# VALE, a switched ethernet for virtual machines

**http://info.iet.unipi.it/~luigi/vale**
**8 june 2012**

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

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

## ABSTRACT

With the use of virtual machines becoming more and more popular, the need for high performance communication between them also grows. 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 implemented 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 regular processes, to achieve 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 as a kernel module which extends our recently proposed netmap framework, and uses similar techniques to achieve high packet rates.

## 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 handling network interfaces and operating them at hardware speed.

As it is often the case, hardware assistance comes handy to achieve better performance. As we will see in Section 2.2, several proposals rely on multiqueue network cards (through PCI passthrough) and external switches to copy data between interfaces. However this solution is expensive (especially in terms of space and power) and not necessarily scalable. Software-only solutions, however, tend to have relatively low performance, especially for small packet sizes.

We 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 with the netmap API, an extremely efficient communication mechanism that we recently introduced [13]. The same API can be trivially used to connect VALE to hardware devices, thus also providing communications with external systems at line rate [12].

VALE is 10..20 times faster than other software solutions based on general purpose OS (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 1500-byte frames (corresponding to more than 70 Gbit/s).

To prove that VALE's performance can be exploited by virtual machines, we then added VALE support to qemu [4] and KVM [7] and measured speedups between 4 and 10 times for applications running on the guest OS (at least when we can overcome the limitations of the emulated device driver, see Section 6), reaching over 2 Mpps with short frames, and about 1.5 Mpps with 1500-byte frames, corresponding to over 18 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 detail 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

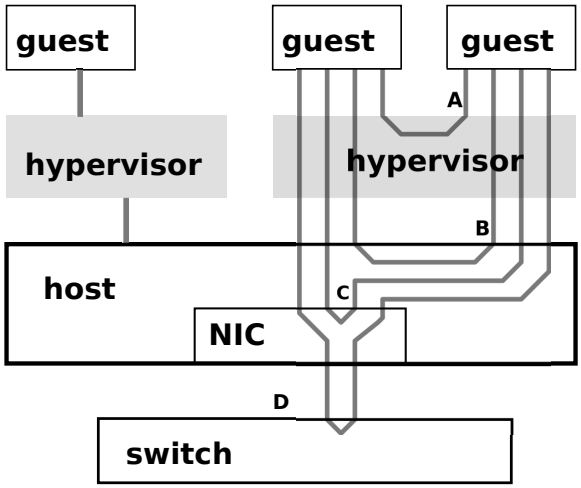


**Figure 1: Communication between guests can follow many different paths. Packet forwarding can happen: A) in the hypervisor; B) in the host, e.g. through native bridging; C) in the NIC, using virtual queues; D) with the support of an external network switch.**

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 measurements 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 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 by virtual machine instances to communicate with each other, with the host, or with 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 any communication between the virtual machine instances ("guest") 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 systems give the virtual machine direct access to the NIC (or some of its queues). In this case, C) the NIC itself can implement packet forwarding between different guests, or D) traffic is forwarded to an external switch which in turn can bounce it back to the appropriate destination.

NIC emulation, as used in A) and B), is normally a low performance mechanism, but 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.

Conversely, direct NIC access (as in cases C and D) is generally much faster, but requires some form of hardware protection to make sure that the guest does not interfere with resources (system memory and NIC registers) to which it has no access rights. We should note that while direct hardware access may help in communication between the guest and external nodes, communication between virtual machines running on the same host might even be faster if done through the hypervisor or the host, as proposed in this paper and also in some commercial products [18, 19].

### 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 latter. We show 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 ordinary processes on the host.

We then extends 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 imply different choices in the communication between guest and hypervisor, and between hypervisor and host/bridging infrastructure. We examine them in turn.

### 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 trigger 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 may be a simple, unprivileged process in the host, as it is the case for QEMU. The hypervisor can then access the network using standard facilities offered by the host, such as TCP or UDP sockets or BPF/libpcap [8]. Pairs of hypervisor processes can bridge their respective guests by encapsulating traffic in a TCP connection or in UDP multicast packets. A common solution is to use tap devices to inject guest produced packets in the host networking stack. Then, tap devices can be connected among them and to real network devices using the software bridges available in the Linux and FreeBSD kernels, or other software bridges such as Open vSwitch [10]. There also exist solutions that run entirely in user space, such as VDE [6]. 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 [18].

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 [9]. Bridging among the guests is performed by a software bridge that connects the driver backends in the guest domain. XenLoop [6] 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.

The KVM [21] hypervisor and the Linux kernel (both as a guest and a host) offer support for virtio [16] paravirtualization. Virtio is a general I/O mechanism based on queues of scatter-gather buffers. The guest and the hypervisor expose 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. Virtio-net notify operations in KVM cause an hardware assisted VM exit to the kvm kernel module. If vhost-net is not enabled, the kvm kernel module then yields control to the KVM process in user space. The KVM process 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 by-passes the KVM process and triggers a kernel thread which directly writes the virtio buffers to the tap. The bridging solutions for this technique are the same as those for full virtualization using tap devices, so performance is ultimately limited by them.

### 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 [20]. 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 support from the hardware to be implemented safely. Moreover, DMA transfers between the guest physical memory and the peripheral benefit from the presence of an IOMMU [5]. More complex scenarios make use of multiqueue NICs to assign a separate queue to each guest [17] or programmable network devices to implement device virtualization in the device itself [11]. These solutions

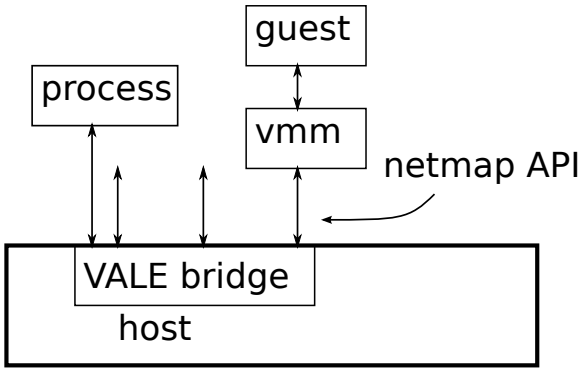


**Figure 2: A VALE local ethernet exposes multiple independent ports to hypervisors and processes, using the netmap API as a communication mechanism.**

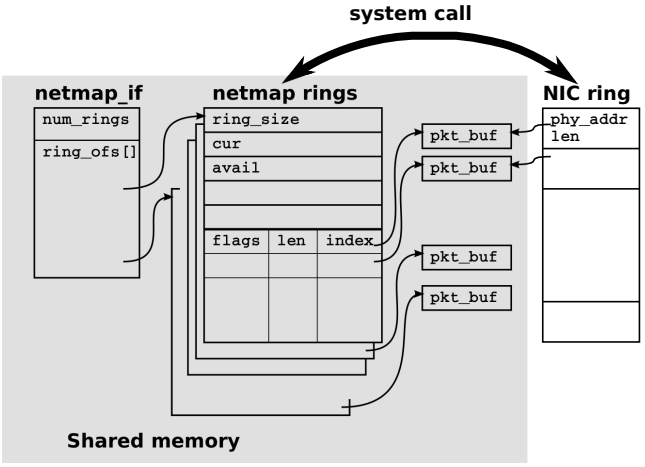


**Figure 3: The memory areas shared between the operating system and processes, and manipulated using the netmap API.**

are generally able to achieve near native performance in the guest, but at the cost or requiring specialized hardware. Bridging can be performed either in the NIC itself, as in [17], or by connecting real external switches.

## 3. VALE, A VIRTUAL LOCAL ETHERNET

Our first objective is to build a software, high performance Virtual Local Ethernet (which we call VALE) as shown in Figure 2, that can provide access ports to multiple clients, be them hypervisors or generic host processes. 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 [13]: 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 [12] we solved these challenges through a series of simple but very effective design choices, aimed at amortizing or removing certain expensive operations from the critical execution paths. Given the similarity to the problem we are addressing here, 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), accessible by the kernel and by userspace processes, hosting packet buffers and their descriptors. A process can gain access to a NIC, and tell the OS to operate in in netmap mode, by opening the special file `/dev/netmap`, issuing an `ioctl()` to select a specific device, and then `mmap()`'ing the region associated to the file descriptor.

The content of this memory region is shown in Figure 3. For each NIC, it contains preallocated buffers for transmit and receive packets, and two[1] circular arrays called *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": these are 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, notifications for an entire batch are reported with a single system calls. This way, the cost of system calls (which can be more than 1 $\mu$s even on modern hardware) is amortized on a large number of packets so it becomes negligible.

The other expensive operations – buffer allocation 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

[1]For card with multiple transmit and receive queues, the shared memory contains one ring per queue.

and the user process.

Netmap is implemented as a kernel module and it is made of two parts. Some generic code implements the basic functions: `open()`, `close()`, `ioctl()`, `poll()/select()`. 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, again permitting 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 ethernet gives each user an independent, virtual interface accessed with the netmap API. Interface names start with the prefix `vale` and each user can define its own private interface. The core of the implementation resides in the netmap backend, which in the case of a VALE port has to implement the traditional algorithm of a learning bridge: the source MAC address of each incoming packet is used to learn on which port stations are located, then the packet is forwarded to zero or more output ports depending on the type of destination address (unicast, multicast, broadcast) and whether the destination is known or not.

The pseudocode for the algorithm is the following:

```
void tx_handler(ring, src_if) {
    cur = ring->cur; avail = ring->avail;
    for (avail-- > 0; cur = NEXT(cur)) {
        pkt = ring->slot[cur].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)
             dst = table[d].src;
        else
             dst = all_ports;
        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);
            }
        }
    }
    ring->cur = cur; ring->avail = avail;
}
```

The above, straightforward implementation, operating on a packet-by-packet basis, however, would have horrible performance (as can be seen from Figure 9, where it would correspond to a batch size of 1). The incoming queue on the destination port, in fact, must be protected against concurrent accesses, and the resulting locking overhead, paid on each packets, would exceed by far the amount of processing strictly necessary for the basic bridging functions and data copies.

## 3.3 Batching

To make the locking less expensive, we use a different sequence of operations: the outer loop is on bridge ports, and the inner loop is on the packets. With this arrangement, the expensive lock is done only once per batch of packet, so its cost is amortized and the CPU is used more efficiently.

Reversing the loops however must be done with care. First of all, operations such as source and destination lookups should be done only once. Secondly, holding the lock on an interface for long periods of time may delay the receiver and other transmitters on the same port, so we may want to process packets in relatively short bursts. Finally, it might be worthwhile to prefetch the buffers we want to read so that we reduce the stalls for data not in cache (see Section 3.4).

These requirements suggest to arrange the code as in the following pseudocode listing.

```
void tx_handler(ring, src_if) {
    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)
            i = pool_flush(pool, i, src_if);
    }
    if (i == netmap_batch_size)
        i = pool_flush(pool, i, src_if);
}

int pool_flush(pool, n, src_if) {
    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;
    }
    for (j = 0; j < max_ports; j++) {
        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);
    }
    return 0;
}
```

A first loop copies batches of metadata from the netmap ring to a temporary array, and issues a `prefetch()` instruction on the buffer. When the array (whose size is configurable at runtime) is full we call a `pool_flush()` function that first does the learning and destination lookup, and then enters the two nested forwarding loops. Figure 4 shows how the temporary array

is filled and scanned. The actual code [15] contains a few more optimizations to improve performance and reduce the size of the critical sections but their discussion is beyond the scope of this paper.

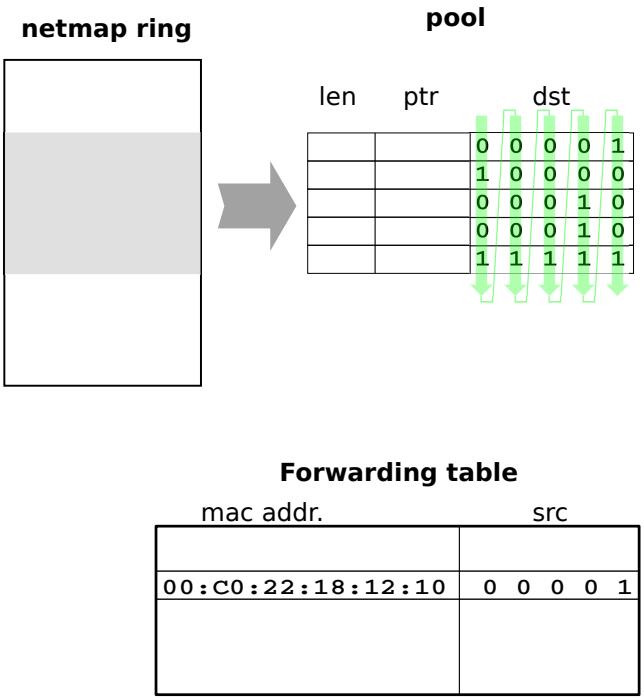


**Figure 4: The data structures used in VALE to support prefetching and reduce the locking overhead. Chunks of the netmap ring are copied to a temporary array, issueing prefetch instructions. Then a second pass implements a source and destination lookup in the forwarding table, and finally the temporary array is scanned in column order to serve each interface in a single critical section.**

### 3.4 Data prefetch

In addition to the reduction of the number of lock operations, the above arrangement of the code lets us prefetch the packets' payload some time before we actually need to access it. This shortens the potential stalls due to cache misses. Although the prefetch scheme we use is still not perfect, it saves about 40 ns per packet, which means doubling the throughput at small packet sizes.

### 3.5 Copying versus sharing

The function `pkt_forward()`, not shown in the listings, must queue the packet on the destination interface(s). There are three possible ways to implement this operations.

1. If the packet has only a single destination, and memory protection between the different ports are not an issue (i.e. all rings for all ports are in the same shared region), we can do zero-copy forwarding by swapping the buffers between the transmit and the receive rings.

2. If we need to implement memory protection, a simple way is to give each bridge port its own private memory region. This means that an actual copy is needed for every port that needs the packet.

3. With some more complications, we could do the copy at most once, and use reference counts to share the buffers between multiple receive queues.

We used the first approach in the original netmap system [12] to achieve very high speed forwarding between interfaces. The motivation there was to support true zero-copy, non data-touching forwarding between interfaces, which could be useful e.g. when bridging between different network technologies.

However, in VALE the conditions to use this approach do not hold. The bridging code has to access the MAC header to learn about sources and determine the destination port for a packet. Also, we cannot trust clients of the bridge not to interfere with each other, so we definitely need to provide some form of protection on memory accesses.

Among the other two alternatives we chose #2, namely copy packets for each destination interface, for the following reasons:

- the (expected) normal mode of operation is that packets are directed to a single output port, so there is not much of a point in trying to optimize the case of multicast/broadcast;

- we use an optimized copy function which is extremely fast. If the data is already in cache (which is likely, due to the the `prefetch()` and the reading of the MAC header), it takes less than 15 ns to copy a 60-bytes packet, and 150 ns for a 1500-bytes packet. Especially for short packets, these times are much smaller than the overhead involved in managing shared buffers among all potential destinations (mostly due to the contention in accessing reference counts from different CPU cores to track usage of the shared buffers);

- shared buffers would also complicate the usage model, requiring two separate memory regions: one, accessible in Read/Write mode, to store transmit buffers, and the other, Readonly, to store receive buffers (we cannot make them writable or clients could modify others' traffic).

## 4. IMPLEMENTATION DETAILS

VALE is implemented as a kernel module, available from [15] for both FreeBSD and Linux as an extension

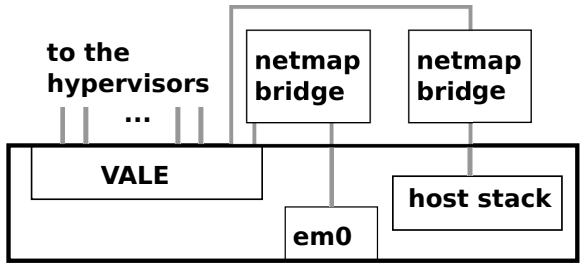


**Figure 5: The connection between VALE, physical network interfaces and the host stack can be built with existing components (netmap bridge) which implement zero-copy high speed transfer between netmap-compatible ports.**

of the netmap module. Thanks to the modular implementation of the netmap system, 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 day one.

In the current prototype, and provided enough memory is available, the maximum number of ports is limited to 64, so we can use bitmaps to represent sets of interfaces, as used in the temporary table in Figure 4 to implement the forwarding. In principle the limit could be extended, but for all practical purposes we do not believe that a software solution can be effective with so many active ports.

At the moment we do not support features such as IGMP snooping to implement selective forwarding of multicast traffic. However, this is a straightforward addition which is already fully supported by our data structures.

The size of the temporary array is configurable in software, to choose the desired tradeoffs between latency and throughput. More details on this will be presented in Section 5.1.3.

## 4.1 External communication

As presented, VALE implements only a local ethernet whose use is limited to hypervisors and processes hosted on the same host. The connection with the external network 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 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 [7], 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 when the VALE file descriptor becomes ready.

Both qemu and KVM manage access to 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 subsequent modifications will be handled the next time the hypervisor will call `poll()`. This enables the hypervisor to exploit the batching that reduces the I/O overhead to acceptable values.

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 [14], and we found similar issues in this work.

Specifically, the code involved in the guest-size 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.

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` [16] or other proprietary APIs [19].

## 4.3 Guest issues

A fast network backend, and a fast hypervisor do not mean 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 [12], 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 communication mechanism with the network card.

As a consequence, for some of our high speed tests we will use, in the guest, a network device in netmap mode. 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 on CPU and memory speeds, as well as on compiler optimizations that may cause stalls in 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.

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:** Intel i7-2600K (4 core, 3.2 GHz) + FreeBSD-9; Intel i7-870 (4 core, 2.93 GHz) + FreeBSD-10; Intel i5-750 (4 core, 2.66 GHz) + Linux 3.2.12. 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 (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.

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 prevent a reasonable performance evaluation.

## 5.1 Bridging performance

The first set of tests analyzes the performance of various software bridging solutions. The most significant performance parameter 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 per packet and relatively independent of packet size. Bridges and switches, however, need also a characterization in bps because they operate at very high rates and often hit other bottlenecks such as memory or link bandwidth.

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.

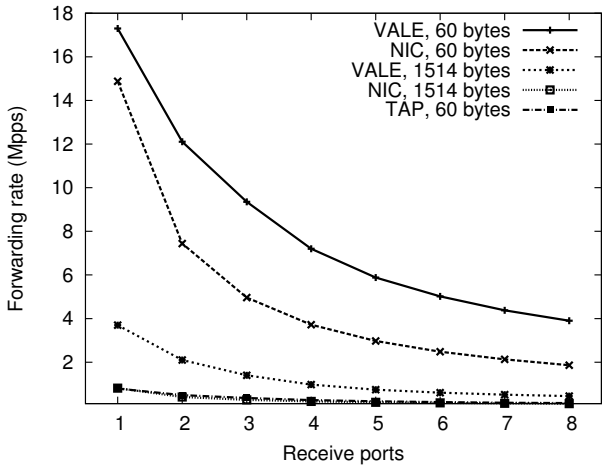


**Figure 6: Forwarding rate depending on the number of destinations. VALE beats even NIC-based bridging, with over 17 Mpps (vs. 14.88) for 60-byte packets, and over 6 Mpps (vs. 0.82) for 1514-byte packets. Tap is at the bottom, peaking at about 0.8 Mpps in the best case.**

In Figure 6 we compare the forwarding throughput when using 60 and 1514 byte packets, for three tech-

nologies: TAP plus native linux bridging[2], our VALE bridge, and NIC-supported bridging (in this case we report the theoretical maximum throughput). The graph shows how the throughput changes for traffic replicated to multiple output ports (the case for a single active destination correspond to the 1-port throughput). This is a worst-case situation that occurs when a bridge is dealing with broadcast traffic, or when the destination of a unicast packet is unknown.

The best option for Linux bridging have a peak rate of about 0.80 Mpps, decreasing as the number of active receivers increases.

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 vact, as we measured in [12], 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 17.3 Mpps for a single destination and 60-byte packets, again decreasing as the number of receiver grows. Here the bottleneck is given by the combination of CPU cycles (needed to do the packet copies) and memory bandwidth.

The difference for 1500-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.

The performance of VALE is heavily dependent on the batch size used internally and externally (see Section 3.3). The numbers in the graph are for a batch of 1024 packets. The dependency on the batch size is shown in Section 5.1.3, but even at a batch size of 128 VALE can match or outperform NIC-based forwarding.

The huge difference in throughput between VALE and other solutions depends mostly on the use of the (much more efficient) netmap API, which amortizes the system call costs over batches of packets, and makes in-kernel management much less expensive.

### 5.1.1 Per packet time

Figure 7 shows the per-packet processing time depending on the number of ports. This gives a better idea of the time budget involved with the various operations (address lookups, data copies, prefetch...). We benchmarked the packet-copy costs separately, and when data is in the L1 cache, we measured around 15 ns for 60-byte packets, and 150 ns for 1514 byte packets; so we would ideally expect those values as the difference between subsequent points in the graph.

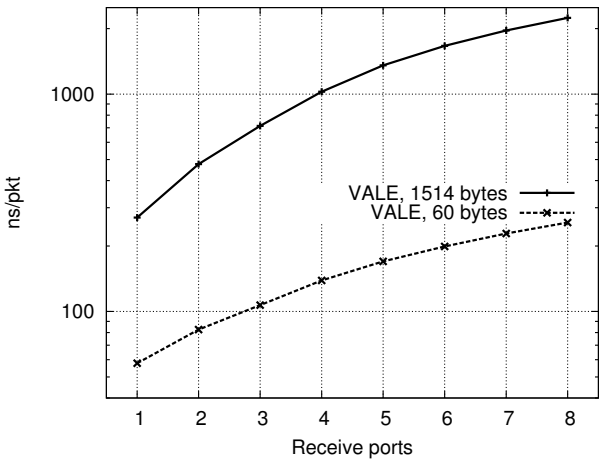


**Figure 7: Per-packet time depending on the number of destinations for VALE. See text in Section 5.1.1 for details.**

In practice, we see that the differences are larger than that. Part of the reason is that data are not guaranteed to be in cache. Also, there is some amount of contention between the sender and the receivers when accessing the receive queues. Finally, there are significant (at these timescales) loop overheads in running the forwarding code.

The base value (for 1 receive port) is also affected by the cost of computing the source and destination lookups, which we measured at around 15 ns using the Jenkins hash function, taken from the FreeBSD bridging code.

### 5.1.2 Variable packet size

We now measure the behaviour of various bridging solutions for different packet sizes. Given that the common/reasonable configuration involves a single active destination, we use that as the test configuration.

Figure 8, presents the variations of the packet processing time depending on 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, so the two curves have a similar slope, although TAP has a much higher base value. The curve for NIC-based bridging, instead, grows much faster 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

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

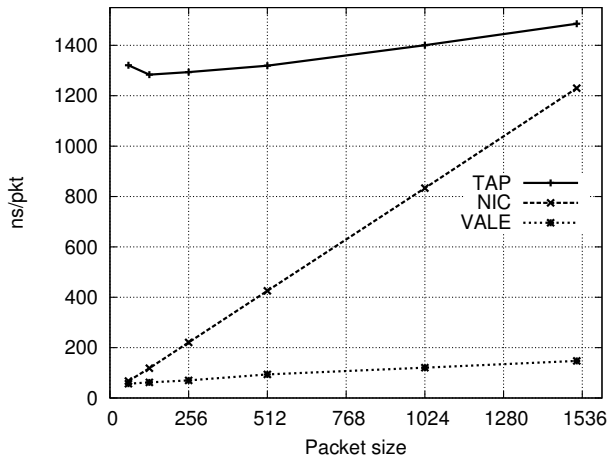


**Figure 8: Per-packet time depending on packet size. This experiments shows the impact of data copies. VALE and TAP have a similar slope, as they operate at memory bus speed. NIC-based bridging operates at a much lower PCI-e or link speed, hence producing a steeper curve.**

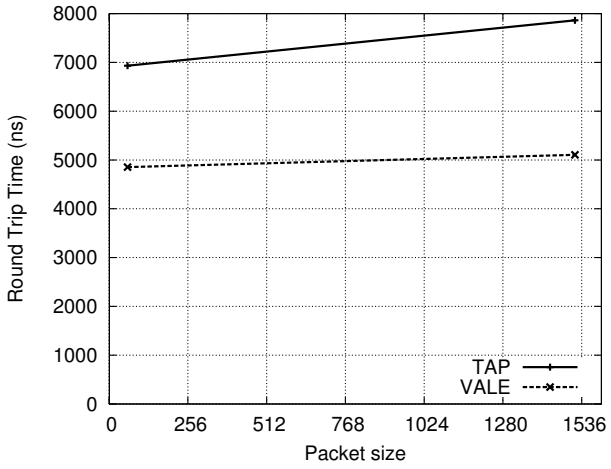


**Figure 10: Round trip time in the communication between two processes connected through a linux native bridge (top) or VALE (bottom). In both cases, the times are dominated by the cost of the `poll()` system call on the sender and the receiver.**

phenomenon, it is not unlikely that there are different code paths (and optimizations – in this case failed ones) traversed depending on different packet sizes.

### *5.1.3 Batch size*

A significant performance boost in VALE, netmap and many other systems comes from the ability to transfer multiple packets in each system call, as well as process multiple packets while holding the lock on an interface. Figure 9 shows the influence of the batch size on processing time for the receive side. For small batch sizes we are dominated by the combination of high system call costs, and excessive locking overhead. Both areas will need some investigation in the future, as there is potential for significant performance improvements, not just for VM bridging but for accelerating the operating system in general.

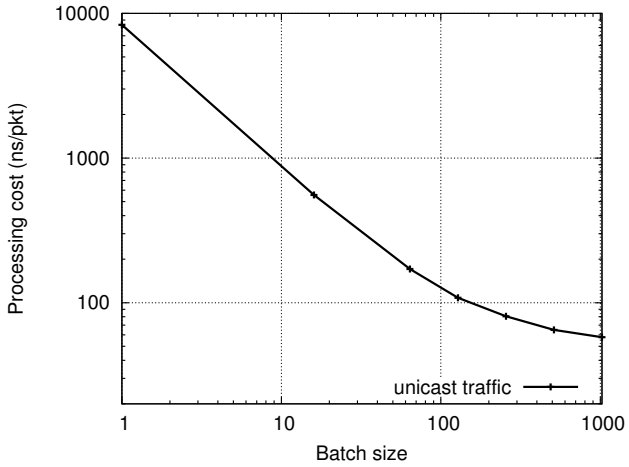


**Figure 9: Average per-packet processing time on the receive side of VALE, depending on the batch size. At small batch sizes, the majority of the cost comes from the system call and lock contention on the netmap ring.**

### *5.1.4 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 10 shows the round trip time between two processes talking through VALE (bottom curve) or a TAP bridge (top curve) as the communication channel. The processes used for the test invoke a `poll()` to determine when a message is available. Slightly better results (trimming almost 1 $\mu$s on each side) could be achieved by moving to a busy-wait scheme (using an `ioctl()` for VALE, or a non blocking `read()` for TAP).

As expected, VALE has a barely visible dependency on the message size, as there is only a single copy involved in each direction. TAP instead uses at least two (and possibly three) copies on each direction, hence the different slope. In both cases, the numbers (5..8 $\mu$s) are dominated by the system call costs.

| SRC DEV Hypervisor | Speed, Mpps TAP | | Speed, Mpps VALE | | Notes |
|---|---|---|---|---|---|
| | tx | rx | tx | rx | |
| Raw bridge speed | .78 | .78 | 9.0 | 9.0 | |
| netperf e1000 qemu | .01 | .01 | .01 | .01 | |
| netperf e1000 KVM | .02 | .02 | .02 | .02 | |
| netperf virtio qemu | .02 | .01 | | | |
| netperf virtio KVM | .70 | .34 | | | |
| pkt-gen e1000 qemu | .21 | .21 | **1.78** | **1.41** | |
| pkt-gen e1000 KVM | .21 | .21 | **2.64** | **1.86** | |
| pkt-gen e1000 KVM | | | **1.59** | **1.49** | 1500 b |
| netperf vmxnet3 ESX | .80 | .80 | - | - | see [18] |
| netperf vmxnet3 vSphere | .80 | .80 | - | - | see [19] |

**Table 1: Communication speed between virtual machine instances for different combinations of source-sink, emulated device, and hypervisor. Tests are on an i5-750, using Linux as both host and guest OS. Some combinations were not tested due to software incompatibilities. The numbers for VMWare are extracted from [19] and [18] and refer to their own software switch.**

# 6. RUNNING A HYPERVISOR ON A FAST BRIDGE

The final part of our work measures how the hypervisor and the guest OS can make use of the fast interconnection we have built. All tests involve:

- a traffic source and sink. Our options are `netperf`, a popular test tool which can source or sink TCP or UDP traffic, and `pkt-gen`, a netmap-based traffic sink-source 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 to the hypervisor without exiting the virtual machine;
- a hypervisor. We ran our tests with qemu and KVM.
- a virtual bridge, in our case 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 modules. As an example, as of this writing the VALE backend is still not compatible with the virtio driver.

All tests were run on an i5-750 CPU running Linux in the host, and with Linux guests. The host is slightly slower than the one used in previous tests, so the top row reports the raw speed of the bridge for both TAP and VALE.

The next two rows report standard configurations with netperf and e1000 driver, running on both KVM and qemu. In both cases the packet rate is extremely low, between 10 and 20 Kpps. Note that with large packets (1500 bytes), 20 Kpps still translate to 240 Mbit/s which is a reasonable performance for some applications, but not comparable with the speed (around 500 Kpps) that can be achieved on the host.

Measurements on qemu show that, on each packet transmission, the bottom part of the device driver emulation consumes over 50 $\mu$s. The virtio driver improves the situation at least on top of KVM, where it can almost match the speed of the underlying TAP-based bridge in transmit mode, and is slightly lower in receive mode. The combination virtio+qemu, however, still appears extremely slow.

With such a high per-packet overhead, replacing the TAP bridge with the (much faster) VALE one has no 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), our throughput increases by a large factor, even with the TAP bridge. The main reason is that `pkt-gen` accesses the NIC's registers very sparingly, typically once or twice per group of packets, with significant benefits on the throughput. 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.

In particular, two VMs on top of qemu and connected using a VALE backend can send at 1.78 Mpps and receive at 1.41 Mpps, respectively. On top of KVM we have another small (or large ?) improvement reaching 2.64 Mpps (tx) and 1.86 Mpps (rx), well beyond the speed of a 10 Gbit/s interface.

Another significant data point is the throughput with 1500 byte packets, where we can transmit 1.59 Mpps (almost 20 Gbit/s) and receive 1.49 Mpps (∼18 Gbit/s).

## 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 improve performance of virtualized I/O devices. To the best of our knowledge, solutions such as vSphere [19] and ESX [18] are among the best performer on the market[3], claiming a speed of about 800 Kpps between two virtual machines (which translates to slightly less than 10 Gbit/s with 1500-byte packets). The vendor's documentation [19] report up to 27 Gbit/s TCP throughput with jumbo frames (9000 bytes) which should correspond to a similar or slightly lower packet rate.

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

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

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. CONCLUSIONS AND FUTURE WORK

We have presented the architecture of VALE, a high performance Virtual Local Ethernet, and given a detailed performance evaluation comparing VALE with NIC-based bridging and with the best existing option 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 speedups of 5..10 times.

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 be accessible also from within the TCP stack in the guest. We also believe that at such operating speeds, certain operating systems functions (schedulers, synchronizing system calls) and emulator-friendlyness 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.

## 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 *19th ACM Symposium on Operating Systems Principles, 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] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang. Towards high-quality i/o virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 12:1–12:8, New York, NY, USA, 2009. ACM.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

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

[17] 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.

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

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

[20] 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.

[21] 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.