mSwitch: A Highly-Scalable, Modular Software Switch

Michio Honda[§] [*], Felipe Huici[†], Giuseppe Lettieri[‡], Luigi Rizzo[‡]
NetApp Inc.[§] NEC Europe Ltd.[†], Universita‘ di Pisa[‡]
michio@netapp.com, felipe.huici@neclab.eu, {g.lettieri,rizzo}@iet.unipi.it

ABSTRACT

In recent years software network switches have regained eminence as a result of a number of growing trends, including the prominence of software-defined networks, as well as their use as back-ends to virtualization technologies, to name a few. Consequently, a number of high performance switches have been recently proposed in the literature, though none of these *simultaneously* provide (1) high packet rates, (2) high throughput, (3) low CPU usage, (4) high port density and (5) a flexible data plane. This is not by chance: these features conflict, and while achieving one or a few of them is (now) a solved problem, addressing the combination requires significant new design effort.

In this paper we fill the gap by presenting mSwitch. To prove the flexibility and performance of our approach, we use mSwitch to build four distinct modules: a learning bridge consisting of 45 lines of code that outperforms FreeBSD's bridge by up to 8 times; an accelerated Open vSwitch module requiring small changes to the code and boosting performance by 2.6-3 times; a protocol demultiplexer for user-space protocol stacks; and a filtering module that can direct packets to virtualized middleboxes.

Categories and Subject Descriptors

C.2.6 [**Computer-communication Networks**]: Internetworking; D.4.4 [**Operating Systems**]: Communications Management; D.4.8 [**Operating Systems**]: Performance

General Terms

Algorithms, Design, Performance

Keywords

Software switch, Scalability, Programmability

[*]The work was mostly performed while at NEC.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.
Copyright is held by the owner/author(s).
SOSR '15 June 17–18, 2015, Santa Clara, CA, USA
ACM 978-1-4503-3451-8/15/06.
http://dx.doi.org/10.1145/2774993.2775065

1. INTRODUCTION

Software packet switching has been part of operating systems for almost 40 years, but mostly used as a prototyping tool or as a low-cost alternative to hardware-based devices. A number of recent events have brought renewed importance to software switches: the increased importance of virtualization technologies which rely on a back-end software switch to multiplex packets between physical interfaces and containers or VMs; the growing prominence of middleboxes [25, 24] in today's networks along with their virtualization (NFV [6, 14]); and the widespread use of Software-Defined Networking, for which software switches provide significant flexibility.

In the SDN field in particular, a recent trend named SDNv2 and targeting operator networks calls for a model comprising a simple, hardware-based core network providing limited functionality, with complex functionality being pushed into software switches at the edge [23]; a similar call has been made in the context of data centers [2]. Another recent and important trend is the growing popularity of containers, which will require higher port densities in software switches, beyond what current solutions offer.

These trends suggest the need for a software switch that enables SDN research and experimentation on commodity servers by allowing for easy customization of switching fabrics while yielding production-quality packet rates. This requires simultaneous support for (1) a programmable data plane (beyond what Openflow supports), (2) high throughput, (3) high packet rates, (4) efficient CPU usage and (5) high port density. These features are difficult to harmonize, and there are, to the best of our knowledge, no available solutions, either as products or research prototypes, that simultaneously provide all of them. Next, we briefly cover existing switches.

1.1 State of the Art

Software switches embedded in commodity OSes (Linux and FreeBSD) have a relatively slow data plane, offering decent performance (up to 30-40 Gbit/s) only for bulk TCP traffic using large (32-64 K) TSO segments. Packet rates are limited to 1-2 Mp/s, significantly below the peak 14.88 Mp/s for a 10 Gbit/s interface, due to the high per-packet I/O costs. This limitation is shared by the in-kernel version of Open vSwitch [16] which provides flexible processing features but inherits the low performance of the OS's dataplane [20].

In contrast, high packet rates are supported by "OS bypass" or "network stack bypass" frameworks such as DPDK [11]

and netmap [19] which rely heavily on batched I/O. DPDK uses direct hardware access from userspace (UIO) and a number of optimizations (e.g., huge pages, vector instructions, direct cache access) to achieve impressive packet I/O rates and latency. However, the interconnection between clients in DPDK-based switches either gives up protection (as in NetVM [10], which provides a slow fallback path through the host stack for untrusted VMs), or relies on multiqueue NICs with SR-IOV to implement a protected datapath, as in IX [3]. Further, the use of active polling not only keeps one (or more) cores always busy, but also makes it difficult to scale to large number of ports due to an explosion of the amount of unnecessary polls on idle resources [1]. Two switches based on DPDK are CuckooSwitch [26], which achieves high throughput when handling very large numbers of L2 rules but is targeted at replacing hardware Ethernet switches and so does not have a flexible switching logic; and DPDK vSwitch [13], which takes the Open vSwitch code base and accelerates it by using DPDK, inheriting the disadvantages of DPDK already mentioned.

The VALE software switch [21] based on the netmap framework [19] uses a different approach: an in-kernel module provides a high packet rate, protected datapath between virtual ports using memory copy. Full support for interrupts and synchronization avoids the need for active polling loops (so that the CPU load is proportional to the actual load), but since there is no free lunch, VALE pays this with slightly worse latency and throughput than pure zero-copy solutions[2]. The original VALE only supports a few tens of ports (the Linux/FreeBSD datapaths in principle scale to very large numbers, but at 1/10 or lower throughput) and has limited output port parallelism (same as other published software switches).

Hardware-based solutions relying on SR-IOV give VMs direct access to one of the NIC's "virtual functions", thus potentially supporting wire speed and high packet rates (at least with one of the high performance frameworks mentioned above), as well as isolating virtual ports thanks to data copies done by the NIC. Arrakis [17], for instance, leverages this hardware feature, and so does IX [3]. The convenience in delegating switching and protection to the NIC comes with three limitations: low aggregate bandwidth, limited filtering capabilities, and reduced scalability. The switch in the NIC sits behind a PCIe bus, so the aggregate bandwidth for all VMs is limited to the bus' speed (20-40 Gbit/s typically); this is less than what software switches offer for a single port (e.g., Hyper-Switch [18] reports up to 81 Gbit/s between two VMs; Linux/OVS datapath gives more than 30 Gbit/s). The fact that filtering is limited to whatever the NIC supports aggravates the bandwidth problem, requiring additional filtering to be performed in software after an expensive transfer through the bus. Finally, scalability is directly related to the number of virtual functions supported by the hardware, which is not always large (e.g., the popular 82599 chipset from Intel only supports 64 VFs).

1.2 Our contributions

The system we propose in this paper, mSwitch, *simultaneously* addresses all five requirements that we consider necessary to deploying efficient, scalable software switches in next-generation SDN networks: (1) a programmable data plane, (2) high throughput, (3) high packet rates, (4) efficient CPU usage and (5) high port density.

mSwitch extends the VALE software switch in various ways, providing features not available in (and which could not be easily added to) other solutions proposed in the literature. Among our contributions we have:

- A clear separation between a high-performance switching fabric (the dataplane that switches packets between ports) and the switching logic (which decides a packet's destination port). The latter can be implemented with C functions and plugged at runtime into the switching fabric without sacrificing performance, thus providing the flexibility needed to accommodate different use cases and drive experimentation.
- Scalability to large numbers of virtual ports. To the best of our knowledge, this is the first paper to evaluate a software switch with up to 120 ports.
- Parallel access to destination ports, to achieve high throughput when multiple senders (e.g., processes, containers or VMs) collide onto a single destination port (e.g., a NIC). mSwitch reaches speeds of 75 Mp/s (small packets) or 476 Gbit/s (large segments) to a single destination port on an inexpensive server.

We validate our approach by implementing or porting a range of different types of mSwitch switching logics: a learning bridge consisting of 45 lines of code which outperforms the FreeBSD one by up to 8 times; an Open vSwitch module comprising small code changes and resulting in a 2.6-3 times boost; a filtering module that allows mSwitch to act as a back-end switch to direct packets to virtualized middleboxes (and drop unwanted ones); and a module used to support user-level network stacks.

mSwitch and all of these modules are open source and available for FreeBSD and Linux.

1.3 A Note on Novelty

At first sight, mSwitch might seem like a trivial extension to VALE, with the techniques used appearing to be well known, simple, or too incremental for publication. We humbly suggest that this is not the case, as explained below.

VALE [21] was published in December 2012 and widely used since then. Despite this, none of the subsequent papers on the same topic (e.g., CuckooSwitch, NetVM, HyperSwitch, Arrakis, IX) provides (and could not, as designed) the features we introduce in mSwitch: hardware-based solutions cannot possibly scale to higher bandwidths; DPDK-based solutions fail to provide isolation at the same time as efficiency. In truth, with one exception (Hyper-Switch [18], which performs well for large TSO segments), related work does not address dataplane performance other than by reusing existing solutions (SR-IOV or DPDK) and inheriting their limitations.

2. REQUIREMENTS AND PROBLEM SPACE

As mentioned, there are a number of requirements for a software switch to be viable in next generation SDN networks and as NFV back-ends, including flexibility, high throughput, packet rates and port density, and reasonable CPU utilization. Before developing yet another software switch, it

[1] The polling thread could of course be stopped periodically, but that would trash the latency.

[2] We note that netmap does provide a true zero-copy point-to-point channel called a netmap pipe.

makes sense to see if any of the existing solutions meet these requirements.

Throughput: To study the throughput of a number of software switches, we ran a simple experiment which measured the forwarding rate for packets of different sizes between two 10 Gbit/s ports (Intel 82599 chipset cards) using a single CPU core (Figure 1).

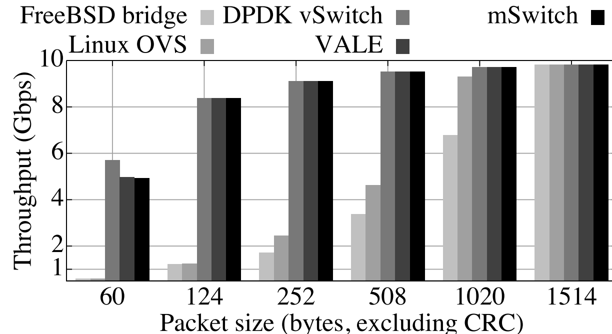


Figure 1: Forwarding throughput between two 10 Gbit/s NICs for major software switches.

The native FreeBSD switch and Open vSwitch only reach line rate at moderately large packet sizes (the Linux bridge module produces similar results). In all, such "standard" switches suffer from a combination of inefficiencies including heavyweight packet data representation, per-packet I/O semantics and queuing.

More recent offerings such as VALE [3] and DPDK vSwitch [4] yield much higher performance by eliminating these shortcomings. As shown in the graph, these, along with mSwitch, yield line rate for almost all packet sizes. DPDK vSwitch edges mSwitch out by a small margin for short packets, but as we will see next, at the cost of utilizing significantly more CPU cycles. CuckooSwitch [26], based on DPDK, provides high throughput for all packet sizes. Finally, Hyper-Switch relies on unoptimized Open vSwitch code to process packets, and so results in lower throughput speeds for non jumbo-sized packets.

CPU Usage: To get a sense of the switches' efficiency, we measured the utilization of all 12 CPU cores in our system while forwarding packets at high rates using DPDK vSwitch or VALE (the other switches do not yield high throughput or are not publicly available). In terms of CPU usage, the fundamental feature of DPDK vSwitch, and indeed, of any DPDK-based package, is that DPDK's poll-mode driver results in 100% utilization irrespective of the traffic rates being processed. In contrast, VALE relies on interrupts, so that user processes are woken up only on packet arrival. In our experiments, for the 10 CPU cores handling packets this results in a *cumulative* CPU utilization of about 140% for mSwitch (which also adopts an interrupt-based model) and a much higher but expected 1,000% for DPDK vSwitch (the full results for these experiments are in Section 4.2).

High Density: Despite its high throughput, VALE, as we will show in Section 4, scales poorly when packets are forwarded to an increasing number of ports, and the throughput further drops when packets from multiple senders are sent to a common destination port; both of these are common scenarios for a back-end virtualization switch containing a single NIC and multiple containers or VMs.

For DPDK vSwitch, its requirement of having a core dedicated to each port limits its density. While it is possible to have around 62-78 or so cores on a system (e.g., 4 AMD CPU packages with 16 cores each, minus a couple of cores for the control daemon and operating system, or 4 Intel 10-core CPUs with hyper threading enabled), that type of hardware represents an expensive proposition, and ultimately it may not make sense to have to add a CPU core just to be able to connect an additional VM or process to the switch. Finally, CuckooSwitch targets physical NICs (i.e., no virtual ports), so the experiments presented in that paper are limited to 8 ports total.

Flexibility: Most of the software switches currently available do not expressly target a flexible forwarding plane, limiting themselves to L2 forwarding. This is the case for the standard FreeBSD and Linux bridges, but also for newer systems such as VALE and CuckooSwitch. In contrast, Open vSwitch supports the OpenFlow protocol, and as such provides the ability to match packets against a fairly comprehensive number of packet headers, and to apply actions to matching packets. However, as shown in Figure 1 and in [20], Open vSwitch does not yield high throughput.

	Throughput	CPU Usage	Density	Flexibility
FreeBSD switch	×	√	√	×
Linux switch	×	√	√	×
Open vSwitch	×	√	√	√
Hyper-Switch	×	√	×	√
DPDK vSwitch	√	×	×	√
CuckooSwitch	√	×	×	×
VALE	√	√	×	×

Table 1: Characteristics of software switches with respect to throughput, CPU usage, port density and flexibility.

Summary: Table 1 summarizes the characteristics of each of the currently available software switches with respect to the stated requirements; none of them simultaneously meet them.

3. mSwitch DESIGN

The discussion in the previous section leads us towards a number of design principles. First, in terms of throughput, there is no need to re-invent the wheel: several existing switches yield excellent performance, and we can leverage the techniques they use such as packet batching [5, 7, 19, 21], lightweight packet representation [7, 19, 11] and optimized memory copies [21, 18, 11] to achieve this.

In addition, the switch's data plane should be programmable while ensuring that this mechanism does not harm the system's ability to quickly switch packets between ports. This points towards a split between highly optimized switch code in charge of switching packets, and user-provided code to decide destination ports and potentially modify or filter packets.

Further, to obtain relatively low CPU utilization and flexible core assignment we should opt for an interrupt-based model (and variants of it such as NAPI [22]), such that idle

[3] We modified the original VALE slightly so that it supports connecting NICs to switch ports.
[4] We used DPDK vSwitch 1.1.0-27.

ports do not unnecessarily consume cycles that can be better spent by active processes, containers, or VMs. This is crucial if the switch is to act as a back-end, and has the added benefit of reducing the system's overall power consumption.

Finally, we should design a forwarding algorithm that is lightweight and that, ideally, scales linearly with the number of ports on the switch; this would allow us to reach higher port densities than current software switches. Moreover, for a back-end switch muxing packets from a large number of sending virtual ports to a common destination port (e.g., a NIC), it is imperative that the forwarding algorithm is efficiently able to handle this incast problem.

3.1 Starting Point

Having identified a set of design principles, the next question is whether we should base a solution on one of the existing switches previously mentioned, or start from scratch. The Linux, FreeBSD and Open vSwitch switches are non-starters since they are not able to process packets with high enough throughput. CuckooSwitch provides the best performance, but it has high CPU utilization, does not support virtual ports, targets L2 forwarding only and, on a more practical level, source code for it is not available. DPDK vSwitch suffers from the same CPU utilization issue and has a difficult programming environment.

VALE is limited to at most 64 ports and can only do L2 forwarding. However, it *is* based on an interrupt model and provides high throughput (at least for small port numbers), two of our four design principles. It is also actively maintained and provides the widest compatibility with different operating systems and hardware devices. The question then is whether it is possible to re-architect VALE to achieve high port densities and provide a programmable data plane while retaining its high performance.

We answer in the affirmative, and spend the rest of the section providing an in-depth description of mSwitch's architecture, including the algorithms and mechanisms that allow it to achieve high performance, high port densities and the ability to have different modules be seamlessly plugged into the switch, even at run-time.

3.2 Architecture

mSwitch is based around the principle of a split data plane consisting of a switch *fabric* in charge of moving packets quickly between ports; and the switch *logic*, the modular part of mSwitch which looks at incoming packets and decides which their destination port(s) should be. This split allows mSwitch to provide a fast fabric that yields high throughput, while giving users the ability to program the logic without having to worry about the intricacies of high performance packet I/O.

Figure 2 shows mSwitch's overall architecture. mSwitch can attach virtual or physical interfaces as well as the protocol stack of the system's operating system. From the switch's point of view, all of these are abstracted as *ports*, each of which is given a unique index in the switch. When a packet arrives at a port, mSwitch *switches* it by relying on the (pluggable) packet processing stage to tell it which port(s) to send the packet to[5]. In addition, multiple instances of mSwitch can be run simultaneously on the same

[5]The processing function is completely arbitrary, so it can also apply transformations to the packet such as encap/decap, NAT, etc.

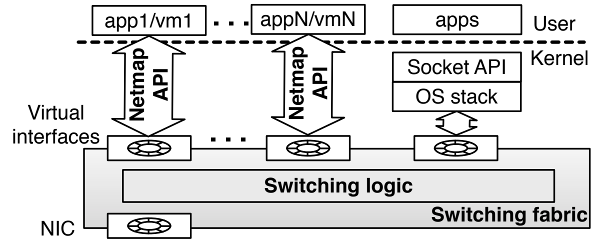


Figure 2: mSwitch architecture: the switch fabric handles efficient packet delivery between ports, while the switch logic (forwarding decisions, filtering etc.) is implemented through loadable kernel modules.

system, and ports can be created and connected to a switch instance dynamically.

Virtual ports are accessed from user space using the netmap API: they can be connected to virtual machines (e.g., QEMU instances), or generic netmap-enabled applications. The netmap API uses shared memory between the application and the kernel, but each virtual port uses a separate address space to prevent interference between clients. Physical ports are also connected to the switch using a modified version of the netmap API, providing much higher performance than the default device drivers. Each port can also contain multiple packet *rings* which can be assigned to separate CPU cores in order to scale the performance of a port.

Ports connected to the host network stack are linked to a physical (NIC) port. Traffic that the host stack would have sent to the NIC is diverted to mSwitch instead, and from here can be passed to the NIC, or to one of the virtual ports depending on the packet processing logic running on the switch. Likewise, traffic from other ports can reach the host stack if the switch logic decides so. For instance, the Multi-Stack module presented in Section 5 directs incoming traffic matching 3-tuple rules to specific ports, and all other traffic to the host stack; the latter allows us to keep compatibility with socket-based applications using the host stack (we of course have to take care that there are no 3-tuple rules with an IP address/port in use by the host stack).

Packet forwarding is always performed within the kernel and in the context of the thread that generated the packet. This is a user application thread (for virtual ports), a kernel thread (for packets arriving on a physical port), or either for packets coming from a host port, depending on the state of the host stack protocols. Several sending threads may thus contend for access to destination ports; in Section 3.3 we discuss a mechanism to achieve high throughput even in the face of such contention.

mSwitch always copies data from the source to the destination netmap buffer. In principle, zero-copy operation could be possible (and cheap) between physical ports and/or the host stack, but this seems an unnecessary optimization: for small packet sizes, once packet headers are read (to make a forwarding decision), the copy comes almost for free; for large packets, the packet rate is much lower so the copy cost (in terms of CPU cycles and memory bandwidth) is not a bottleneck. Cache pollution might be significant, so we may revise this decision in the future.

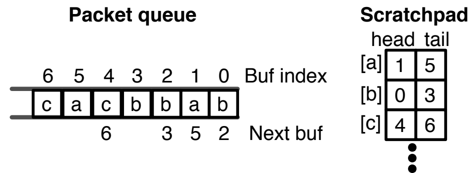


Figure 3: mSwitch's list-based packet forwarding. Packet buffers are labeled 0 to 6 and together constitute a batch. Destination port indices are labeled a to c.

Copies are instead the best option when switching packets from/to virtual ports: buffers cannot be shared between virtual ports for security reasons, and altering page mappings to transfer ownership of the buffers is immensely more expensive.

3.3 Switch Fabric Algorithm

In order to scale to large numbers of ports, mSwitch needs a different algorithm than VALE's. For reference, VALE's algorithm works, at a high level, as follows. For each batch of packets, it goes through *all* ports on the switch (whether they are active, meaning that there are packets destined for them, or idle), and for each it scans the entire list of packets in the batch to see if there's a packet for that port. If there is, it locks that port and delivers the packet; the lock is then released when the last packet in the batch is scanned, so that a lock is acquired and released only once when multiple packets are destined for a port.

This algorithm means that VALE incurs overheads that are linear with respect to the number of ports, irrespective of whether they are idle or active. Since it is a common scenario to have batches that do not contain packets for every port, VALE's algorithm is inefficient as port density increases. Worse, if the batch contains a single packet for a port, the algorithm still goes through the entire batch, even once that packet has been processed.

Instead, mSwitch takes a significant departure from VALE's algorithm and consists of two phases (see Figure 3). In the first phase, the algorithm creates a list of packets for each destination port. For instance, in the figure there are 3 destination ports labeled a through c. The scratchpad structure keeps track of the head and the tail packet buffer for each port: the head allows us, in the second phase, to reach the first packet for the port without having to traverse the entire batch; and the tail lets us, in the first phase, append a packet to the list, again without having to traverse it. When a new packet destined for port "a" arrives, the algorithm looks up the list's tail (e.g., 5 for port "a" in the figure), sets that packet buffer's "Next buf" pointer to the new packet, and the tail gets also set to the new packet.

In the second phase, mSwitch goes through each port in the scratchpad, that is, *only through ports on the switch which are active*, i.e., those that have packets in the batch destined to them. For each port, the algorithm looks up the head and retrieves the packets destined to it by following the linked list. mSwitch also supports broadcast by using a separate list and a special value to denote a broadcast destination.

Finally, a short word on using a lock to resolve contention on a destination port (e.g., a NIC receiving outgoing packets from multiple processes, containers or VMs). While we could rely on technologies such as hardware multi-queue to alleviate the costs related to the lock, we opt for a generic solution that does not depend on hardware support, and show that we can obtain good performance with it. More importantly, we want to allow virtual ports to be processed by any CPU core, which is important for flexible CPU core assignment of VMs (with hardware multi-queue, each queue has to be assigned its own CPU core, seriously restricting the flexibility of the scheduling).

Destination Port Parallelism: Once the list of packets destined to a given port has been identified, the packets must be copied to the destination port and made available on the ring. However, care must be taken in scenarios where a potentially large number of sender ports overwhelm a single destination port, somewhat akin to the incast problem in data centers.

In order to cope with this, mSwitch achieves destination port parallelism by implementing a two-phase mechanism: a sender reserves (under lock[6]) a sufficiently large contiguous range of slots in the output queue, then releases the lock during the copy, allowing concurrent operation on the queue, and finally acquires the lock again to advance queue pointers. This latter phase also handles out-of-order completions of the copy phases, which may occur for many reasons (different batch or packet sizes, cache misses, page faults). With this strategy, the queue is locked only for the short intervals needed to reserve slots and update queue pointers. As an additional side benefit, we can now tolerate page faults during the copy phase, which allows using userspace buffers as data sources. We provide evaluation results for this mechanism in Section 4.4.

3.4 Modular Switching Logic

mSwitch runs in the kernel, and each switch instance can define its own switch logic function by loading a suitable kernel module which implements it. The function is called once for each packet in a batch, before the actual forwarding takes place, and the response is used to add the packet to the unicast or broadcast lists discussed in Section 3.3. The function receives a pointer to the packet's buffer and its length, and should return the destination port (with special values indicating "broadcast" or "drop"). The function can perform arbitrary actions on the packet, including modifying its content or length, within the size of the buffer. It can possibly even block, and this will only harm the current sending port, not the whole switch. The mSwitch switching fabric will take care of performance optimizations (such as prefetching the payload if necessary), as well as validating the return values (e.g., making sure that packets are not bounced back to their source port).

By default, any newly created mSwitch instance uses a switch logic that implements a basic level-2 learning bridge. At any time during the bridge's lifetime, however, the logic may be replaced.

Writing a basic module is straightforward and involves registering a function with mSwitch. The following code shows a full implementation of a simple module which uses

[6]The spinlock could be replaced by a lock-free scheme, but we doubt this would provide any measurable performance gain.

the 8 least significant bits of the destination IP address to select an output port:

```
int my_lookup_fn(char *buf, int len,
                 struct ifnet *srcif) {
    struct ip *iph;

    iph = (struct ip *)(buf + ETHER_HDR_LEN);
    return ntohl(iph->ip_dst) & 0xff;
}

void my_module_register(char *switch_instance_name) {
        bdg_ctl(switch_instance_name, REG_LOOKUP,
                my_lookup_fn);
}
```

3.5 Other Extensions

The netmap API only supported fixed-size packet buffers (2 Kbytes by default) allocated by the kernel. Many virtualization solutions achieve huge performance improvements by transferring larger frames or even entire 64 Kbyte segments across the ports of the virtual switch [18]. As a consequence, we extended the netmap API to support scatter-gather I/O, allowing up to 64 segments per packet. This has an observable but relatively modest impact on the throughput, see Figure 6. More importantly, it permits significant speedups when connected to virtual machines, because the (emulated) network interfaces on the guest can spare the segmentation of TSO messages.

Additionally, forcing clients to use netmap-supplied buffers to send packets might cause an unnecessary data copy if the client assembles outgoing packets in its own buffers, as is the case for virtual machines. For this reason, we implemented another extension to the netmap API so that senders can now store output packets in their own buffers. Accessing those buffers (to copy the content to the destination port) within the forwarding loop may cause a page fault, but mSwitch can handle this safely because copies are done without holding a lock as described in Section 3.3.

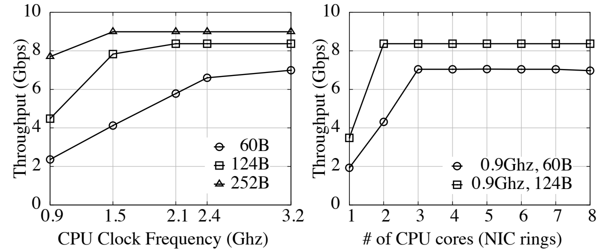
4. PERFORMANCE EVALUATION

In this section we evaluate the performance of mSwitch's switch fabric, including its scalability with increasing number of ports and NICs, its CPU utilization and its latency. One of the switch's biggest assets is its flexible data plane; in the next section we will give examples of this through the implementation and performance evaluation of four different modules.

In all experiments we run mSwitch on a server with an Intel Xeon E5-2695@2.8GHz 12-core CPU (3.2 GHz with Turbo Boost), 32GB of DDR3-1600 RAM (quad channel) and a dual-port, 10 Gbit/s Intel X520-T2 NIC. An additional machine is used for external traffic sources/sinks as needed. The operating system is FreeBSD 10 for most experiments, and Linux 3.9 for the Open vSwitch experiments in the next section.

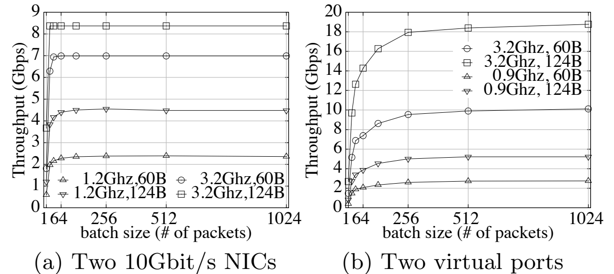
We normally run the CPU of the mSwitch machine at 3.2 GHz, but in some experiments we also vary the CPU frequency using OS-specific interfaces (sysctl on FreeBSD, sysfs on Linux). Further, we disable hyper-threading and enable direct cache access (DCA).

To generate and count packets we use `pkt-gen`, a fast generator that uses the netmap API and so can drive both NICs and mSwitch's virtual ports. Throughout, we use Gb/s to mean Gigabits per second and Mp/s for millions of packets per second. Unless otherwise specified, we use a batch



(a) CPU frequency, single core (b) Number of CPU cores

Figure 4: mSwitch throughput between two 10 Gbit/s NICs for different CPU frequencies



(a) Two 10Gbit/s NICs (b) Two virtual ports

Figure 5: Throughput between 10 Gbit/s NICs and virtual ports for different batch sizes.

size of 1,024 packets. Finally, packet sizes in the text and graphs do not include the 4-byte Ethernet checksum; this is for consistency with virtual ports experiments, where there is no CRC.

Note: the bandwidths reported in the figures do not include framing overheads (160 bits) and CRC. As a result, the maximum net rate on a 10 Gbit/s link varies from 7.14 Gb/s for 60-byte frames to 9.84 Gb/s for maximum sized (1514-byte) frames. The corresponding packet rates vary from 14.88 Mp/s to 0.81 Mp/s, respectively.

4.1 Basic Performance

To derive the baseline performance of the switch fabric, we use a simple switch logic that for each input port returns a statically configured output port; the processing cost (just an indirect function call) is thus practically negligible. We then evaluate mSwitch's throughput for different packet sizes and combinations of NICs and virtual ports.

Forwarding rates between NICs are bounded by CPU or NIC hardware, and mSwitch achieves higher forwarding rates with increasing CPU frequency (Figure 4(a)) and number of CPU cores (Figure 4(b)). We need relatively small batch sizes, such as 128 or 256 packets, to obtain line rate (Figure 5(a)), because the rates are limited by the NIC hardware. Note that the rates for 1 CPU core in Figure 4(b) are slightly lower than those in Figures 4(a) and 5(a) due to costs associated with our use of Flow Director [12], in charge of distributing packets to multiple CPU cores.

Packet forwarding between virtual ports exhibits similar characteristics, except that the rates are not bounded by the NIC hardware. Figure 6 shows throughput when assigning

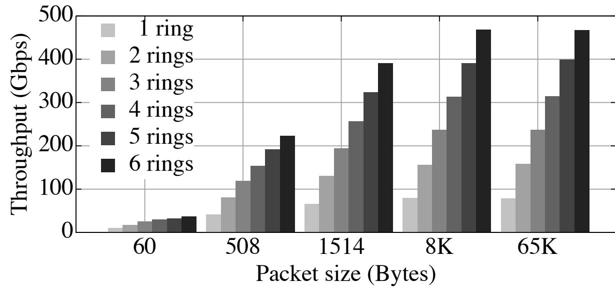


Figure 6: Forwarding performance between two virtual ports and different number of CPU cores/rings per port.

an increasing number of rings to each port (up to 6 per port, at which point all 12 CPU cores in our system are in use). Packet rates scale fairly linearly with increasing number of rings and CPU cores, achieving a maximum of about 75 million packet per second for minimum-sized packets, a rate of 390 Gb/s for maximum-sized ones and a maximum rate of 466 Gb/s for 8K and bigger frames. Note that the effect of using larger batch sizes is more pronounced than in the experiment with NICs since here the throughputs are higher (Figure 5(b)).

The reader may have noticed that this last rate exceeds the maximum memory bandwidth supported by our quad-channel, DDR3-1600 memory (12.8 GB/s per channel, for a total of 409Gb/s). We note that the 466 Gb/s is not an error, but rather the result of cache effects (recall that we run all experiments on a single CPU package server).

Finally, packet forwarding between a NIC and a virtual port (Figures 7(a) and 7(b)) is slightly cheaper than that between NICs, because passing packets to and from a virtual port does not need to convert buffer descriptors, issue memory-mapped I/O instructions to the hardware, nor handle interrupts.

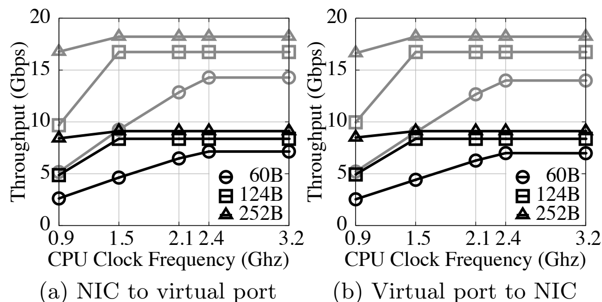


Figure 7: Throughput between a 10 Gbit/s NIC and a virtual port for different CPU frequencies. Gray plots show results with an extra pair of NIC and virtual port (20 Gbit/s total).

4.2 CPU Utilization

To test CPU utilization, we first compare mSwitch to DPDK vSwitch. We forward minimum-sized packets from a single NIC to an increasing number of receivers attached to virtual ports. To direct packets mSwitch uses a learning bridge module described in Section 5.1 and DPDK vSwitch inserts equivalent rules. DPDK vSwitch *requires* a separate CPU core per virtual port or user process, so we are limited to 9 ports (our system has 12 cores, of which 1 is used for the NIC, 1 for the `vswitchd` control daemon, 1 for the operating system and the remaining ones for the ports). As shown in Figure 8(a), DPDK vSwitch's poll-mode driver results in 100% utilization for all cores handling packets (10 in total: 9 for the virtual ports and one for the NIC). In contrast, mSwitch relies on interrupts so that user processes are woken up only on packet arrival, resulting in a maximum cumulative utilization of around 140%, versus 1000% for DPDK vSwitch for almost the same throughput. To quantify this even further, Figure 8(b) shows the power consumption for each of the two switches (removing the consumption used by the server when idle): for 9 ports, DPDK vSwitch consumes up to 80W, almost double what mSwitch uses.

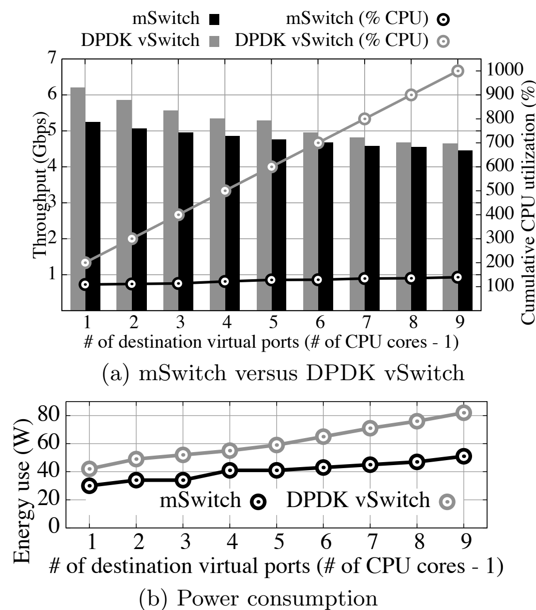


Figure 8: CPU utilization, throughput and power consumption. DPDK vSwitch edges out mSwitch in terms of throughput but at the cost of much higher CPU utilization and energy use.

Next, we compare mSwitch to VALE. For VALE we are not limited by the one-core-per-port requirement, and so we push the number of ports to 60 (out of its maximum of 64) for VALE and 120 for mSwitch (Figure 9) while still forwarding packets from a single NIC to all of the ports. One CPU core (NIC-CPU) is dedicated to packet switching and forwarding from the NIC. A user process runs on each virtual port and all the user processes share a single CPU core (App-CPU), for a total of two CPU cores in all for this experiment. Thanks to mSwitch's scalable forwarding algorithm, its throughput does not decrease as quickly as VALE's, and CPU consumption (NIC-CPU) is lower, especially for larger number of virtual ports (30–80). The reason why the CPU utilization for the user processes (App-CPU) on VALE is lower than mSwitch's for 1–50 virtual ports is

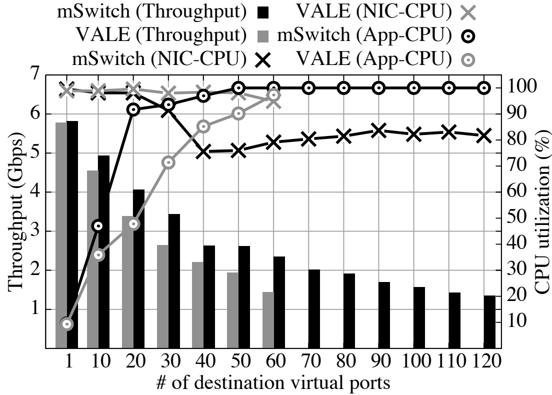


Figure 9: mSwitch outperforms VALE both in terms of scalability and throughput

that packet forwarding on the NIC-CPU is a bottleneck in VALE, resulting in the virtual ports CPU being more idle than in mSwitch's case.

4.3 Ports Scalability

Out of the available high performance software switches, VALE's the only one that can have a large number of virtual ports, and so in this experiment we perform a head-to-head comparison against it. First, we set up a single sender and 11 receivers, each of these assigned one of the system's 12 CPU cores. The sender sends packets to only one of the destination ports, so that the remaining ones are essentially "idle". Figures 10(a) and 10(b) show that VALE's algorithm incurs costs with increasing number of ports, even if they are not currently receiving packets; this would have negative effects when acting as a back-end switch to containers or VMs, since idle ones would affect the performance of active ones. mSwitch, instead, experiences minor or no throughput degradation.

Next, we conduct the same experiment but change the switch logic so that packets from the sender are delivered to all 11 destination ports in a round-robin fashion. Figures 10(c) and 10(d) show a similar general pattern as before. For short packets, mSwitch's throughput now degrades with increasing number of ports as a result of the algorithm having to handle a larger number of linked lists, one per active destination port.

To test the algorithm's scalability to much larger port numbers, we would have to have much larger number of CPU cores. We could, of course, simply add more ports and have them share whatever CPU cores are available, but this would mean that we would measure the system's ability to process packets from destination ports' rings, rather than mSwitch's algorithm itself. To get around this, we automatically drain the receivers' rings right after forwarding, as if packets were immediately consumed for "free". We do this from the sender's context, meaning that only the CPU core of the sender is doing actual work.

Further, we use a constant batch size of N=1,024 on the sender irrespective of the number of ports P, hence in each batch the number of packets per output port is N/P. Smaller batches mean that the per-output costs (locking, synchronization) are amortized on fewer packets. We thus

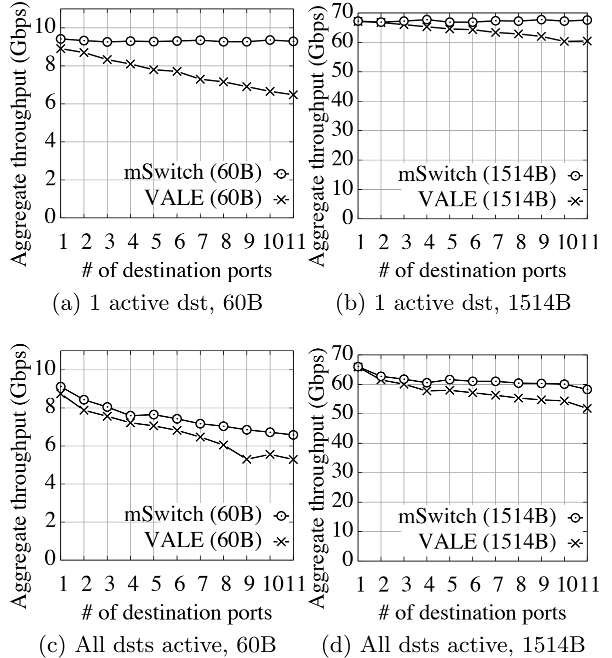


Figure 10: Packet forwarding from a single sender to multiple destination ports. The experiment uses virtual ports only (no NICs) and one CPU core for each of the 11 receiver and the single sender.

expect the aggregate throughput to decay with increasing number of output ports, and that of mSwitch to do so less quickly than VALE thanks to the new algorithm.

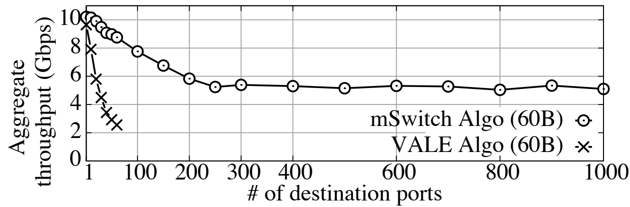
This is precisely what Figure 11 shows when comparing mSwitch's algorithm with VALE's in the presence of up to 1,000 destination ports. While both curves decay with increasing number of ports, mSwitch retains more than 50% of the throughput when going from 1 to 250 ports, whereas VALE drops to 25% with only 60 outputs.

As a final, more realistic, port density test, we configure mSwitch to have up to 120 virtual ports sharing a CPU core, and a NIC port with its own core (two total). Packets arriving at the NIC are sent to the 120 ports in a round-robin fashion and we measure throughput (Figure 9). As opposed to the previous test, which used a dummy module, this test uses an actual L2 learning bridge module (described in more detail in Section 5). As shown, while the throughput decreases with more ports, it does so linearly, and still achieves a respectable rate of 1.35 Gbit/s for 120 ports and minimum-sized packets on a *single* CPU core.

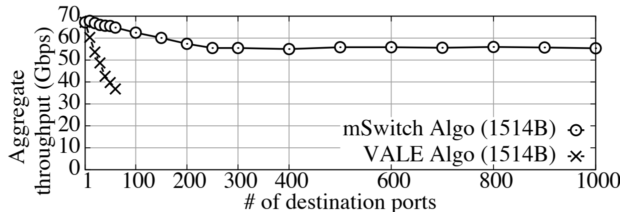
4.4 Destination Port Parallelization

We now evaluate the performance gains derived from using mSwitch's parallelization when handling the output rings (recall from Section 3.3 that mSwitch allows concurrent copies to the output buffers). This feature is useful when several sender ports converge on a single output one, a situation common in virtualized server environments.

To create contention on a single destination port, we use one receiver and up to eleven senders, for a total of 12, the number of CPU cores in our system. The results in Figure 12

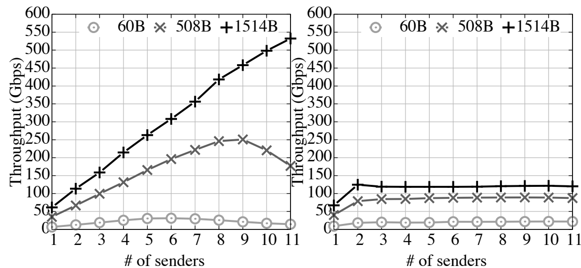


(a) 60B packets.



(b) 1514B packets.

Figure 11: Comparison of mSwitch's and VALE's forwarding algorithms in the presence of many ports. Batch size is set to 1,024.



(a) mSwitch, batch size 64(b) mSwitch, batch size 512.

Figure 12: mSwitch output ring management performance for different packet and batch sizes.

show mSwitch's througput with different packet and batch sizes, and exactly match our expectations. With small batch sizes the output ring is large enough to support concurrent operation for all senders. mSwitch exploits this opportunity and scales almost linearly (Figure 12(a) up to the point where memory bandwidth starts becoming a bottleneck, i.e., for large packets. With a larger batch (512 packets versus a ring size of 1,024) only a couple of senders at a time can effectively work in parallel, and mSwitch saturates with just 2 senders (Figure 12(b)). Overall, mSwitch achieves a maximum throughput of 532 Gb/s (1514-byte packets, batch of 64, 11 senders) and 65 Mp/s (60-byte packets, batch of 64 and 6 senders).

Note that in Figure 12(a) only the 508B curve seems to collapse (for $x > 9$). In fact, all three curves in the graphs do: the 60B curve does so for 7 senders (even if it is somewhat difficult to see from the graph) and the 1514B would do if we had more CPU cores to extend the X axis values. These drops are the result of higher contention on the single destination port, leading to livelock.

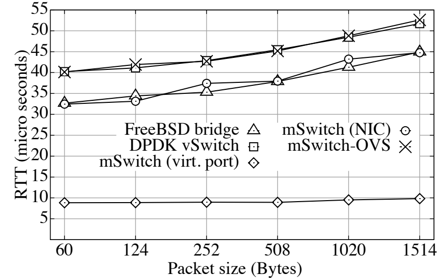


Figure 13: RTT of a request-response transaction (going through the switch twice).

4.5 Switch Latency

Virtualization technologies are infamous for inserting additional delay in paths, and so it is paramount that a backend switch, if at all possible, does not make matters worse. To show that this is not the case for mSwitch, for the final basic performance test we measure the latency of the switch. The experiment uses a request-response transaction (similar to `ping`) between a client and a server connected through mSwitch operating as a learning bridge. In this configuration, the round trip time (RTT) includes twice the latency of the switch, plus two thread wakeups (which consume at least $0.5 - 1$ μs each). When running between two mSwitch virtual ports, the latency is extremely low (see Figure 13).

For reference we also include results when using two physical 10 Gbit/s ports for various switches. For mSwitch and the native FreeBSD bridge, these two curves practically overlap, because the bulk of the delay in each RTT is the same in both cases, and it is due to interrupt processing (4 times), thread wakeups (4 times) traversals of buses (PCIe and link), and processing delays in the NIC. mSwitch's Open vSwitch module (mSwitch-OVS), discussed in Section 5.4, exhibits higher latency because of more complex packet processing (see Figure 16), as does DPDK vSwitch.

It is worth pointing out that the fact that there is almost no latency difference between mSwitch-OVS, which adopts an interrupt model and performs OpenFlow packet matching, and DPDK vSwitch, which uses a polling model with the same packet matching, demonstrates that the latency penalty arising from the interrupt model is negligible.

Finally, the difference in RTT between 60- and 1514-byte packets seems high but has a clear explanation. When traversing each link, the packet must be transferred, in sequence, through the PCIe bus (16 Gbit/s) on the transmit side, then the link (10 Gbit/s) and then the PCIe bus on the receive side (memory copies in mSwitch are negligible because of the much larger memory bandwidth). The difference in size between small and large packets is over 11 Kbits, and adding the transmission times for the 8 PCIe bus and 4 link traversals gives a figure in the order of $12 - 15$ μs, which matches the numbers in the graph rather well.

5. USE CASES

Having a clear understanding of the performance of mSwitch's switching fabric, we can focus on its flexible data plane and the sort of use cases it can enable. We further provide a performance evaluation to show how mSwitch is affected by dif-

ferent switching logics. As case studies we implemented four systems as modules: i) a basic layer-2 learning bridge (45 LoC); ii) a simple 3-tuple wildcard filter used for directing packets to virtualized middleboxes (50 LoC); iii) a 3-tuple exact-match filter used for user-level network stacks [8] (150 LoC); and iv) an mSwitch instance using the Open vSwitch datapath code as switching logic to drive the mSwitch fabric (only about 500 modified lines of code out of 10,000 total).

5.1 Layer 2 Learning Bridge

This system relies on a hash table to store forwarding entries (`<src-mac,src-port>` pairs, learned from incoming traffic). On each packet, the switching logic hashes the source MAC address to possibly insert a `<src-mac,src-port>` entry in the table, and then hashes the destination MAC to determine the destination port, returning a "broadcast" response if not found. We use SpookyHash [4] as the hash function.

The implementation (excluding the hash function) consists of only 45 lines of code. This module is used in measurement of throughput in Section 2, and measurement of throughput and CPU utilization in Section 4.2 and 4.3. The complexity of this module is also discussed in Section 5.5.

5.2 Virtualized Middlebox Filter

It is becoming increasingly common to run instances of middleboxes as virtualized, software-based functions running on commodity hardware, a model championed by the NFV trend. In many cases, such middleboxes are only interested in a subset of traffic, and so it would be wasteful to send packets from the NIC through the back-end software switch and onto the container or VM (i.e., the middlebox) only to be discarded. Instead, we can let mSwitch perform this filtering early and thus reduce load on the system. To this end, we program a simple module that carries out wildcard matching on source or destination ports, which is a reasonable filter type for a number of different middleboxes [9].

More specifically, we keep two simple arrays, one for UDP traffic and another one for TCP, containing mappings of protocol ports to switch ports. When a packet arrives, we attempt to match its source *or* its destination UDP/TCP port (to capture two-way traffic); if there is no match the packet is discarded.

To measure the performance of this module, we populate the mappings such that no packets are dropped, and packets are either all sent to a single destination port (Figure 14(a)) or sent to all available destination ports in a round-robin fashion (Figure 14(b)). For the former, mSwitch achieves 89% of line rate in the presence of 16 destination ports, compared to about 55% for a version that adopts VALE's forwarding algorithm. For the latter, mSwitch achieves line rate for all packet sizes for up to 8 ports (the rates are higher than in the previous graph because the additional active destinations mean that there are more CPU cores processing packets). Note that in Figure 14(b) the sudden throughput degradation at 16 ports is due to the fact that half of the receivers have to share CPU cores (again, our system has 12 cores in total).

5.3 Support for Userspace Protocol Stacks

User-space transport protocol implementations can greatly ease testing and quick deployment of new features [8]. However, when the protocol shares the namespace (protocol number, port numbers) with existing in-kernel implementations, the OS must make sure that each instance receives its own traffic and does not interfere with the other.

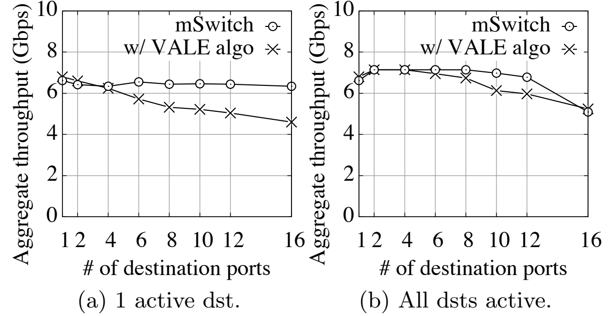


Figure 14: Performance of the virtual middlebox demultiplexer module. Minimum-sized packets arrive at a single NIC and the module distributes them to the corresponding destination ports. CPU cores are distributed among destination ports in a round-robin fashion.

These requirements can be met with an mSwitch module which we call MultiStack. In greater detail, we implement a filter able to direct packets between mSwitch ports based on the `<protocol, local IP, local port>` 3-tuple. User-level processes (which include a protocol's implementation) can connect to an mSwitch virtual port, and request the use of a given 3-tuple through an `ioctl()`. If the request does not collide with tuples already in use, the 3-tuple is reserved by the OS (preventing `bind()`s on it) and inserted in a hash table along with the mSwitch port number. Matching incoming packets from a NIC are then sent to the relevant virtual port, whereas others are sent to the host's network stack. The hash table is also used for output verification (using the source IP/port) to make sure that clients only generate traffic for addresses they "own".

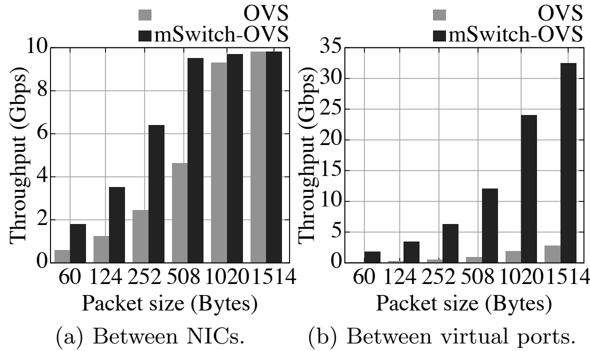
We tested the 3-tuple mSwitch performance by forwarding traffic between two 10 Gbit/s ports, reaching about 10 Mp/s (68% of line rate, see Figure 16) for 60-byte packets, and full line rate for larger packet sizes These values are only slightly lower than the mSwitch learning bridge, and largely above the speed of a host stack.

Taking this approach one step further, we also built a simple user-level TCP stack library along with an HTTP server built on top of it. Using these, and relying on mSwitch's 3-tuple filter module for back-end support, we were able to achieve HTTP rates of up to 8 Gbps for 16 KB and higher fetch sizes, and about 100 K requests per second for short requests; for more details on this experiment see [8].

5.4 Open vSwitch Datapath

In our final use case, we study how difficult and effective it is to integrate existing software with the mSwitch switch fabric. In particular, we used Open vSwitch 2.3's code to implement the switch logic of what we term mSwitch-OVS. We modified only Open vSwitch's datapath part, which is implemented as a Linux kernel module and consists of about 10,000 LoC. This required the implementation of some glue code (approximately 100 LoC) to create an mSwitch instance on startup, and an additional 400 LoC (including

70 modified lines) to hook the Open vSwitch code to the mSwitch switching logic. In essence, mSwitch-OVS replaces Open vSwitch's datapath, which normally uses Linux's standard packet I/O, with mSwitch's fast packet I/O. As a result, we can avoid expensive, per-packet `sk_buff` allocations and deallocations.



(a) Between NICs. (b) Between virtual ports.

Figure 15: Throughput of mSwitch's Open vSwitch module (mSwitch-OVS) as opposed to that of standard Open vSwitch (OVS) on a single CPU core. Measurements are done when forwarding between NICs (left) and between virtual ports (right). In the latter, for standard Open vSwitch we use `tap` devices.

The results when forwarding between NICs using a single CPU core (Figure 15(a)) show that with relatively few small changes to Open vSwitch, mSwitch is able to achieve important throughput improvements: For small packets, we notice a 2.6-3x speed-up. The difference is also large when forwarding between virtual ports (Figure 15(b)), although part of those gains are certainly due to the presence of slow tap devices for Open vSwitch.

5.5 Module Complexity

As the final evaluation experiment, we look into how expensive the various modules are with respect to CPU frequency. Figure 16 summarizes how the throughput of mSwitch is affected by the complexity of the switching logic for minimum-sized packets and different CPU frequencies. As shown, hash-based functions (learning bridge or 3-tuple filter) are relatively inexpensive and do not significantly impact the throughput of the system. The middlebox filter is even cheaper, since it does not incur the cost of doing a hash look-up.

Conversely, Open vSwitch processing is much more CPU intensive, because OpenFlow performs packet matching against several header fields across different layers; the result is reflected in a much lower forwarding rate, and also an almost linear curve even at the highest clock frequencies.

6. GENERALITY AND LESSONS LEARNED

Through the process of designing, implementing and experimenting with mSwitch we have learned a number of lessons, as well as developed techniques that we believe are general and thus applicable to other software switch packages:

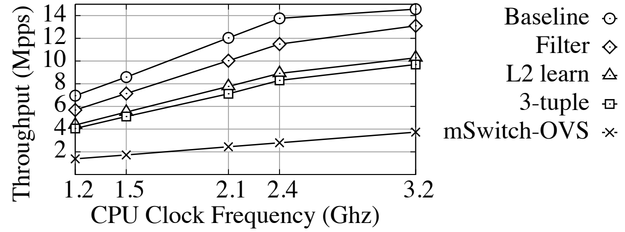


Figure 16: Throughput comparison between different mSwitch modules for 60 Byte packets.

- **Interrupt vs. Polling Model**: Using a polling model can yield some throughput gains with respect to an interrupt-based one, but at the cost of much higher CPU utilization. For a dedicated platform (e.g., CuckooSwitch, which uses the server it runs on solely as a hardware-switch replacement) this may not matter so much, but for a system seeking to run a software switch as a back-end to processes, containers, or virtual machines, an interrupt-based model (or a hybrid one such as NAPI) is more efficient and spares cycles that those processes can use. Either way, high CPU utilization equates to higher energy consumption which is always undesirable. We also showed that latency penalty arising from the use of an interrupt model is negligible for OpenFlow packet matching (Section 4.5).
- **Data Plane Decoupling**: Logically separating mSwitch into a fast, optimized switching fabric and a specialized, modular switching logic achieves the best from both worlds: the specialization required to reach high performance with the flexibility and ease of development typically found in general packet processing systems (Section 3.4).
- **High Port Density**: The algorithm presented in Section 3.3 permits the implementation of a software switch with both high port density *and* high performance. The algorithm is not particular to mSwitch, and so it can be applied to other software packages.
- **Destination Port Parallelism**: Given the prevalence of virtualization technologies, it is increasingly common to have multiple sources in a software switch (e.g., the containers or VMs) needing to concurrently access a shared destination port (e.g., a NIC). The algorithm described in Section 3.3 yields high performance under these scenarios and is generic, so applicable to other systems.
- **Huge Packets**: Figure 6 suggests that huge packets provide significant advantages for the switching plane in a high performance switch. Supporting these packets is important when interacting with entities (e.g., virtual machines) which have massive per packet overheads (e.g., see [18]).
- **Zero-Copy Client Buffers**: Applications or virtual machines connected to ports on a software switch are likely to assemble packets in their own buffers, different from those of the underlying switch. To prevent costly transformations and memory copies, the switch should allow such clients to store output packets in their own buffers (Section 3.5).
- **Application Model**: How should functionality be

split between an mSwitch instance and the user-level application or container/VM connecting to a switch port? The experience in writing the modules in this paper suggests that the best approach is to place computationally expensive processing in the application, leaving only basic functionality to the switch (e.g., running some filtering and demuxing packets to the right switch ports, as in Section 5.2 and 5.3, respectively). Following this model ensures that the system does not unnecessarily drop "cheap" packets while handling expensive ones. This is akin to OpenFlow's model: fast, simple rule matching in hardware with more advanced processing carried out by software.

7. CONCLUSIONS

We have presented mSwitch, a flexible software switch targeted at next generation software-defined networks and providing high throughput, low CPU utilization, high port density and a programmable data plane. Through an extensive performance evaluation we have shown that it can switch packets at rates of hundreds of gigabits per second on inexpensive, commodity hardware. We have further presented four different modules to give a flavor of the use cases that could be implemented using the switch.

As future work, we will implement the ability for mSwitch's modules to receive entire batches of packets instead of having per-packet semantics. This should further increase performance, as well as allow the module to split and coalesce packets, useful, for example, for implementing re-segmenting middleboxes [9].

mSwitch has been integrated in the netmap/VALE code [1], is part of standard FreeBSD distributions, and available in source format for a wide range of Linux kernels. All the modules introduced in Section 5 are also publicly available at [15]

8. ACKNOWLEDGEMENTS

Michio Honda and Felipe Huici have received funding from the European Union's Horizon 2020 research and innovation program 2014-2018 under grant agreement No. 644866 (SSICLOPS). The authors would like to thank Costin Raiciu for help with steering the paper.

9. REFERENCES

[1] The netmap project. `http://info.iet.unipi.it/~luigi/netmap/`.

[2] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing datacenter packet transport. In *Proc. ACM HotNets*, pages 133–138, 2012.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proc. USENIX OSDI*, pages 49–65, Oct. 2014.

[4] Bob Jenkins' Web Site. SpookyHash: a 128-bit noncryptographic hash. `http://www.burtleburtle.net/bob/hash/spooky.html`, October 2013.

[5] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proc. ACM SOSP*, pages 15–28, 2009.

[6] ETSI Portal. Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action. `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`, October 2012.

[7] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proc. ACM SIGCOMM*, September 2010.

[8] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. Rekindling network protocol innovation with user-level stacks. *ACM CCR*, 44(2):52–58, Apr. 2014.

[9] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it Still Possible to Extend TCP? In *Proc. ACM IMC*, pages 181–192, 2011.

[10] J. Hwang, K. Ramakrishnan, and T. Wood. Netvm: high performance and flexible networking using virtualization on commodity platforms. In *Proc. USENIX NSDI*.

[11] Intel. Intel DPDK: Data Plane Development Kit. `http://dpdk.org`, September 2013.

[12] Intel. Intel Ethernet Flow Director. `http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html`, May 2015.

[13] Intel Corporation. Packet Processing - Intel DPDK vSwitch - OVS. `https://github.com/01org/dpdk-ovs`, January 2014.

[14] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proc. USENIX NSDI*, pages 459–473, Apr. 2014.

[15] NEC Laboratories Europe. Cloud Networking Performance Lab. `http://cnp.neclab.eu/`.

[16] OPEN VSWITCH. Open vSwitch. `http://openvswitch.org`, 2013.

[17] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *Proc. USENIX OSDI*, pages 1–16, Oct. 2014.

[18] K. K. Ram, A. L. Cox, and S. Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proc. USENIX ATC*, 2013.

[19] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proc. USENIX ATC*, 2012.

[20] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Proc. IEEE INFOCOM*, pages 2471–2479, 2012.

[21] L. Rizzo and G. Lettieri. Vale: a switched ethernet for virtual machines. In *Proc. ACM CoNEXT*, December 2012.

[22] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proc. USENIX Linux Showcase and Conference*, 2001.

[23] SDNCentral.com. Time for an SDN Sequel? Scott Shenker Preaches SDN Version 2. `https://www.sdxcentral.com/articles/news/`

scott-shenker-preaches-revised-sdn-sdnv2/2014/10/, October 2014.
[24] V. Sekar, N. Egi, S. Ratnasamy, M. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. USENIX NSDI*, 2012.
[25] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *Proc. ACM SIGCOMM*, pages 13–24, 2012.
[26] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proc. ACM CoNEXT*, December 2013.