has always offered the possibility to install the VMware Tools in the guest to improve interoperability with the host and boost I/O performance. Their vSphere virtualization infrastructure also offers high performance vSwitches to interconnect virtual machines [22].

Xen offers paravirtualized I/O in the form a special *driver domain* and pairs of backend-frontend drivers. Frontend drivers run in the guests and exchange data with the backend drivers running in the driver domain, where a standard Linux kernel finally completes the I/O operations. This architecture achieves fault isolation and driver reuse, but performance suffers [11]. Bridging among the guests is performed by a software bridge that connects the driver backends in the guest domain.

XenLoop [24] is a solution that improves throughput and latency among Xen guests running on the same host. It uses FIFO message queues in shared memory to bypass the driver domain. XenLoop is tightly integrated with the hypervisor, and in fact, its exclusive focus seems to provide a fast network channels to hypervisors. For this reason is not completely comparable with the VALE switch presented in this work, which also aims to implement be a generic communication mechanism useful also outside the virtualization world. In terms of performance, XenLoop seems to peak at around 200 Kpps (on 2008 hardware), which is significantly below the performance of VALE.

The KVM [26] hypervisor and the Linux kernel (both as a guest and a host) offer support for virtio [20] paravirtualization. Virtio is an I/O mechanism (including virtio-net for network and virtio-disk for disk support) based on queues of scatter-gather buffers. The guest and the hypervisor export shared buffers to the queues and notify each other when batches of buffers are consumed. Since notifications are expensive, each endpoint can disable them when they are not needed. The main idea is to reduce the number of context switches between the guest and the hypervisor.

Vhost-net [2] is an in-kernel data-path for virtio-net. Vhost-net is used by KVM, but it is not specifically tied to it. In KVM, virtio-net *notify* operations cause a hardware-assisted VM-exit to the KVM kernel module. Without vhost-net, the KVM kernel module then yields control to the KVM process in user space, which then accesses the virtio buffers of the guest, and writes packets to a TAP device using normal system calls. A similar, reversed path is followed for receive operations.

With vhost-net enabled, instead, the KVM kernel module completely bypasses the KVM process and triggers a kernel thread which directly writes the virtio buffers to the TAP device. The bridging solutions for this technique are the same as those for full virtualization using TAP devices, and the latter typically become the performance bottleneck.

2.2.3 Direct I/O access

The third approach is to avoid guest/hypervisor communication altogether and allow the guest to directly access the hardware [25]. In the simplest scenario a NIC is dedicated to a guest which gains exclusive access to it, e.g., by PCI passthrough. This generally requires hardware support to be implemented safely. Moreover, DMA transfers between the guest physical memory and the peripheral benefit from the presence of an IOMMU [7]. More complex scenarios make use of multi-queue NICs to assign a separate queue to each guest [21], or of programmable network devices to implement device virtualization in the device itself [13]. These solutions are generally able to achieve near native performance in the guest, but at the cost of requiring specialized hardware. Bridging can be performed either in the NIC itself, as in [21], or by connecting real external switches.

3. VALE, A VIRTUAL LOCAL ETHERNET

The first goal of this work is to show how we built a software, high performance Virtual Local Ethernet (which we call VALE), that can provide access ports to multiple clients, be them hypervisors or generic host processes, as shown in Figure 2. The target throughput we are looking for is in the millions of packets per second (Mpps) range, comparable or exceeding that of 10 Gbit/s interfaces. Given that the hypervisor might be running the guest as a userspace process, it is fundamental that the virtual ethernet is accessible from user space with low overhead.

As mentioned, network I/O is challenging even for systems running on real hardware, for the reasons described in [17]:

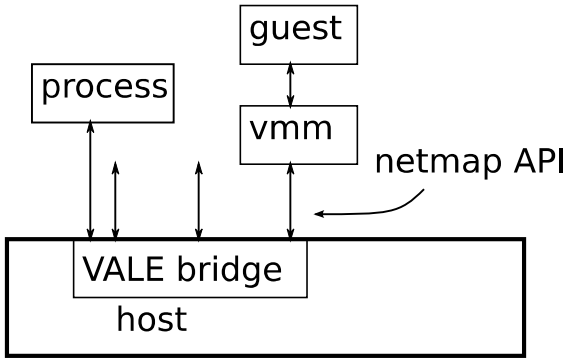

Figure 2: A VALE local ethernet exposes multiple independent ports to hypervisors and processes, using the netmap API as a communication mechanism.

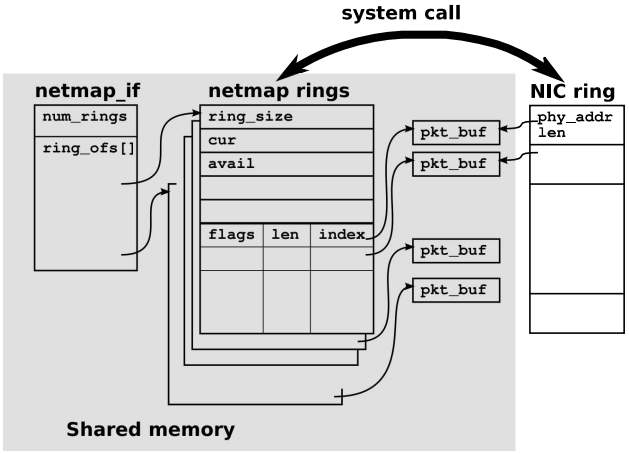

Figure 3: The memory areas shared between the operating system and processes, and manipulated using the netmap API.

expensive system calls and memory allocations are incurred on each packet, while packet rates of millions of packets per second exceed the speed at which system calls can be issued.

In netmap [16] we solved these challenges through a series of simple but very effective design choices, aimed at amortizing or removing certain costly operations from the critical execution paths. Given the similarity to the problem we are addressing in VALE, we use the netmap API as the communication mechanism between the host and the hypervisor. A brief description of the netmap API follows.

3.1 The netmap API

The netmap framework was designed to implement a high performance communication channel between network hardware and applications in need of performing raw packet I/O. The core of the framework is based on a shared memory region (Figure 3), which hosts packet buffers and their descriptors, and is accessible by the kernel and by userspace processes. A process can gain access to a NIC, and tell the OS to operate it in netmap mode, by opening the special file `/dev/netmap`. An `ioctl()` is then used to select a specific device, followed by an `mmap()` to make the netmap data structures accessible.

The content of the memory mapped region is shown in Figure 3. For each NIC, it contains preallocated buffers for transmit and receive packets, and one or more[2] pairs of circular arrays (*netmap rings*) that store metadata for the transmit and receive buffers. Besides the OS, buffers are also accessible to the NIC through its own *NIC rings* – circular arrays of buffer descriptors used by the NIC's hardware to store incoming packets or read outgoing ones.

Using a single `ioctl()` or `poll()` system call, a process can notify the kernel to send multiple packets at once (as many as they fit in the ring). Similarly, receive notifications for an entire batch of packets are reported with a single system call. This way, the cost of system calls (up to 1 μs or more even on modern hardware) is amortized and their impact on individual packets can become negligible.

The other expensive operations – buffer allocations and data copying – are removed because buffers are preallocated and shared between the user process and (ultimately) the NIC itself. The role of the system calls, besides notifications, is to validate and convert metadata between the netmap and the NIC ring, and to perform safety-critical operations such as writing to the NIC's register. The implicit synchronization provided by the system call makes access to the netmap ring safe without the need of additional locking between the kernel and the user process.

Netmap is a kernel module made of two parts. Device-independent code implements the basic functions (`open()`, `close()`, `ioctl()`, `poll()/select()`), while device-specific *netmap-backends* extend device drivers and are in charge of transferring metadata between the netmap ring and the NIC ring (see Figure 3). The backends are very compact and fast, allowing netmap to send or receive packets at line rate even with the smallest packets (14.88 Mpps on a 10 Gbit/s interface) and with a single core running at less than 900 MHz. True zero-copy between interfaces is also supported, achieving line-rate switching with minimum-sized packets at a fraction of the maximum CPU speed.

3.2 A netmap-based Virtual Local Ethernet

From a user's perspective, our virtual switch shown in Figure 2 offers each user an independent, virtual NICs connected to a switch and accessible with the netmap API. NIC names start with the prefix `vale` (to differentiate them from physical NICs); both virtual NICs and switches are created dynamically as users access them.

When the netmap API requests to access a NIC named `valeX:Y` (where X and Y are arbitrary strings), the system creates a new VALE switch instance called X (if not existing), and attaches to it a port named Y.

Within each switch instance, packets are transferred between ports as in a learning ethernet bridge: the source MAC address of each incoming packet is used to learn on which port the source is located, then the packet is forwarded to zero or more outputs depending on the type of destination address (unicast, multicast, broadcast) and whether the destination is known or not. The following pseudocode describes the forwarding algorithm. The set of destinations is represented by a bitmap, so up to 64 output ports can be easily supported. A packet is forwarded to port `j` if `dst & (1<<j)` is non zero.

[2]for NICs with multiple transmit and receive queues

```
void tx_handler(ring, src_if) {
    cur = ring->cur; avail = ring->avail;
    while (avail-- > 0) {
        pkt = ring->slot[cur].ptr;
        // learn and store the source MAC
        s = mac_hash(pkt->src_mac);
        table[s] = {pkt->src_mac, src_if};
        // locate the destination port(s)
        d = mac_hash(pkt->dst_mac);
        if (table[d].mac == pkt->dst_mac)
             dst = table[d].src;
        else
             dst = ALL_PORTS;
        dst &= ~(1<<src_if); // avoid src_if
        // forward as needed
        for (j = 0; j < max_ports; j++) {
            if (dst & (1<<j)) {
                lock_queue(j);
                pkt_forward(pkt, ring->slot[cur].len, j);
                unlock_queue(j);
            }
        }
        cur = NEXT(cur);
    }
    ring->cur = cur; ring->avail = avail;
}
```

This code, operating on one packet at a time, is very similar to the implementation of most software switches found in common OSes. However its performance is poor (relatively speaking; we measured almost 5 Mpps as shown in Figure 8, which is still 5 times faster than existing in-kernel bridging code), for two main reasons: locking and data access latency.

The incoming queue on the destination port, in fact, must be protected against concurrent accesses, and in the above code the cost of locking (or equivalent mechanism) is paid on each packet, possibly exceeding the packet processing costs related to bridging. Secondly, the memory accesses to read the metadata and headers of each packet may have a significant latency if these data are not in cache.

3.3 Performance enhancements

To address these performance issues, we use a different sequence of operations, which permits processing packets in batches and supports data prefetching.

Batching is a well know technique to amortize the cost of some expensive operations over a large set of objects. The downside of batching is that, depending on how it is implemented, it can increase the latency of a system. Given that the netmap API used by VALE supports the transmission of potentially large sets of packets on each system call, we want to provide the system administrator with mechanisms to enforce the desired tradeoffs between performance and latency.

The key idea is to implement forwarding in multiple stages. In a first stage we collect a batch of packets of *bounded maximum batch size* from the input set of packets; a short batch is created if there are fewer packets available than the batch size. In this stage we copy metadata (packet sizes and indexes/buffer pointers), and also issue prefetch instructions for the payload of the packets in the batch, so that the CPU can start moving data towards caches before using them. Figure 4 shows how packets from the netmap ring are copied into a working array (`pool`) which also has room to store the set of destinations for each packet (these fields will be filled in the next stage). The pseudocode for this stage is the following:

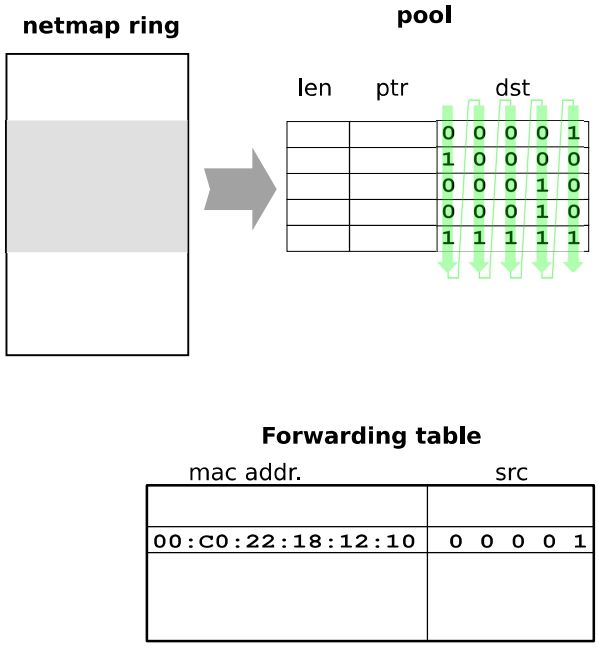

Figure 4: The data structures used in VALE to support prefetching and reduce the locking overhead. Chunks of metadata from the netmap ring are copied to a temporary array, while at the same time prefetching packets' payloads. A second pass then runs a destination lookup in the forwarding table, and finally the temporary array is scanned in column order to serve each interface in a single critical section.

```
void tx_handler(ring, src_if) {
    // stage 1, build batches and prefetch
    i = 0; cur = ring->cur; avail = ring->avail;
    for (; avail-- > 0; cur = NEXT(cur)) {
        slot = &ring->slot[cur];
        prefetch(slot->ptr);
        pool[i++] = {slot->ptr, slot->len, 0};
        if (i == netmap_batch_size) {
            process_batch(pool, i, src_if);
            i = 0;
        }
    }
    if (i > 0)
        process_batch(pool, i, src_if);
}
```

3.3.1 *Processing a batch*

The next processing stage involves updating the forwarding table, and computing the destination port(s) for each packet. The fact that the packet's payload has been brought into caches by the previous prefetch instructions should reduce the latency in accessing data. Furthermore, repeated source/destination addresses within a batch should improve locality in accessing the forwarding table.

Once destinations have been computed, the final stage can forward traffic iterating on output ports, which satisfies our requirement of paying the cost of locking/unlocking a port only once per batch. The following pseudocode shows a simplified version of the forwarding logic.

```
void process_batch(pool, n, src_if) {
    // stage 2, compute destinations
    for (i = 0; i < n; i++) {
        pkt = pool[i].ptr;
        s = mac_hash(pkt->src_mac);
        table[s] = {pkt->src_mac, src_if};
        d = mac_hash(pkt->dst_mac);
        if (table[d].mac == pkt->dst_mac)
            pool[i].dst = table[d].src;
        else
            pool[i].dst = ALL_PORTS;
    }
    // stage 3, forward, looping on ports
    for (j = 0; j < max_ports; j++) {
        if (j == src_if)
            continue; // skip src_if
        lock_queue(j);
        for (i = 0; i < n; i++) {
            if (pool[i].dst & (1<<j))
                pkt_forward(pool[i].pkt, pool[i].len, j);
        }
        unlock_queue(j);
    }
}
```

In the actual implementation, the final stage is further optimized to reduce the cost of the inner loop. As an example, the code skips ports for which there is no traffic, and delays the acquisition of the lock until the first packet for a destination is found, to shorten of critical sections[3].

3.3.2 *Avoiding multicast*

Forwarding to multiple destinations is a significant complication in the system, both in terms of space and runtime. The worst case complexity of stage 3, even for unicast traffic, is still $O(N)$ per packet, where N is the number of ports in the bridge. If a bridge is expected to have a very large number of ports (e.g. connected to virtual machines), it may make sense to remove support for multicast/broadcast forwarding[4].

To achieve this, the field used to store bitmap addresses can be recycled to build lists of packets for a given destination, thus avoiding an expensive worst case behaviour. Traffic originally directed to multiple ports (unknown destinations, or ARP requests/advertisements, BOOTP/DHCP requests, etc.) can be directed to a default port where a user-level process will respond appropriately, e.g. serving ARP and DHCP requests (a similar strategy is normally used in access nodes – DSLAM/BRAS – that terminate DSL lines).

3.3.3 *Alternative forwarding functions*

We should note that stage 2 is the only place where the forwarding function is invoked. It is then relatively simple, and it will be the subject of future work, to replace it with alternative algorithms such as an implementation of a OpenFlow dataplane.

3.4 Copying data

The final processing stage calls function `pkt_forward()`, which is in charge of queueing the packet on one of the destination ports. The simplest way to do this, and the one we use in VALE, is to copy the payload from the source to the destination buffer. Copying is especially convenient because it supports the case where the source and destination ports have buffers mapped in mutually-inaccessible address spaces. This is exactly one of the key requirements in VALE, where clients attached to the ports are typically virtual machines, or other untrusted entities, that should not interact in other ways than through the packets sent to them.

Ideally, if a packet has only a single destination, we could simply swap buffers between the transmit and the receive queue. However this method requires that buffers be accessible in both the sender and the receiver's address spaces and is not compatible with the need to isolate virtual machines.

While data copies may seem expensive, on modern systems the memory bandwidth is extremely high so we can achieve good performance as long as we make sure to avoid cache misses. We use an optimized copy function which is extremely fast: if the data is already in cache (which is likely, due to the `prefetch()` and the previous access to the MAC header), it takes less than 15 ns to copy a 60-bytes packet, and 150 ns for a 1514-bytes packet.

4. IMPLEMENTATION DETAILS

VALE is implemented as a kernel module, and is available from [19] for both FreeBSD and Linux as an extension of the netmap module. Thanks to the modular architecture of the netmap software, the additional features to implement VALE required less than 1000 additional lines of code, and the system was running on both FreeBSD and Linux from the very first version. The current prototype supports up to 8 switches per host (the number is configurable at compile time), with up to 64 ports each.

4.1 External communication

As presented, VALE implements only a local ethernet switch, whose use is restricted to processes and hypervisors running on the same host. The connection with external networks is however trivially achieved with one of the tools that are part of our netmap framework, which can bridge two arbitrary netmap interfaces at line rate, using an arrangement similar to that in Figure 5. We can use one netmap-bridge to connect to the host stack, and one or more to connect to physical interfaces.

Because the relevant code is already existing and operational with the desired performance, we will not discuss it in

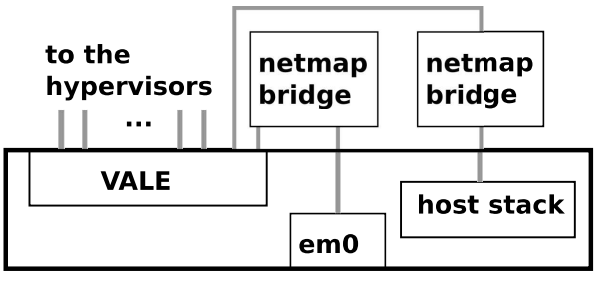

Figure 5: The connection between VALE, physical network interfaces and the host stack can be built with existing components (netmap bridge) which implement zero-copy high speed transfer between netmap-compatible ports.

[3]Computing the first and last packet for each destination is a lot more expensive in the generic case, as it requires iterating on the bitmap containing destinations. Even more expensive, from a storage point of view, is tracking the exact set of packets for a given destination.

[4]This feature will be implemented in future releases.

the experimental section. As part of future work, we plan to implement a (trivial) extension of VALE to directly access the host stack and network interfaces without the help of external processes.

4.2 Hypervisor changes

In order to make use of the VALE network, hypervisors must be extended to access the new network backend. For simplicity, we have made modifications only to two popular hypervisors, QEMU [4] and KVM [9], but our changes apply in a similar way to VirtualBox and other systems using host-based access to network interfaces. We do not foresee much more complexity in making VALE ports accessible to Xen DomU domains.

The QEMU/KVM backend is about 400 lines of code and it implements the open and close routines, and the read and write handlers called on the VALE file descriptor when there is traffic to transfer.

Both QEMU and KVM access the network backend by `poll()`'ing on a file descriptor, and invoking read/write handlers when ready. For VALE, the handlers do not need to issue system calls to read or write the packets: the payload and metadata are already available in shared memory, and information on new packets to send or receive are passed to the kernel the next time the hypervisor calls `poll()`. This enables the hypervisor to exploit the batching, in turn contributing to reduce the I/O overhead.

As we will see in the performance evaluation, VALE is much faster than other software solutions used to implement the network backend. This extra speed may stress the hypervisor in unusual ways, possibly emphasizing some pre-existing performance issue or bugs. We experienced similar problems when modifying Open vSwitch to use netmap [18], and we found similar issues in this work.

Specifically, we encountered two performance-related bugs in the NIC emulation code in QEMU.

First of all, after some asymmetric tx versus rx performance results [15], we discovered that the code involved in the guest-side emulation of most network cards was missing a notification to the backend when the receive queue changed status from full to not-full. This made the input processing timeout-based rather than traffic based, effectively limiting the maximum receive packet rate to 100-200 Kpps. The fix, which we pushed to the QEMU developers, was literally one line of code and gave us a speedup of almost 10 times, letting us reach 1-2 Mpps range depending on the hypervisor.

The second problem involves the emulation of interrupt moderation, a feature that is fundamental to achieve high packet rates even on real hardware. Moderation is even more important in virtual machines where the handling of interrupts (which involves numerous accesses to I/O ports) is exceptionally expensive. We found that the e1000 emulation code implements the registers related to moderation, but then makes no use of them, causing interrupts to be generated immediately on every packet transmission or reception. We developed a partial fix [14] which improved the transmit performance by 7-8 times with this change alone.

We note that problems of this kind are extremely hard to identify (it takes a very fast backend to generate this much traffic, and a fast guest to consume it) and easy to misattribute. As an example, the complexity of device emulation is usually indicated as the main reason for poor I/O performance, calling for alternative solutions such as `virtio` [20] or other proprietary APIs [23].

4.3 Guest issues

A fast network backend and a fast hypervisor do not imply that the guest machines can communicate at high speed. Several papers in the literature show that even on real hardware, packet I/O rates on commodity operating systems are limited to approximately 1 Mpps per core. High speed communication (1..10 Gbit/s) is normally achieved thanks to a number of performance-enhancing techniques such as the use of jumbo buffers and hardware offloading of certain functions (checksum, segmentation, reassembly).

As we recently demonstrated [16], this low speed is not an inherent limitation in the hardware, but rather the result of exceeding complexity in the operating system, and we have shown how to achieve much higher packet rates using netmap as the mechanism to access the network card.

As a consequence, for some of our high speed tests we will use the network device in netmap mode *also within the guest*. The same reasons that make netmap very fast on real hardware, also help when running on emulated hardware: on both the transmit and the receive side, operations that need to be run in interpreted mode, or to trap outside the emulator, are executed once per each large batch of packets, thus contributing to improving performance.

5. PERFORMANCE EVALUATION

We have measured the performance of VALE on a few different multicore systems, running both FreeBSD and Linux as the host operating systems, and QEMU and KVM as hypervisors. In general, these experiments are extremely sensitive to CPU and memory speeds, as well as to data layout and compiler optimizations that may affect the timing of critical inner loops of the code. As a consequence, for some of the (many) tests we have run there are large variations (10-20%) of the experimental results, also due to slightly different versions of the code or to the synchronization of the processes involved. We also noted a steady and measurable decline in QEMU/KVM network throughput (about 15% on the same hardware) between versions 0.9, 1.0 and 1.1.

This said, the difference in performance between VALE and competing solutions is much larger (4..10 times) than the variance on the experimental data, so we can draw correct conclusions even in presence of noisy data.

For the various tests, we have used a combination of the following components:

- **Hardware and host operating systems:**
  i7-2600K (4 core, 3.2 GHz) + FreeBSD-9;
  i7-870 (4 core, 2.93 GHz) + FreeBSD-10;
  i5-750 (4 core, 2.66 GHz) + Linux 3.2.12.
  i7-3930K (6 core/12 threads, 3.2 GHz) + Linux 3.2.0.
  In all cases RAM is DDR3-1.33 GHz and the OS is running in 64-bit mode.

- **Hypervisors:** QEMU 1.0.1 (both FreeBSD and Linux); KVM 1.0.1/1.1.0 (Linux).

- **Network backends:** TAP with/without vhost-net (Linux); VALE (Linux and FreeBSD).

- **Guest network interface/OS:** plain e1000, e1000-netmap (Linux and FreeBSD); virtio (only Linux).

Not all combinations have been tested due to lack of significance, unavailability, or bugs which prevented certain configurations from working. We should also mention that, especially for the data reporting peak performance, the numbers we report here are conservative. During the development of this work we have implemented some optimizations that shave a few nanoseconds from each packet's processing time, resulting in data rates in excess of 20 Mpps at minimum packet sizes.

Following the same approach as in the description of the system, we first benchmark the performance of the virtual local ethernet, be it our VALE system or equivalent ones. This is important because in many cases the clients are much slower, and their presence would lead to an underestimate of the performance of the virtual switch.

5.1 Bridging performance

The first set of tests analyzes the performance of various software bridging solutions. The main performance metric for packet forwarding is the throughput, measured in *packets per second* (pps) and *bits per second* (bps).

pps is normally the most important metric for routers, where the largest cost factors (source and destination lookup, queueing) are incurred on each packet and are relatively independent of packet size. Bridges and switches, however, need *also* (but not *only*) a characterization in bps, because they operate at very high rates and often hit other bottlenecks such as memory or bus bandwidth. We will then run our experiments with both minimum and maximum ethernet-size packets (60 and 1514 bytes).

Note that many *bps* figures reported in the literature are actually measured using jumbograms (8-9 Kbytes). The corresponding *pps* rates are between 1/6 and 1/150 of those shown here.

The traffic received by a bridge should normally go to a single destination, but there are cases (multicast or unknown destinations) where the bridge needs to replicate packets to multiple ports. Hence the number of active ports impacts the throughput of the system.

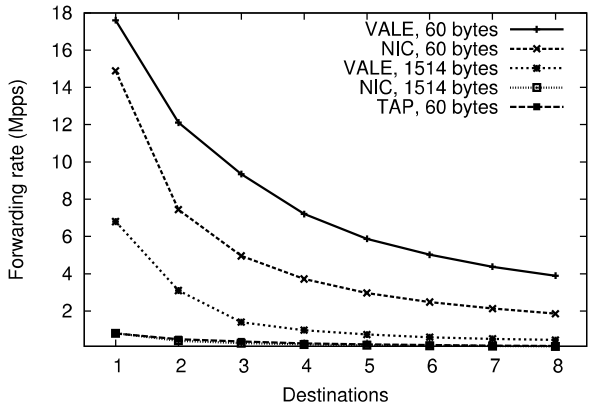

Figure 6: Forwarding rate versus number of destinations. VALE beats even NIC-based bridging, with over 17 Mpps (vs. 14.88) for 60-byte packets, and over 6 Mpps (vs. 0.82) for 1514-byte packets. TAP is at the bottom, peaking at about 0.8 Mpps in the best case.

In Figure 6 we compare the forwarding throughput when using 60 and 1514 byte packets, for three technologies: TAP plus native Linux bridging[5], our VALE bridge, and NIC/switch-supported bridging (in this case we report the theoretical maximum throughput on a 10 Gbit/s link).

Since the traffic directed to a bridge can be forwarded[6] to a single output port (when the destination is known), or to multiple ports (for multicast/broadcast traffic, or for unknown destinations), we expect a decrease in the forwarding rate as the number of active ports grows.

Indeed, all the three solutions (VALE, NIC, TAP) expose a $1/N$ behaviour. For VALE and TAP this is because the forwarding is done by a single core while the work is proportional to the number of ports. For switch-based bridging, the bottleneck is instead determined by the bandwidth available on the link from the switch, which has to carry all the replicas of the packet for the various destinations. A similar phenomenon occurs for NIC-based bridging, this time the bottleneck being the PCI-e bus.

In absolute terms, for traffic delivered to a single destination and 60-byte packets, the best options available for Linux bridging achieve a peak rate of about 0.80 Mpps. Next comes NIC-based forwarding, which is limited by the bandwidth on the PCI-e interconnection between the NIC and the system. Most 10 Gbit/cards on the market use 4-lane PCI-e slots per port, featuring a raw speed of 16 Gbit/s per port per direction. Considering the overhead for the transfer of descriptors, each port has barely enough bandwidth to sustain line rate. In fact, as we measured in [16], packet sizes that are not multiple of a cache line size cannot even achieve line rate due to extra traffic generated to read and write entire cache lines.

The curves for VALE are still above, peaking at 17.6 Mpps for a single destination and 60-byte packets, again decreasing as the number of receivers grows. Here the bottleneck is given by the combination of CPU cycles (needed to do the packet copies) and memory bandwidth.

The numbers for 1514-byte packets are even more impressive. Linux bridging is relatively stable at a low value (packet size is not a major cost item). NIC based forwarding is still limited to line rate, approximately 820 Kpps, whereas VALE reaches over 6 Mpps, or 72 Gbit/s.

5.1.1 Per packet time

Using the same data as in Figure 6, Figure 7 shows the per-packet processing time used by VALE depending on the number of destinations. This representation gives a better idea of the time budget involved with the various operations (hashing, address lookups, data copies, loop and locking overheads). Among other things, these figures can be used to determine a suitable batch size so that the total processing time matches the constraint of latency-sensitive applications.

The figure shows that for one port and small packets the processing time is 50-60 ns per packet when using large batches (128 and above). With separate measurements we estimated that about 15 ns are spent in computing the Jenk-

[5]we also tested the in-kernel Open vSwitch module, but it was always slightly slower than native bridging.

[6]In principle we should also consider the case of traffic being dropped by the bridge, but this is always much faster than the other cases (as an example, VALE can drop packets at almost 100 Mpps), so we do not need to worry about it.

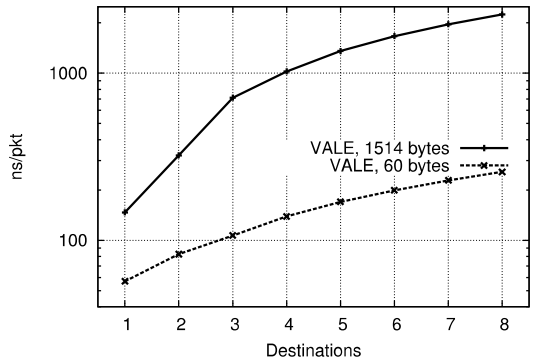

Figure 7: Per-packet time versus number of destinations for VALE. See text in Section 5.1.1 for details.

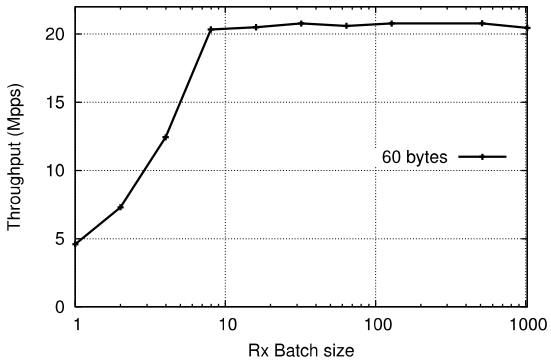

Figure 9: Throughput versus receive batch size. Kernel and sender use batch=1024.

ins hash function, taken from the FreeBSD bridging code, and approximately 15 ns are also necessary to perform a copy of a 60-byte packet (with data in cache). The cost of copying a 1514 byte packet is estimated at about 150 ns.

As mentioned in Section 3.4, VALE uses data copies in all cases. In the case of multicast/broadcast traffic, the only alternative to copying data would be to implement shared readonly buffers and a reference-counted mechanism to track when the buffer can be freed. Apart from the complications of the mechanism, the cost in accessing the reference count from different CPU cores is likely comparable or greater than the data copy costs, even for 1514-byte packets.

5.2 Effectiveness of batching

The huge difference in throughput between VALE and other solutions depends on both the use of the netmap API, which amortizes the system call costs, and also on the processing of packets in batches in the forwarding loop. Figure 6 has been computed for a batch size of 1024 packets, but as we will show, even smaller batch sizes are very effective.

There are in fact three places where batching takes place: on the TX and RX sides, where its main goal is to amortize the cost of the system call to exchange packets with the kernel; and within the kernel, as discussed in Section 3.3, where the goal is to reduce locking contention.

The following experiments show the effect of various combinations of these three batching parameters. Experiments have been run on a fast i7-3930K using Linux 3.2, and unicast traffic between one sender and one receiver; the CPU has multiple cores so we expect sender and receiver to run in parallel on different cores.

Figure 8 shows the throughput versus kernel batch size, when both sender and receiver use a large value (1024) to amortize the system call cost as much as possible. Starting at about 5 Mpps with a batch of 1, we achieve dramatic increase even with relatively small kernel batches (e.g. 8 packets), and there is a diminishing return as we move above 128 packets. Here we have the following dynamics: the kernel, with its small batch size, acts as the bottleneck in the chain, waking up the receiver frequently with small amounts of packets. Because it uses a large batch size, the receiver cannot become the bottleneck: if during one iteration it is too slow in processing packets, in the next round it will be able to catch up draining a larger set of packets.

Figure 9 shows the behaviour of the system with different receiver batch sizes, and kernel and sender using a batch of 1024. In this experiment the receiver is the bottleneck, and the main cost component here is the system call, which however always finds data available so it never needs to sleep (a particularly costly operation). As a consequence, a small batching factor suffices to reach the peak performance.

Finally, Figure 10 shows the effect of different transmit

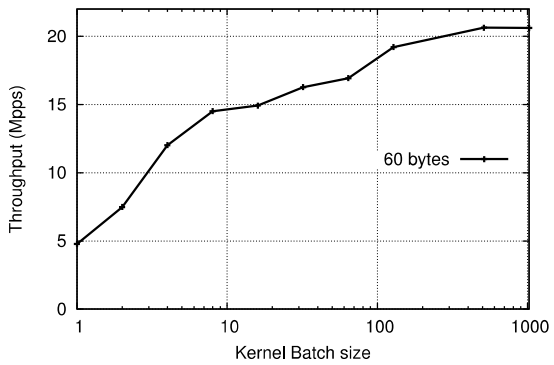

Figure 8: Throughput versus kernel batch size. Sender and receiver use batch=1024.

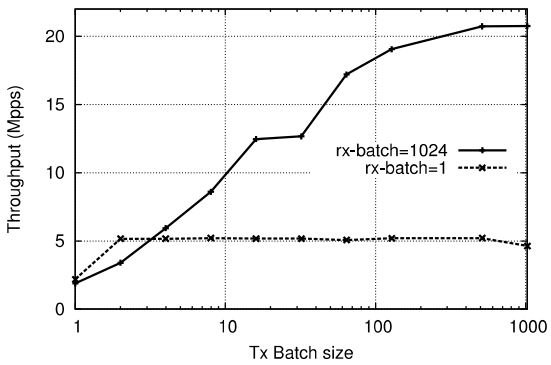

Figure 10: Throughput versus transmit batch size. Kernel uses batch=1024.

batch sizes. The kernel is always using a batch of 1024. The top curve, with a receive batch of 1024, resembles the behaviour of Figure 8. The kernel (and the receiver) indeed behave in a similar way in the two cases, because the sender in the first place is feeding the bridge with only a few packets at a time. The initial region, however, shows lower absolute values because the interval between subsequent invocations of `process_batch()` now includes a complete system call, as opposed to a simple iteration in `tx_handler()`.

The bottom curve, computed with a receive batch of 1 packet, gives a better idea of where the bottleneck lies depending on the tx and rx batch sizes. A small batching factor on the receiver will definitely limit throughput no matter how the other components work, but the really poor operating point – about 2 Mpps in these experiments – is *when the entire chain processes one packet at a time.* It is unfortunate that this is exactly what happens in practice with most network APIs.

5.3 Processing time versus packet size

Coming back to the comparative analysis of different bridging solutions, we now move to the evaluation of the effect of packet size on throughput. Same as in Section 5.2, we run the test between one sender and one receiver connected through a bridge (VALE or native linux bridging), and for variable packet sizes. The measurement is done in the best possible conditions (which, for VALE, means large batches). Also, it is useful to study the effect of packet sizes by looking at the time per packet, rather than absolute throughput.

Figure 11, presents the packet processing time versus the packet size. At these timescales the most evident phenomenon is the cost of data copies. VALE and TAP operate at the speed of the memory bus. This is confirmed by the experimental data, as the two curves have a similar slope. TAP of course has a much higher base value, which is the root cause of its overall poor performance. The curve for NIC-based bridging, instead, is much steeper. This is also expected, because the bottleneck bandwidth (PCI-e or link speed) is several times smaller than that of the memory bus.

The curve for TAP presents a small dip between 60 and 128 bytes, which has been confirmed by a large number of tests. While we have not investigated the phenomenon, it is not unlikely that the code tries (and fails) to optimize the processing of small packets.

5.4 Latency

We conclude our performance analysis with an investigation on the communication latency. This test is mostly pointing out the poor performance of the operating system's primitives involved, rather than the qualities of the bridging code. Nevertheless it is important to know what kind of latency we can expect at best between communicating processes on the same system.

Figure 12 shows the round trip time between two processes talking through VALE (bottom curve) or a TAP bridge (top curve). In this experiment a "client" transmits a single packet and then blocks until a reply is received. The "server" instead issues a first system call to receive the request, and a second one to send the response. The actual amount of processing involved in the process (in the sender, receiver and forwarding code) is negligible, well below 1 μs, which means that the cycle is dominated by the system calls and the interactions with the scheduler (both client and server block waiting for the incoming packet).

As expected, VALE is almost unaffected by the message size (in the range of interest), as there is only a single, and very fast, copy in each direction. TAP instead uses at least two (and possibly three) copies on each direction, hence the different slope.

Care should be taken in comparing these numbers with other RPC mechanisms (e.g. InfiniBand, RDMA, etc.) designed to achieve extremely low latency. These system in fact exploit a number of latency-removal techniques that often require direct hardware access (such as exposing device registers to userspace to save the cost of system calls), are not efficient (such as running a busy-wait loop on the client to remove the scheduler cost), and rely on hardware sup-

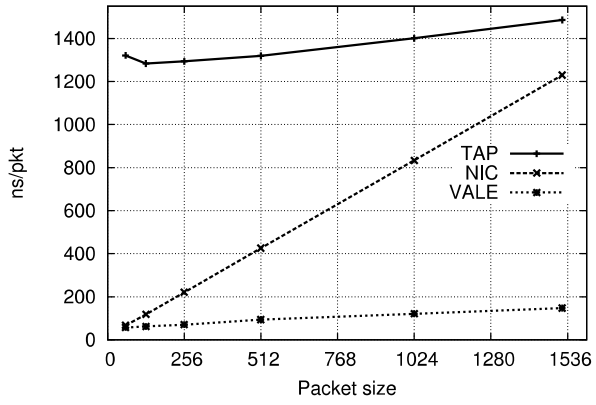

Figure 11: Per-packet time versus packet size. This experiments shows the impact of data copies. VALE and TAP have a similar slope, as they operate at memory bus speed. NIC-based bridging operates at a much lower (PCI-e or link) speed, hence producing a steeper curve.

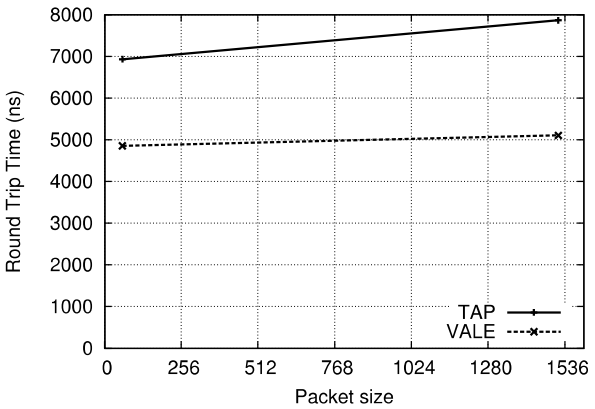

Figure 12: Round trip time in the communication between two processes connected through a linux native bridge (top) or VALE (bottom). In both cases, the times are dominated by the cost of the `poll()` system call on the sender and the receiver.

Configuration	Speed, Mpps TAP		VALE		Notes
	tx	rx	tx	rx	
Raw bridge speed	.90	.90	19.7	19.7	
e1000 QEMU	.018	.014	.023	.023	
e1000 KVM	.020	.020	.024	.024	
netperf virtio QEMU	.012	.010	.023	.012	
virtio KVM	.400	.300	.480	.470	kvm 1.0.1
virtio KVM	.370	.300	1.200	–	kvm 1.1.1
virtio KVM vhost	.600	.580			
pkt-gen e1000 QEMU	.490	.490	**2.400**	**1.850**	
pkt-gen e1000 KVM	.550	.550	**3.470**	**2.550**	
pkt-gen e1000 KVM	.500	.490	**2.300**	**2.000**	1514 b
vmxnet3 ESX	.800	.800	-	-	see [22]
vmxnet3 vSphere	.800	.800	-	-	see [23]

Table 1: Communication speed between virtual machine instances for different combinations of source/sink (netperf if not specified), emulated device, and hypervisor. Some combinations were not tested due to software incompatibilities. The numbers for VMWare are extracted from [23] and [22] and refer to their own software switch.

port (e.g. in RDMA the the server side is completely done in hardware, thus cutting almost half of the processing cost).

Some improvements could be achieved also in our case, e.g. spinning on a shared memory location to wait for data, but our goal is mostly to show what level of performance can be achieved with safe and energy-efficient techniques without hardware assist.

6. RUNNING A HYPERVISOR ON A FAST BRIDGE

The final part of this paper measures how the hypervisor and the guest OS can make use of the fast interconnection provided by VALE. All tests involve:

- a traffic source and sink, running in the guest. We use `netperf`, a popular test tool which can source or sink TCP or UDP traffic, and `pkt-gen`, a netmap-based traffic source/sink which generates UDP traffic;
- an emulated network device. We use `e1000` (emulating a 1 Gbit/s device) and `virtio`, which provides a fast I/O interface that can talk efficiently to the hypervisor;
- a hypervisor. We run our tests with QEMU and KVM;
- a virtual bridge. We use native linux bridging accessed with TAP+vhost, and VALE.

The performance of some of the most significant combinations is shown in Table 1. In some cases we were unable to complete the tests due to incompatibilities between the various options.

All tests were run on an i7-3930K CPU running Linux 3.2 in the host, and with Linux guests.

In the table, the top row reports the raw speed of the bridge for both TAP and VALE. This serves as a baseline to evaluate the virtualized versus native throughput.

The next two rows report standard configurations with netperf and e1000 driver, running on QEMU or KVM. In both cases the packet rate is extremely low, between 14 and 24 Kpps. The interrupt moderation patches [14] bring the transmit rate to 56 and 140 Kpps, respectively, though we are still far from the the throughput that can be achieved on real hardware (about 1.3 Mpps in this configuration). Measurements on QEMU show that, on each packet transmission, the bottom part of the device driver emulation consumes almost 50 μs, a big part of which is spent for handling interrupts.

The virtio driver improves the situation at least when running on top of KVM. Here the use of VALE as a backend gives some improvements[7], although limited by the fact that both source and sink process only one packet at a time.

With such a high per-packet overhead, replacing the TAP bridge with the (much faster) VALE switch can only have a very limited effect on performance. The recipe for improving performance is thus to make better use of the (emulated) network device. The netmap API comes to our help in this case, and the numbers using `pkt-gen` prove that.

When running `pkt-gen` on top of e1000+QEMU (or KVM), the throughput increases by a large factor, even with the TAP bridge (from 20 to ≈500 Kpps). The main reasons are that `pkt-gen` accesses the NIC's registers very sparingly, typically once or twice per group of packets, thus making the emulation a lot less expensive. The improvement is even more visible when running on top of the VALE, which can make use of the aggregation in the guest, and issue a reduced number of system calls to transfer packets from/to the host.

This experiment shows that even without virtio is in principle possible to transfer minimum-size packets at rates that exceed the speed of a 1 Gbit/s interface, and for 1514-byte packets we reach 20 Gbit/s even without virtio.

A lot of these performance improvements come from the use of larger batch sizes when talking to the backend. We are investigating solutions to exploit this operating regime, both with the help of modified device drivers in the host, and implementing mechanisms similar to interrupt moderation within the VALE switch.

6.1 Comparison with other solutions

QEMU and KVM are neither the only nor the fastest hypervisors on the market. Commercial solutions go to great lengths to increase the speed of virtualized I/O devices. To the best of our knowledge, solutions such as vSphere [23] and ESX [22] are among the best performer on the market[8], claiming a speed of about 800 Kpps between two virtual machines (which translates to slightly less than 10 Gbit/s with 1514-byte packets). The vendor's documentation [23] reports up to 27 Gbit/s TCP throughput with jumbo frames (9000 bytes) which should correspond to packet rates in the 4-500 Kpps range.

These numbers cannot be directly compared with the ones we achieved on top of VALE. Even though we get higher packet rates, we are not running through a full TCP stack on the guest; on the other hand we have a much worse virtualization engine and device driver to deal with. What we can still claim, however, is that we are able to achieve the same level of performance of high-end commercial solutions.

7. CONCLUDING REMARKS

We have presented the architecture of VALE, a high speed Virtual Local Ethernet freely available for FreeBSD and Linux, and given a detailed performance evaluation compar-

[7]the low number for KVM+VALE on the receive side seems due to a livelock in KVM 1.1.1 which we are investigating

[8]possibly not the only ones to provide such speeds.

ing VALE with NIC-based bridging and with various existing options based on linux bridging. Additionally, we have developed QEMU and KVM modifications that show how these hypervisors, with proper traffic sources and sinks, can make use of the fast interconnect and achieve very significant speedups.

We should note that VALE is neither limited to use with virtual machines, nor to pure ethernet bridging.

Indeed, the fact that VALE ports use the netmap API, and the availability of `libpcap` emulation library for netmap, means that a VALE switch can be used to interconnect various packet processing tools (traffic generators, monitors, firewalls etc.), and this permits performance testing at high data rates without the need of expensive hardware.

As an example, we recently used VALE switches to test a high speed version of the ipfw and dummynet traffic shaper [5], including the QFQ packet scheduler [6]. In this environment we were able to validate operation at rates exceeding 6 Mpps, well beyond the ability of regular OSes to source or sink traffic, let alone pass it to applications.

Also, the code that implements the forwarding decision (which in VALE is simply a lookup of the destination MAC address) can be trivially replaced with more complex actions, such as a software implementation of an OpenFlow switch. The main contribution of VALE, in fact, is in the framework to move traffic efficiently between ports and the module implementing forwarding decisions.

We are confident that, as part of future work, we will be able to make VALE compatible with better hypervisors and emulated device drivers (virtio and similar ones), and make its speed exploitable also by the network stack in the guest. At the high pps rates supported by VALE, certain operating systems functions (schedulers, synchronizing system calls) and emulator-friendliness need to be studied in some detail to identify possible performance bottlenecks.

The simplicity of our system and its availability should help the identification and removal of performance problems related to virtualization in hypervisors, device drivers and operating systems.

Acknowledgements

This work was funded by the EU FP7 projects CHANGE (257422) and OPENLAB (287581).

8. REFERENCES

[1] `http://virt.kernelnewbies.org/MacVTap`.

[2] `http://www.linux-kvm.org/page/VhostNet`.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP'03*, Bolton Landing, NY, USA, pages 164–177. ACM, 2003.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference ATC'05*, Anaheim, CA. USENIX Association, 2005.

[5] M. Carbone and L. Rizzo. An emulation tool for planetlab. *Computer Communications*, 34:1980–1990.

[6] F. Checconi, P. Valente, and L. Rizzo. QFQ: Efficient Packet Scheduling with Tight Guarantees. *IEEE/ACM Transactions on Networking*, (to appear, doi:10.1109/TNET.2012.2215881), 2012.

[7] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang. Towards high-quality I/O virtualization. In *SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 12:1–12:8. ACM, 2009.

[8] M. Goldweber and R. Davoli. VDE: an emulation environment for supporting computer networking courses. *SIGCSE Bull.*, 40(3):138–142, June 2008.

[9] R. A. Harper, M. D. Day, and A. N. Liguori. Using KVM to run Xen guests without Xen. In *2007 Linux Symposium*.

[10] J. R. Lange and P. A. Dinda. Transparent network services via a virtual traffic layer for virtual machines. In *16th Int. Symposium on High Performance Distributed Computing*, HPDC'07, pages 23–32, Monterey, California, USA, 2007. ACM.

[11] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in xen. In *USENIX Annual Technical Conference ATC'06*, Boston, MA. USENIX Association, 2006.

[12] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. Extending networking into the virtualization layer. In *Proc. of workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[13] H. Raj and K. Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proc. of HPDC'07*, pages 179–188, 2007.

[14] L. Rizzo. Email to qemu-devel mailing list re. interrupt mitigation for hw/e1000.c, 24 july 2012. https://lists.gnu.org/archive/html/qemu-devel/2012-07/msg03195.html.

[15] L. Rizzo. Email to qemu-devel mailing list re. speedup for hw/e1000.c, 30 may 2012. https://lists.gnu.org/archive/html/qemu-devel/2012-05/msg04380.html.

[16] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference ATC'12*, Boston, MA. USENIX Association, 2012.

[17] L. Rizzo. Revisiting network I/O APIs: the netmap framework. *Communications of the ACM*, 55(3):45–51, 2012.

[18] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Infocom 2012*. IEEE, 2012.

[19] L. Rizzo and G. Lettieri. The VALE Virtual Local Ethernet home page. http://info.iet.unipi.it/∼luigi/vale/.

[20] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *Operating Systems Review*, 42(5):95–103, 2008.

[21] S. Tripathi, N. Droux, T. Srinivasan, and K. Belgaied. Crossbow: from hardware virtualized nics to virtualized networks. In *1st ACM workshop on Virtualized infrastructure systems and architectures*, VISA '09, pages 53–62, Barcelona, Spain, 2009. ACM.

[22] VMWare. Esx networking performance. http://www.vmware.com/files/pdf/ESX_networking_performance.pdf.

[23] VMWare. vSphere 4.1 Networking performance. http://www.vmware.com/files/pdf/techpaper/PerformanceNetworkingvSphere4-1-WP.pdf.

[24] J. Wang, K.-L. Wright, and K. Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 109–118, New York, NY, USA, 2008. ACM.

[25] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized i/o devices. In *USENIX 2008 Annual Technical Conference, ACT'08*, pages 15–28, Boston, Massachusetts, 2008. USENIX Association.

[26] B. Zhang, X. Wang, R. Lai, L. Yang, Z. Wang, Y. Luo, and X. Li. Evaluating and optimizing i/o virtualization in kernel-based virtual machine (kvm). In *NPC'10*, pages 220–231, 2010.