ClickOS and the Art of Network Function Virtualization

Joao Martins[†], Mohamed Ahmed[†], Costin Raiciu[‡], Vladimir Olteanu[‡], Michio Honda[†], Roberto Bifulco[†], Felipe Huici[†]

[†] NEC Europe Ltd. [‡] University Politehnica of Bucharest

Abstract

Over the years middleboxes have become a fundamental part of today's networks. Despite their usefulness, they come with a number of problems, many of which arise from the fact that they are hardware-based: they are costly, difficult to manage, and their functionality is hard or impossible to change, to name a few.

To address these issues, there is a recent trend towards network function virtualization (NFV), in essence proposing to turn these middleboxes into software-based, virtualized entities. Towards this goal we introduce ClickOS, a high-performance, virtualized software middlebox platform. ClickOS virtual machines are small (5MB), boot quickly (about 30 milliseconds), add little delay (45 microseconds) and over one hundred of them can be concurrently run while saturating a 10Gb pipe on a commodity server. We further implement a wide range of middleboxes including a firewall, a carrier-grade NAT and a load balancer and show that ClickOS can handle packets in the millions per second.

1 Introduction

The presence of hardware-based network appliances (also known as middleboxes) has exploded, to the point where they are now an intrinsic and fundamental part of today's operational networks. They are essential to network operators, supporting a diverse set of functions ranging from security (firewalls, IDSes, traffic scrubbers), traffic shaping (rate limiters, load balancers), dealing with address space exhaustion (NATs) or improving performance (traffic accelerators, caches, proxies), to name a few. Middleboxes are ubiquitous: a third of access networks show symptoms of stateful middlebox processing [12] and in enterprise networks there are as many middleboxes deployed as routers and switches [37].

Despite their usefulness, recent reports and operator feedback reveal that such proprietary middleboxes come with a number of significant drawbacks [9]: middleboxes are expensive to buy and manage [37], and introducing new features means having to deploy new hardware at the next purchase cycle, a process which on average takes four years. Hardware middleboxes cannot easily be scaled up and down with shifting demand, and so must provisioned to cope with peak demand, which is wasteful. Finally, a considerable level of investment is needed to develop new hardware-based devices, which leaves potential small players out of the market and so raises innovation barriers.

To address these issues, Network Function Virtualization (NFV) has been recently proposed to shift middlebox processing from hardware appliances to software running on inexpensive, commodity hardware (e.g., x86 servers with 10Gb NICs). NFV has already gained a considerable momentum: seven of the world's leading telecoms network operators, along with 52 other operators, IT and equipment vendors and technology providers, have initiated a new standards group for the virtualization of network functions [8].

NFV platforms must support multi-tenancy, since they are intended to concurrently run software belonging to the operator and (potentially untrusted) third parties: colocated middleboxes should be isolated not only from a security but also a performance point of view [10]. Further, as middleboxes implement a large range of functionality, platforms should accommodate a wide range of OSes, APIs and software packages.

Is it possible to build a software-based virtualized middlebox platform that fits these requirements? Hypervisor-based technologies such as Xen or KVM are well established candidates and offer security and performance isolation out-of-the-box. However, they only support small numbers of tenants and their networking performance is unsatisfactory[1]. At a high-level, the reason for the poor performance is simple: neither the hypervisors (Xen or KVM), nor the guest OSes (e.g., Linux) have been optimized for middlebox processing.

In this paper we present the design, implementation and evaluation of ClickOS, a Xen-based software platform optimized for middlebox processing. To achieve high performance, ClickOS implements an extensive overhaul of Xen's I/O subsystem, including changes to the back-end switch, virtual net devices and back and front-end drivers. These changes enable ClickOS to significantly speed up networking in middleboxes running in Linux virtual machines: for simple packet generation, Linux throughput increases from 6.46 Gb/s to 9.68 Gb/s for 1500B packets and from 0.42 Gb/s to 5.73 Gb/s for minimum-sized packets.

This work was partly funded by the EU FP7 CHANGE (257422) project.

[1] In our tests, a Xen guest domain running Linux can only reach rates of 6.5 Gb/s on a 10Gb card for 1500-byte packets out-of-the-box; KVM reaches 7.5 Gb/s.

A key observation is that developing middleboxes as applications running over Linux (and other commodity OSes) is a complex task and uses few of the OS services beyond network connectivity. To allow ease of development, a much better choice is to use specialized frameworks to program middleboxes. Click [17] is a stand-out example as it allows users to build complex middlebox processing configurations by using simple, well known processing elements. Click is great for middlebox processing, but it currently needs Linux to function and so it inherits the overheads of commodity OSes.

To support fast, easily programmable middleboxes, ClickOS implements a minimalistic guest virtual machine that is optimized from the ground up to run Click processing at rates of millions of packets per second. ClickOS images are small (5MB), making it possible to run a large number of them (up to 400 in our tests). ClickOS virtual machines can boot and instantiate middlebox processing in under 30 milliseconds, and can saturate a 10Gb/s link for almost all packets sizes while concurrently running as many as 100 ClickOS virtual machines on a single CPU core.

2 Problem Statement

Our goal is to build a versatile, high performance software middlebox platform on commodity hardware. Such a platform must satisfy a number of performance and security requirements:

Flexibility to run different types of software middleboxes, relying on different operating systems or frameworks, coming from different vendors, and requested by the operator itself or potentially untrusted third-parties.

Isolation of memory, CPU, device access and performance to support multiple tenants on common hardware.

High Throughput and Low Delay: Middleboxes are typically deployed in operator environments so that it is common for them to have to handle large traffic rates (e.g., multiple 10Gb/s ports); the platform should be able to handle such rates, while adding only negligible delay to end-to-end RTTs.

Scalability: Running middleboxes for third-parties must be very efficient if it is to catch on. Ideally, the platform should ideally support a large number of middleboxes belonging to different third-parties, as long as only a small subset of them are seeing traffic at the same time. This implies that platforms must be able to quickly scale out processing with demand to make better use of additional resources on a server or additional servers, and to quickly scale down when demand diminishes.

How should middleboxes be programmed? The default today is to code them as applications or kernel changes on top of commodity OSes. This allows much **flexibility** in choosing the development tools and languages, at the cost of having to run one commodity OS to support a middlebox.

In addition, a large fraction of functionality is common across different middleboxes, making it important to support **code re-use** to reduce prototyping effort, and processing re-use to reduce overhead [36].

3 Related Work

There is plenty of related work we could leverage to build NFV platforms. Given that the goal is to isolate different middleboxes running on the same hardware, the choice is either containers (`chroot`, FreeBSD Jails, Solaris Zones, OpenVZ [44, 45, 27]) or hypervisors (VMWare Server, Hyper-V, KVM, Xen [40, 21, 16, 3]).

Containers are lightweight but inflexible, forcing all middleboxes to run on the same operating system. This is a limitation even in the context of an operator wanting to run software middleboxes from different vendors.

Hypervisors provide the flexibility needed for multi-tenant middleboxes (i.e., different guest operating systems are able to run on the same platform), but this is at the cost of high performance, especially in networking. For high-performance networking with hypervisors, the typical approach today is to utilize device pass-through, whereby virtual machines are given direct access to a device (NIC). Pass-through has a few downsides: it complicates live migration, and it reduces scalability since the device is monopolized by a given virtual machine. The latter issue is mitigated by modern NICs supporting technologies such as hardware multi-queuing, VMDq and SR-IOV [14], however the number of VMs is still limited by the number of queues offered by the device. In this work we will show that it is possible to maintain performance scalability even without device pass-through.

Minimalistic OSes and VMs: Minimalistic OSes or micro kernels are attractive because, unlike traditional OSes, they aim provide just the required functionality for the job. While many minimalist OSes have been built [22, 23, 1, 42, 43], they typically lack driver support for a wide range of devices (especially NICs), and most do not run in virtualized environments. With respect to ClickOS, Mirage [19] is also a Xen VM built on top of MiniOS, but the focus is to create Ocaml, type-safe virtualized applications and, as such, its network performance is not fully optimized (e.g., 1.7 Gb/s for TCP traffic). Erlang on Xen, LuaJIT and HalVM also leverage MiniOS to provide Erlang, Lua, and Haskell programming environments; none target middlebox processing nor are optimized for network I/O.

Network I/O Optimization: Routebricks [7] looked into creating fast software routers by scaling out to a number of servers. PacketShader [11] took advantage of low cost GPUs to speed up certain types of network

processing. More recently, PFQ, PF_RING, Intel DPDK and netmap [25, 6, 13, 29] focused on accelerating networking by directly mapping NIC buffers into user-space memory; in this work we leverage the last of these to provide a more direct pipe between NIC and VMs.

Regarding virtualization, work in the literature has looked at improving the performance of Xen networking [28, 35], and we make use of some of the techniques suggested, such as grant re-use. The works in [47, 24] look into modifying scheduling in the hypervisor in order to improve I/O performance; however, the results reported are considerably lower than ClickOS. Finally, Hyper-Switch [15] proposes placing the software switch used to mux/demux packets between NICs and VMs inside the hypervisor. Unfortunately, the switch's data plane relies on open vSwitch code [26], resulting in sub-optimal performance. More recently, two separate efforts have looked into optimizing network I/O for KVM [4] [32]; neither of these has focused on virtualizing middlebox processing, and the rates reported are lower than those in this paper.

Software Middleboxes: Comb [36] introduces an architecture for middlebox deployments targeted at consolidation. However, it does not support multi-tenancy nor isolation, and the performance figures reported (about 4.5Gb/s for two CPU cores assuming maximum-sized packets) are lower than the line-rate results we present in Section 9. The work in [37] uses Vyatta software (see below) to run software middleboxes on Amazon EC2 instances. Finally, while a number of commercial offerings exist (Cisco [5], Vyatta [41]), there are no publicly-available detailed evaluations.

It is worth noting that a preliminary version of this paper has appeared as [20]. This version includes a detailed account of our solution and design decisions, extensive benchmarking as well as implementation and evaluation of a range of ClickOS middleboxes.

4 ClickOS Design

To achieve flexibility, isolation and multi-tenancy, we rely on hypervisor virtualization, which adds an extra software layer between the hardware and the middlebox software which could hurt throughput or increase delay. To minimize these effects, para-virtualization is preferable to full virtualization: para-virtualization makes minor changes to the guest OSes, greatly reducing the overheads inherent in full virtualization such as VM exits [2] or the need for instruction emulation [3].

Consequently, we base ClickOS on Xen [3] since its support for para-virtualized VMs provides the possibility to build a low-delay, high-throughput platform, though its potential is not fulfilled out of the box (Section 6).

Middlebox	Key Click Elements
Load balancer	`RatedSplitter, HashSwitch`
Firewall	`IPFilter`
NAT	`[IP|UDP|TCP]Rewriter`
DPI	`Classifier, IPClassifier`
Traffic shaper	`BandwidthShaper, DelayShaper`
Tunnel	`IPEncap, IPsecESPEncap`
Multicast	`IPMulticastEtherEncap, IGMP`
BRAS	`PPPControlProtocol, GREEncap`
Monitoring	`IPRateMonitor, TCPCollector`
DDoS prevention	`IPFilter`
IDS	`Classifier, IPClassifier`
IPS	`IPClassifier, IPFilter`
Congestion control	`RED, SetECN`
IPv6/IPv4 proxy	`ProtocolTranslator46`

Table 1: Key Click elements that allow developing a wide range of middleboxes.

KVM also supports driver para-virtualization through `virtio` [33], but yields lower performance (Section 6).

Programming Abstractions. Today's software middleboxes are written either as user-space applications on top of commodity OSes (e.g., Snort or Bro) or as kernel changes (e.g., iptables, etc). Either way, C is the de-facto programming language as it offers high performance.

Our platform aims to allow today's middleboxes to run efficiently in the context of virtualization. However, we believe that there are much better ways to develop fast middleboxes. C offers great flexibility but has high development and debugging costs, especially in the kernel. In addition, there is not much software one can reuse when programming a new type of middlebox.

Finding the best programming abstraction for middleboxes is an interesting research topic, but we do not set out to tackle it in this paper. Instead, we want to pragmatically choose the best tool out of the ones we have available today. As a result, we leverage the Click modular router software. Previous work [36] showed that a significant amount of functionality is common across a wide range of middleboxes; Click makes it easy to reuse such functionality, abstracting it into a set of re-usable elements. Click comes with over 300+ stock elements which make it possible to construct middleboxes with minimal effort (Table 1). Finally, Click is extensible, so we are not limited to the functionality provided by the stock elements. Click is of course no panacea: it does not cover all types of middlebox processing, for instance middleboxes that need a full-fledged TCP stack. In such cases it is better to use a standard Linux VM.

Running Click Efficiently: By default, Click runs on top of Linux either as a userland process (with poor performance, see [30]) or as a kernel module. To get domain isolation, we would have to run each Click middlebox inside a Linux virtual machine. This, however, violates our scalability requirement: even stripped down Linux VMs are memory-hungry (128MB or more) and take 5s to boot.

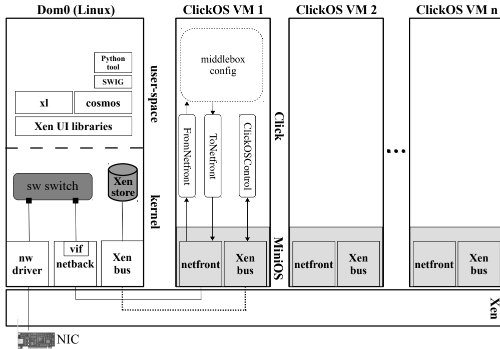


Figure 1: ClickOS architecture.

Instead, we take a step back and ask: *what support does Click need from the operating system to be able to enable a wide range of middlebox processing?* The answer is, surprisingly, not much:

- Driver support to be able to handle different types of network interfaces.
- Basic memory management to allocate different data structures, packets, etc.
- A simple scheduler that can switch between running Click element code and servicing interrupts (mostly from the NICs). Even a cooperative scheduler is enough - there is no need for pre-emptive scheduling, or multi-threading.

The first requirement seems problematic, given the large number of interface vendors and variety of models. Xen elegantly solves this issue through paravirtualization: the guest accesses all NIC types through a single, hardware-agnostic driver connected to the driver domain, and the driver domain (a full-blown Linux machine with the customary driver support) talks to the hardware itself.

Almost all operating systems meet the other two requirements, so there is no need to build one from scratch: we just need an OS that is minimalistic and is able to boot quickly. Xen comes with MiniOS, a tiny operating system that fits the bill and allows us to build efficient, virtualized middleboxes without all of the unnecessary functionality included in a conventional operating system. MiniOS is the basis for our ClickOS VMs.

In short, our ClickOS virtualized middlebox platform consists of (1) a number of optimizations to Xen's network I/O sub-system that allow fast networking for traditional VMs (Section 7); (2) tailor-made middlebox virtual machines based on Click; and (3) tools to build and manage the ClickOS VMs, including inserting, deleting, and inspecting middlebox state (Figure 1).

5 ClickOS Virtual Machines

Before describing what a ClickOS virtual machine is, it is useful to give a brief Xen background. Xen is split into a privileged virtual machine or *domain* called dom0 (typically running Linux), and a set of guest or user domains comprising the users' virtual machines (also known as domUs). In addition, Xen includes the notion of a *driver domain* VM which hosts the device drivers, though in most cases dom0 acts as the driver domain. Further, Xen has a split-driver model, where the back half of a driver runs in a driver domain, the front-end in the guest VM, and communications between the two happen using shared memory and a common, ring-based API. Xen networking follows this model, with dom0 containing a `netback` driver and the guest VM implementing a `netfront` one. Finally, *event channels* are essentially Xen inter-VM interrupts, and are used to notify VMs about the availability of packets.

MiniOS implements all of the basic functionality needed to run as a Xen VM. MiniOS has a single address space, so no kernel/user space separation, and a cooperative scheduler, reducing context switch costs. MiniOS does not have SMP support, though this could be added. We have not done so because a single core is sufficient to support 10 Gbps line-rate real middlebox processing, as we show later. Additionally, we scale up by running many tiny ClickOS VMs rather than a few large VMs using several CPU cores each.

Each ClickOS VM consists of the Click modular router software running on top of MiniOS, but building such a VM image is not trivial. MiniOS is intended to be built with standard GCC and as such we can in principle link any standard C library to it. However, Click is written in `c++`, and so it requires special precautions. The most important of these is that standard `g++` depends on (among others) `ctypes.h` (via `glibc`) which contains Linux specific dependencies that break the standard MiniOS `iostream` libraries. To resolve this we developed a new build tool which creates a Linux-independent `c++` cross-compiler using `newlibc` [38].

In addition, our build tool re-designs the standard MiniOS toolchain so that it is possible to quickly and easily build arbitrary, MiniOS-based VMs by simply linking an application's entry point so that it starts on VM boot; this is useful for supporting middleboxes that cannot be easily supported by Click. Regarding libraries, we have been conservative in the number of them we link, and have been driven by need rather than experimentation. In addition to the standard libraries provided with the out-of-the-box MiniOS build (`lwip, zlib, libpci`) we add support for `libpcre, libpcap and libssl`, libraries that certain Click elements depend on. The result is a ClickOS image with 216/282 Click elements, with many of the remaining ones requiring a filesystem to run, which we plan to add.

Once built, booting a ClickOS image start by creating the virtual machine itself, which involves reading its configuration, the image file, and writing a set of entries

to the Xen store, a `proc`-like database residing in dom0 that is used to share control information with the guest domains. Next, we attach the VM to the back-end switch, connecting it to physical NICs.

MiniOS boots, after which a special control thread is created. At this point, the control thread creates an *install* entry in the Xen store to allow users to install Click configurations in the ClickOS VM. Since Click is designed to run on conventional OSes such as Linux or FreeBSD which, among other things, provide a console through which configurations can be controlled and, given that MiniOS does not provide these facilities, we leverage the Xen store to emulate such functionality.

Once the install entry is created, the control thread sets up a watch on it that monitors changes to it. When written to, the thread launches a second MiniOS thread which runs a Click instance, allowing several Click configurations to run within a single ClickOS VM. To remove the config we write an empty string to the Xen store entry.

We also need to support Click *element handlers*, which are used to set and retrieve state in elements (e.g, the `AverageCounter` element has a read counter to get the current packet count and a write one to reset the count); to do so, we once again leverage the Xen store. For each VM, we create additional entries for each of the elements in a configuration and their handlers. We further develop a new Click element called `ClickOSControl` which gets transparently inserted into all configurations. This element takes care of interacting, on one end, with the read and write operations happening on the Xen store, and communicating those to the corresponding element handlers within Click.

In order to control these mechanisms which are not standard to all Xen VMs, ClickOS comes with its own dom0 CLI called Cosmos (as opposed to the standard, Xen-provided `xl` tool). Cosmos is built directly on top of the Xen UI libraries (Figure 1) and therefore does not incur any extraneous costs when processing requests. To simplify development and user interaction, Cosmos implements a SWIG [39] wrapper enabling users to automatically generate Cosmos bindings for any of the SWIG supported languages. For convenience, we have also implemented a Python-based ClickOS CLI.

Finally, it is worth mentioning that while MiniOS represents a low-level development environment, programming for ClickOS is relatively painless: development, building and testing can take place in user-space Click, and the resulting code/elements simply imported into the ClickOS build process when ready.

6 Xen Networking Analysis

In this section we investigate where the Xen networking bottlenecks are. Figure 1 illustrates the Xen network I/O sub-system: the network driver, software switch, virtual interface and netback driver in dom0 and the netfront driver (either the Linux or MiniOS one) in the guest domains, any of which could be bottlenecks.

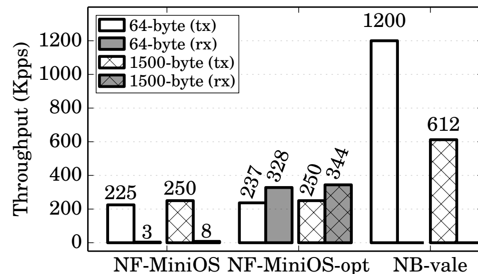


Figure 2: Xen performance bottlenecks using a different back-end switch and netfront (NF) and netback (NB) drivers ("opt" stands for optimized).

In order to get some baseline numbers, we begin by performing a simple throughput test. For this test we used a server with an Intel Xeon E3-1220 3.1GHz 4-core CPU, 16GB memory and an Intel x520-T2 dual Ethernet port 10Gb/s card (about $1,500 including the NIC). The server had Xen 4.2, Open vSwitch as its back-end switch and a single ClickOS virtual machine. The VM was assigned a single CPU core, the remainder given to dom0.

The first result (labeled "NF-MiniOS" in Figure 2) shows the performance of the MiniOS netfront driver when sending (Tx, in which case we measure rates at the netback driver in dom0) and receiving (Rx) packets. Out of the box, the MiniOS netfront driver yields poor rates, especially for Rx, where it can barely handle 8 Kp/s.

To improve this receive rate, we modified the netfront driver to re-use memory grants. Memory grants are Xen's mechanism to share memory between two virtual machines, in this case the packet buffers between dom0 and the ClickOS VM. By default, the driver requests a grant for *each* packet, requiring an expensive *hypercall* to the hypervisor (essentially the equivalent of a system call for an OS); we changed the driver so that it receives the grants for packet buffers at initialization time, and to re-use these buffers for all packets handled. The driver now also uses polling, further boosting performance.

The results are labeled "NF-MiniOS-opt" in Figure 2. We see important improvements in Rx rates, from 8 Kp/s to 344 Kp/s for maximum-sized packets. Still, this is far from the 10Gb/s line rate figure of 822 Kp/s, and quite far from the 14.8 Mp/s figure for minimum-sized packets, meaning that other significant bottlenecks remain.

Next, we took a look at the software switch. By default, Xen uses Open vSwitch, which previous work reports as capping out at 300 Kp/s [30]. As a result, we decided to replace it with the VALE switch [31]. Because VALE ports communicate using the netmap API,

description	function	ns
get vif	poll_net_schedule_list	119
handle frags if any	netbk_count_requests	53
alloc skb	alloc_skb reserve_skb	384
alloc page for packet data	xen_netbk_alloc_page	293
build grant op struct	fills *gnttab_copy*	96
extends the skb with the expected size	__skb_put	96
build grant op struct (for frags)	xen_netbk_get_requests	61
add the skb to the Tx queue	__skb_queue_tail	53
checks for packets received	check_rx_xenvif	206
packet grant copy	HYPERCALL	24708
dequeue packet from Tx queue	__skb_dequeue	94
copy pkt data to skb	memcpy	90
put a response in the ring	fills *xen_netif_tx_response* notify_via_remote_irq	52
copy frag data	xen_netbk_fill_frags	179
calc checksum	checksum_setup	78
forward pkt to bridge	xenvif_receive_skb	3446

Table 2: Per-function netback driver costs when sending a batch of 32 packets. Small or negligible costs are not listed for readability. Timings are in nanoseconds.

we modified the netback driver to implement that API, and removed the Xen virtual interface (*vif*) in the process. These changes ("NB-vale") gave a noticeable boost of up to 1.2 Mp/s for 64B packets, confirming that the switch was at least partly to blame [2].

Despite the improvement, the figures were still far from line rate speeds. Sub-optimal performance in the presence of a fast software switch, no *vif* and an optimized netfront driver seem to point to issues in the netback driver, or possibly in the communication between netback and netfront drivers. To dig in deeper, we carried out a per-function analysis of the netback driver to determine where the major costs were coming from.

The results in Table 2 report the main costs in the code path when transmitting a batch of 32 packets. We obtain timings via the `getnstimeofday()` function, and record them using the `trace_printk` function from the lightweight FTrace tracing utility.

The main cost, as expected, comes from the hypercall, essentially a system call between the VM and the hypervisor. Clearly this is required, though its cost can be significantly amortized by techniques such as batching. The next important overhead comes from transmitting packets from the netback driver through the *vif* and onto the switch. The *vif*, basically a tap device, is not fundamental to having a VM communicate with the netback driver

[2]We did not implement Rx on this modified netback driver as the objective was to see if the only remaining major bottleneck was the software switch.

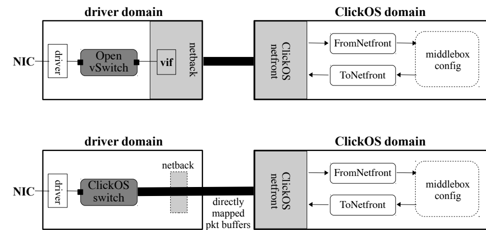


Figure 3: Standard Xen network I/O pipe (top) and our optimized, ClickOS one with packet buffers directly mapped into the VM's memory space.

and switch, but as shown adds non-negligible costs arising from extra queuing and packet copies. Other further penalties come from using the Xen ring API, which for instance requires responses to all packets transmitted in either direction. Finally, a number of overheads are due to `sk_buff` management, not essential to having a VM transmit packets to the network back-end – especially a non-Linux VM such as ClickOS.

In the next section we discuss how we revamped the Xen I/O network pipe in order to remove or alleviate most of these costs.

7 Network I/O Re-Design

The Xen network I/O pipe has a number of components and mechanisms that add overhead but that are not fundamental to the task of getting packets in and out of VMs. In order to optimize this, it would be ideal if we could have a more direct path between the back-end NIC and switch and the actual VMs. Conceptually, we would like to directly map ring packet buffers from the device driver or back-end switch all the way into the VMs' memory space, much like certain fast packet I/O frameworks do between kernel and user-space in non-virtualized environments [29, 25, 6].

To achieve this, and to boost overall performance, we take three main steps. First, we replace the standard but sub-optimal Open vSwitch back-end switch with a high-speed, ClickOS switch; this switch exposes per-port ring packet buffers which are able to we map into a VM's memory space. Second, we observe that since in our model the ClickOS switch and netfront driver transfer packets between one another directly, the netback driver becomes redundant. As a result, we remove it from the pipe, but keep it as a control plane driver to perform actions such as communicating ring buffer addresses (grants) to the netfront driver. Finally, we changed the VM netfront driver to map the ring buffers into its memory space.

These changes are illustrated in Figure 3, which contrasts the standard Xen network pipe (top diagram) with ours (bottom). We dedicate the rest of this section to providing a more detailed explanation of our optimized

switch, netback and netfront drivers (both MiniOS' and the Linux one) and finally a few modifications to Click.

ClickOS Switch. Given the throughput limitations of Xen's standard Open vSwitch back-end switch, we decided to replace it with the VALE high-speed switch [18], and to extend its functionality in a number of ways. First, VALE only supports virtual ports, so we add the ability to connect NICs directly to the switch. Second, we increase the maximum number of ports on the switch from 64 to 256 so as to accommodate a larger number of VMs.

In addition, we add support for each individual VM to configure the number of slots in the packet buffer ring, up to a maximum of 2048 slots. As we will see in the evaluation section, larger ring sizes can improve performance at the cost of larger memory requirements.

Finally, we modify the switch so that its switching logic is modular, and replace the standard learning bridge behavior with static MAC address-to-port mappings to boost performance (since in our environment we are in charge of assigning MAC addresses to the VMs this change does not in any way limit our platform's functionality). All of these changes have been now upstreamed into VALE's main code base.

Netback Driver. We redesign the netback driver to turn it (mostly) into a control-plane only driver. Our modified driver is in charge of allocating memory for the receive and transmit packet rings and their buffers and to set-up memory grants for these so that the VM's netfront driver can map them into its memory space. We use the Xen store to communicate the rings' memory grants to the VMs, and use the rings themselves to tell the VM about the ring buffers' grants; doing so ensures that the numerous grants do not overload the Xen store.

On the data plane side, the driver is only in charge of (1) setting up the `kthreads` that will handle packet transfers between switch and netfront driver; and (2) proxy event channel notifications between the netfront driver and switch to signal the availability of packets.

We also make a few other optimizations to the netback driver. Since the driver is no longer involved with actual packet transfer, we no longer use *vifs* nor OS-specific data structures such as `sk_buffs` for packet processing. Further, as suggested in [46], we adopt a 1:1 model for mapping kernel threads to CPU cores: this avoids unfairness issues. Finally, the standard netback uses a single event channel (a Xen interrupt) for notifying the availability of packets for both transmit and receive. Instead, we implement separate Tx and Rx event channels that can be serviced by different cores.

Netfront Driver. We modify MiniOS' netfront driver to be able to map the ring packet buffers exposed by the ClickOS switch into its memory space. Further, since the switch uses the netmap API [29], we implement a netmap module for MiniOS. This module uses the standard netmap data structures and provides the same abstractions as user-space netmap: `open`, `mmap`, `close` and finally `poll` to transmit/receive packets.

Beyond these mechanisms, our netfront driver includes a few other changes

- **Asynchronous Transmit:** In order to speed up transmit throughput, we modify the transmit function to run asynchronously.
- **Grant Re-Use:** Unlike the standard MiniOS netfront driver, we set-up grants once, and re-use them for the lifetime of the VM. This is a well-known technique for improving the performance of Xen's network drivers [35].
- **Linux Support:** While our modifications result in important performance increases, the departure from the standard Xen network I/O model means that we break support for other, non-MiniOS guests. To remedy this, we implemented a new Linux netfront driver suited to our optimized network pipe. Using this new netfront results in 10 Gb/s rates for most packet sizes (see Section 8) and allows us to run, at speed, any remaining middleboxes that cannot be easily implemented in Click or on top of MiniOS.

Click Modifications. Finally, we have made a few small changes to Click (version 2.0.1, less than 50 lines of code), including adding new elements to send and receive packets via the netfront driver, and optimizations to the `InfiniteSource` element to allow it to reach high packet rates.

ClickOS Prototype. The ClickOS prototype is open-source software. It includes changes to the XEN back-end (around 1000 LoC) and the frontend (1200 LoC). We are beginning to upstream these changes to Xen, but this process is lengthy; in the meantime, we plan to make the code available so that prospective users can just download our patches and recompile the netback and netfront modules (or recompile the dom0 kernel altogether).

8 Base Evaluation

Having presented the ClickOS architecture, its components and their optimization, we now provide a thorough base evaluation of the system. After this, in Section 9, we will describe the implementation of several middleboxes as well as performance results for them.

Experimental Set-up. The ClickOS tests in this section were conducted using either (1) a *low-end*, single-CPU Intel Xeon E3-1220 server with 4 cores at 3.1 GHz and 16 GB of DDR3-ECC RAM (most tests); or (2) a *mid-range*, single-CPU Intel Xeon E5-1650 server with 6 cores at 3.2 GHz and 16 GB of DDR3-ECC RAM (switch and scalability tests). In all cases we used Linux

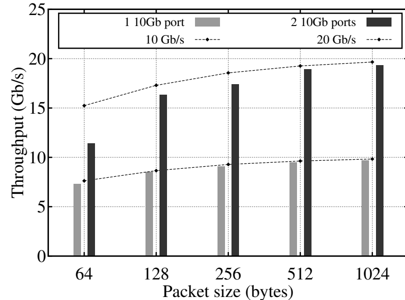


Figure 4: ClickOS switch performance using one and two 10 Gb/s NIC ports.

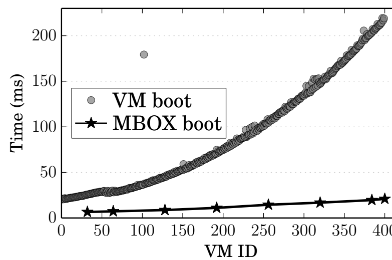


Figure 5: Time to create and boot 400 ClickOS virtual machines in sequence and to boot a Click configuration within each of them.

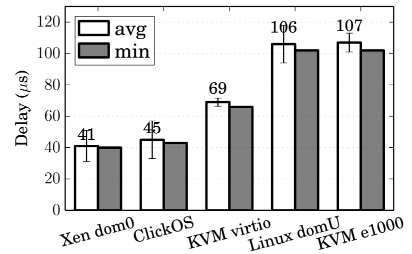


Figure 6: Idle VM ping delays for ClickOS, a Linux Xen VM, dom0, and KVM using the e1000 or virtio drivers.

3.6.10 for dom0 and domU, Xen 4.2.0, Click 2.0.1 and netmap's `pkt-gen` application for packet generation and rate measurements. All packet generation and rate measurements on an external box are conducted using one or more of the low-end servers, and all NICs are connected through direct cables. For reference, 10Gb/s equates to about 14.8 Mp/s for minimum-sized packets and 822 Kp/s for maximum-sized packets.

ClickOS Switch. The goal is to ensure that the switching capacity is high so that it does not become a bottleneck as more ClickOS VMs, cores and NICs are added to the system.

For this test we rely on a Linux (i.e., non-Xen) system. We use a user-space process running `pkt-gen` to generate packets towards the switch, and from there onto a single 10 Gb/s Ethernet port; a separate, low-end server then uses `pkt-gen` once again to receive the packets and to measure rates. We then add another `pkt-gen` user-process and 10Gb/s Ethernet port to test scalability. Each `pkt-gen`/port pair uses a single CPU core (so two in total for the 20Gb/s test).

For the single port pair case, the switch saturated the 10Gb/s pipe for all packet sizes (Figure 4). For the two port pairs case, the switch fills up the entire cumulative 20Gb/s pipe for all packet sizes except minimum-sized ones, for which it achieves 70% of line rate. Finally, we also conducted receive experiments (where packets are sent from an external box towards the system hosting the switch) which resulted in roughly similar rates.

Memory Footprint. As stated previously, the basic memory footprint of a ClickOS image is 5MB (including all the supported Click elements). In addition to this, a certain amount of memory is needed to allocate the netmap ring packet buffers. How much memory depends on the size of the rings (i.e., how many slots or packets the ring can hold at a time), which can be configured on a per-ClickOS VM basis.

To get an idea of how much memory might be required, Table 3 reports the memory requirements for different ring sizes, ranging from kilobytes for small rings all the way up to 8MB for a 2048-slot ring. As we will see later on in this section, this is a trade-off between the higher throughput that can be achieved with larger rings and the larger number of VMs that can be concurrently run when using small ring sizes. Ultimately, it might be unlikely that a single ClickOS VM will need to handle very large packet rates, so in practice a small ring size might suffice. It is also worth pointing out that larger rings require more memory grants; while there is a maximum number of grants per VM that a Xen system can have, this limit is configurable at boot time.

Ring size	Required memory (KB)	# of grants
64	264	65
128	516	129
256	1032	258
512	2064	516
1024	4128	1032
2048	8260	2065

Table 3: Memory requirements for different ring sizes.

What about the state that certain middleboxes might contain? To get a feel for this, we inserted 1,000 forwarding rules into an IP router, 1,000 rules into a firewall and 400 into an IDS (see Section 9 for a description of these middleboxes); the memory consumption from this was 20KB, 87KB and 30KB, respectively, rather small amounts. All in all, even if we use large ring sizes, a ClickOS VM requires approximately 15MB of memory.

Boot Times. In this set of tests we use the Cosmos tool to create ClickOS VMs and measure how long it takes for them to boot. A detailed breakdown of the ClickOS boot process may be found in [20]; for brevity, here we provide a summary. During boot up most of the time is spent issuing and carrying out the hypercall to create the VM (5.2 milliseconds), building the image (7.1 msecs) and creating the console (4.4 msecs), for a total of about 20.8 msecs. Adding roughly 1.4 msecs to attach the VM to the back-end switch and about 6.6 msecs to install a Click configuration brings the total to about 28.8 msecs

from when the command to create the ClickOS VM is issued until the middlebox is up and running.

Next we measured how booting large numbers of ClickOS VMs on the same system affects boot times. For this test we boot an increasing number of VMs in sequence and measure how long it takes for each of them to boot and install a Click configuration (Figure 5). Both the boot and startup times increase with the number of VMs, up to a maximum of 219 msecs boot and 20.0 msecs startup for the 400th VM. This increase is due to contention on the Xen store and could be improved upon.

Delay. Most middleboxes are meant to work transparently with respect to end users, and as such, should introduce little delay when processing packets. Virtualization technologies are infamous for introducing extra layers and with them additional delay, so we wanted to see how ClickOS' streamlined network I/O pipe would fare.

To set-up the experiment, we create a ClickOS VM running an ICMP responder configuration based on the `ICMPPingResponder` element. We use an external server to ping the ClickOS VM and measure RTT. Further, we run up to 11 other ClickOS VMs that are either idle, performing a CPU-intensive task (essentially an infinite loop) or a memory-intensive-one (repeatedly allocating and deallocating several MBs of memory).

The results show low delays of roughly 45 μsecs for the test with idle VMs, a number that stays fairly constant as more VMs are added. For the memory intensive task test the delay is only slightly worse, starting again at 45 μsecs and ramping up to 64 μsecs when running 12 VMs. Finally, the CPU intensive task test results in the largest delays (RTTs of up to 300 μsecs), though these are still small compared to Internet end-to-end delays.

Next, we compared ClickOS' idle delay to that of other systems such as KVM and other Xen domains (Figure 6). Unsurprisingly, dom0 has a small delay of 41 μsecs since it does not incur the overhead of going through the netback and netfront drivers. This overhead does exist when measuring delay for the standard, unoptimized netback/netfront drivers of a Xen Linux VM (106 μsecs). KVM, in comparison, clocks in at 69 μsecs when using its para-virtualized `virtio` drivers and 107 μsecs for its virtualized e1000 driver.

Throughput. In the next batch of tests we perform a number of baseline measurements to get an understanding of what packet rates ClickOS can handle. All of these tests are done on the low-end servers, with one CPU core dedicated to the VM and the remaining three to dom0.

Before testing a ClickOS VM we would like to benchmark the underlying network I/O pipe, from the NIC through to the back-end switch, netback driver and the netfront one. To do so, we employ our build tool to create a special VM consisting of only MiniOS and `pkt-gen` on top of it. After MiniOS boots, `pkt-gen` begins to immediately generate packets (for Tx tests) or measure rates (Rx). We conduct the experiment for different ring sizes (set using a `sysctl` command to the netmap kernel module) and for different packet sizes (for Tx tests this is set via Cosmos before the VM is created).

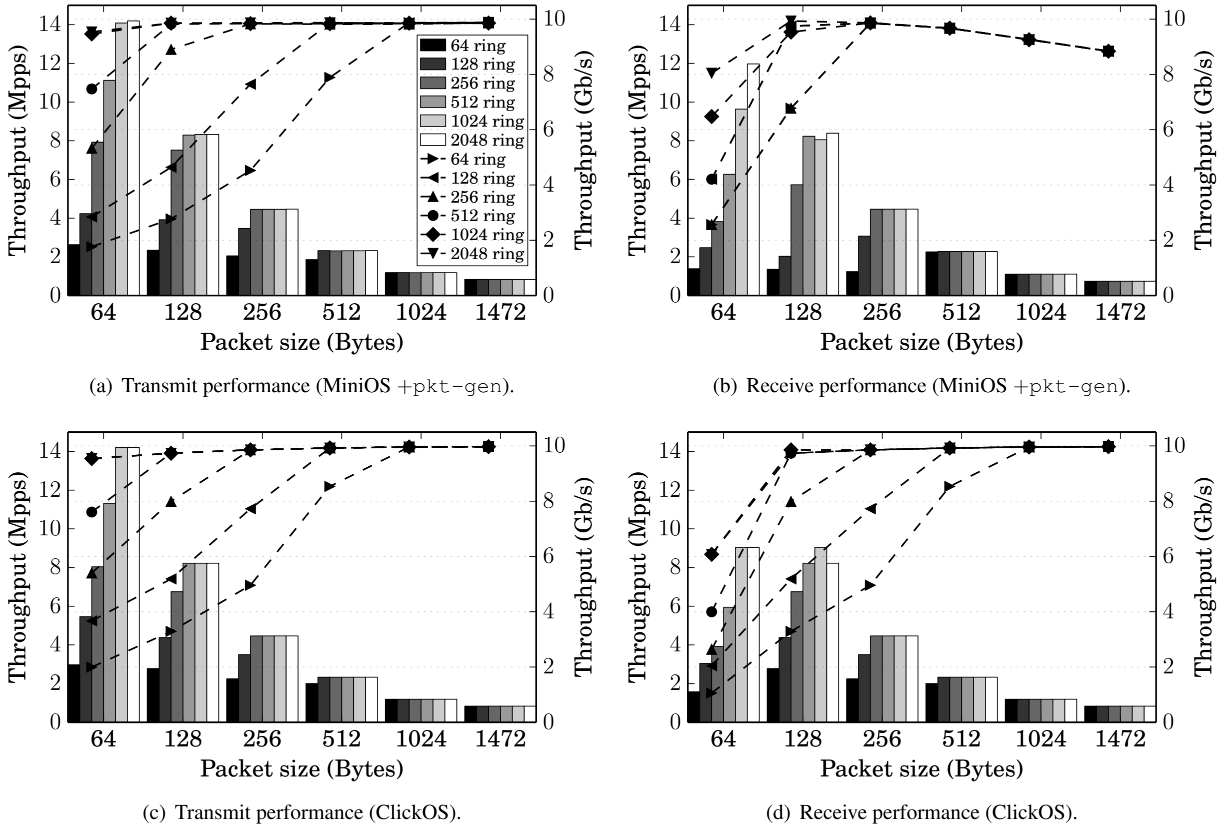
Figure 7 reports the results of the measurements. On transmit, the first thing to notice is that our optimized I/O pipe achieves close to line rate for minimum-sized packets (14.2 Mp/s using 2048-slot rings out of a max of 14.8 Mp/s) and line rate for all other sizes. Further, ring size matters, but mostly for minimum-sized packets. The receive performance is also high but somewhat lower due to extra queuing overheads at the netfront driver.

With these rates in mind, we proceed to deriving baseline numbers for ClickOS itself. In this case, we use a simple Click configuration based on the `AverageCounter` element to measure receive rates and another one based on our modified `InfiniteSource` to generate packets. Figure 7(c) shows ClickOS' transmit performance, which is comparable to that produced by the pkt-gen VM, meaning that at least for simple configurations ClickOS adds little overhead. The same is true for receive, except for minimum-sized packets, where the rate drops from about 12.0 Mp/s to 9.0 Mp/s.

For the last set of throughput tests we took a look at the performance of our optimized Linux domU netfront driver, comparing it to that of a standard netfront/Linux domU and KVM. For the latter, we used Linux version 3.6.10, the emulated e1000 driver, Vhost enabled, the standard Linux bridge, and `pkt-gen` once again to generate and measure rates. As seen in Figure 8 the Tx and Rx rates for KVM and the standard Linux domU are fairly similar, reaching only a fraction of line rate for small packet sizes and up to 7.88 Gb/s (KVM) and 6.46 Gb/s (Xen) for maximum-sized ones. The optimized netfront/Linux domU, on the other hand, hits 8.53 Mp/s for Tx and 7.26 Mp/s for Rx for 64-byte frames, and practically line rate for 256-byte packets and larger.

State Insertion. In order for our middlebox platform to be viable, it has to allow the middleboxes running on it to be quickly configured. For instance, this could involve inserting rules into a firewall or IDS, or adding extra external IP addresses to a carrier-grade NAT. In essence, we would like to test the performance of ClickOS element handlers and their use of the Xen store to communicate state changes. In this test we use Cosmos to perform a large number of reads and writes to a dummy ClickOS element with handlers, and measure how long these take for different transaction sizes (i.e., the number of bytes in question for each read and write operation).

Figure 9 reports read times of roughly 9.4 msecs and writes of about 0.1 msecs, numbers that fluctuate little



(a) Transmit performance (MiniOS +pkt-gen).

(b) Receive performance (MiniOS +pkt-gen).

(c) Transmit performance (ClickOS).

(d) Receive performance (ClickOS).

Figure 7: Performance of a single VM `pkt-gen` running on top of MiniOS/ ClickOS on a single CPU core, when varying the number of ring slots. The line graphs correspond to the right-hand y-axis.

across different transaction sizes. Note that read takes longer since it basically involves doing a write, waiting for the result, and then reading it. However, the more critical operation for middleboxes should be write, since it allows state insertion and deletion. For completeness, we also include measurements when using the XEN python API; in this case, the read and write operations jump to 10.1 and 0.3 msecs, respectively.

Chaining. Is it quite common for middleboxes to be chained one after the other in operator networks (e.g., a firewall followed by an IDS). Given that ClickOS has the potential to host large numbers of middleboxes on the same server, we wanted to measure the system's performance when chaining different numbers of middleboxes back-to-back. In greater detail, we instantiate one ClickOS VM to generate packets as fast as possible, another one to measure them, and an increasing number of intermediate ClickOS VMs to simply forward them. As with other tests, we use a single CPU core to handle the VMs and assign the rest to dom0.

As expected, longer chains result in lower rates, from 21.7 Gb/s for a chain of length 2 (just a generator VM and the VM measuring the rate) all the way down to 3.1 Gb/s for a chain with 9 VMs (Figure 10). Most of the decrease is due to the single CPU running the VMs being overloaded, but also because of the extra copy operations in the back-end switch and the load on dom0. The former could be alleviated with additional CPU cores; the latter by having multiple switch instances (which our switch supports) or driver domains (which Xen does).

Scaling Out. In the final part of our platform's base evaluation we use our mid-range server to test how well ClickOS scales out with additional VMs, CPU cores and 10 Gb/s NICs. For the first of these, we instantiate an increasing number of ClickOS VMs, up to 100 of them. All of them run on a single CPU core and generate packets as fast as possible towards an outside box which measures the cumulative throughput. In addition, we measure the *individual* contribution of each VM towards the cumulative rate in order to ensure that the platform is fairly scheduling the VMs: all of VMs contribute equally to the rate and that none are starved.

Figure 11 plots the results. Regardless of the number of VMs, we get a cumulative throughput equivalent to line rate for 512-byte packets and larger and a rate of 4.85 Mp/s for minimum-sized ones. The values on top of the bars represent the standard deviation for all the individual

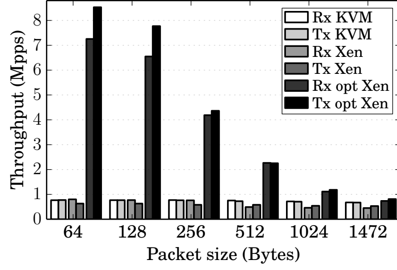


Figure 8: Linux domU performance with an optimized (opt) netmap-based netfront driver versus the performance of out-of-the-box Xen and KVM Linux virtual machines.

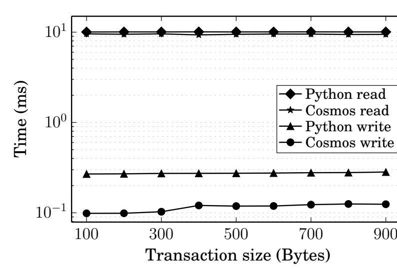


Figure 9: ClickOS middlebox state insertion (write) and retrieval (read) for different transaction sizes (log scale).

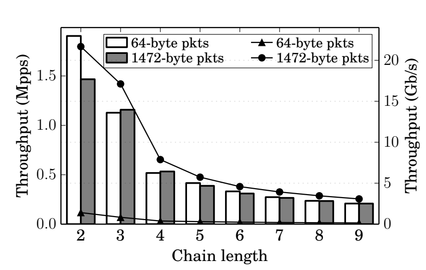


Figure 10: Performance when chaining ClickOS VMs back-to-back. The first VM generates packets, the ones in the middle forward them and the last one measures rates. Ring size is set to 64 slots.

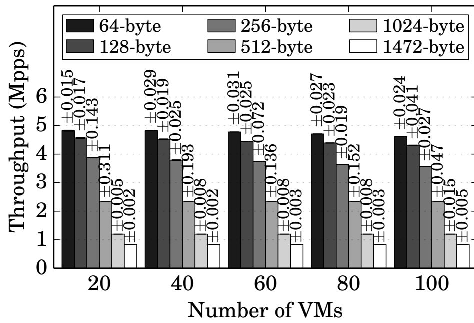


Figure 11: Running many ClickOS packet generator VMs on one core and a 10 Gb/s port. Fairness is shown by the low standard deviations above the the bars.

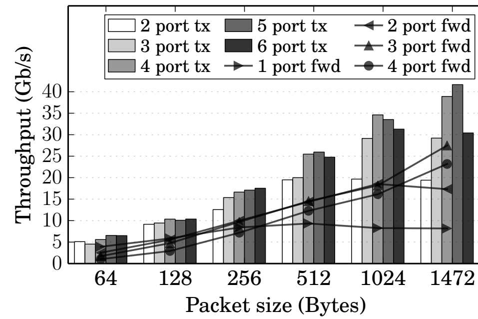


Figure 12: Cumulative throughput when using multiple 10 Gb/s ports and one ClickOS VM per port to (1) send out traffic (tx) or (2) forward traffic (fwd).

rates contributed by each VM; the fact that these values are rather low confirms fairness among the VMs.

Next, we test ClickOS' scalability with respect to additional CPU cores and 10 Gb/s ports. We use one packet generator ClickOS VM per port, up to a maximum of six ports. In addition, we assign two cores to dom0 and the remaining four to the ClickOS VMs in a round-robin fashion. Each pair of ports is connected via direct cables to one of our low-end servers and we calculate the cumulative rate measured at them; ring size is 1024.

For maximum-sized packets we see a steady, line-rate increase as we add ports, VMs and CPU cores, up to 4 ports (Figure 12). After this point, VMs start sharing cores (our system has six of them, with four of them assigned to the VMs) and the performance no longer scales linearly. For the final experiment we change the configuration that the ClickOS VMs are running from a packet generator to one that bounces packets back onto the same interface that they came on (line graphs in Figure 12). In this configuration, ClickOS rates go up to 27.5 Gb/s.

Scaling these experiments further requires a CPU with more cores than in our system, or adding NUMA support to ClickOS so that performance scales linearly with additional CPU packages; the latter is our future work.

9 Middlebox Implementations

Having evaluated the baseline performance of ClickOS, we now turn our attention to evaluating its performance when running actual middleboxes. Clearly, since the term middleboxes covers a wide range of processing, exhaustively testing them all is impossible. We therefore evaluate the performance of ClickOS on a set candidate middleboxes which vary in the type of workload they generate.

For these set of tests we use two of our low-end servers connected via two direct cables, one per pair of Ethernet ports. One of the servers generates packets towards the other server, which runs them through a ClickOS middlebox and forwards them back towards the first server where their rate is measured. The ClickOS VM is assigned a single CPU core, with the remaining three given to dom0. We test each of the following middleboxes:

Wire (WR): A simple "middlebox" which sends packets from its input to its output interface. This configuration serves to give a performance baseline.

EtherMirror (EM): Like wire, but also swap the Ethernet source and destination fields.

IP Router (IR): A standards-compliant IPv4 router configured with a single rule.

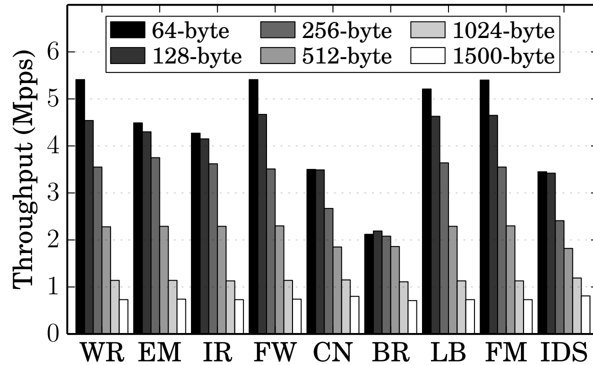


Figure 13: Performance for different ClickOS middleboxes and packet sizes using a single CPU core.

Firewall (FW): Based on the `IPFilter` element and configured with ten rules, none matching any packets.
Carrier Grade NAT (CN): An almost standards-compliant carrier-grade NAT. To stress the NAT, each packet has a different set of source and destination port numbers. Using a single flow/set of ports results in a higher rate of 5.1 Mp/s for minimum-sized packets.
Software BRAS (BR): An implementation of a Broadband Remote Access Server (BRAS), including PPPoE session handling. The data plane checks session numbers and PPPoE/PPP message types, strips tunnel headers, and performs IP lookup and MAC header re-writing.
Intrusion Detection System (IDS): A simple Intrusion Detection System based on regular expression matching. The reported results are for a single rule that matches the incoming packets.
Load Balancer (LW): This re-writes packet source MAC addresses in a round-robin fashion based on the IP src/dst, port src/dst and type 5-tuple in order to split packets to different physical ports.
Flow Monitor (FM) retains per flow (5-tuple) statistics.

Figure 13 reports throughput results for the various middleboxes. Overall, ClickOS performs well, achieving almost line rate for all configurations for 512-byte and larger packets (the BRAS and CG-NAT middleboxes have rates slightly below the 2.3 Mp/s line rate figure). For smaller packet sizes the percentage of line rate drops, but ClickOS is still able to process packets in the millions/second.

To get and idea of how this relates to a real-world traffic matrix, compare this to an average packet size of 744 bytes reported by a recent study done on a tier-1 OC192 (about 10Gb/s) backbone link [34]: if we take our target to be packets of around this size, all middleboxes shown can sustain line rate.

Naturally, some of these middleboxes fall short of being fully functional, and different configurations (e.g., a large number of firewall rules) would cause their performance to drop from what we present here. Still, we believe these figures to be high enough to provide a sound basis upon which to build production middleboxes. The carrier-grade NAT, for instance, is proof of this: it is fully functional, and in stress tests it is still able to handle packets in the millions/second.

10 Conclusions

This paper has presented ClickOS, a Xen-based virtualized platform optimized for middlebox processing. ClickOS can turn Network Function Virtualization into reality: it runs hundreds of middleboxes on commodity hardware, offers millions of packets per second processing speeds and yields low packet delays. Our experiments have shown that a low-end server can forward packets at around 30Gb/s.

ClickOS is proof that software solutions alone are enough to significantly speed up virtual machine processing, to the point where the remaining overheads are dwarfed by the ability to safely consolidate heterogeneous middlebox processing onto the same hardware. ClickOS speeds up networking for all Xen virtual machines by applying well known optimizations including reducing the number of hypercalls, use of batching, and removing unnecessary software layers and data paths.

The major contribution of ClickOS is adopting Click as the main programming abstraction for middleboxes and creating **a tailor-made guest operating system to run Click configurations**. Such specialization allows us to optimize the runtime of middleboxes to the point where they boot in milliseconds, while allowing us to support a wide range of functionality. Our implementations of a software BRAS and a Carrier-Grade NAT show that ClickOS delivers production-level performance when running real middlebox functionality.

In the end, we believe that ClickOS goes beyond replacing hardware middleboxes with the software equivalent. Small, quick-to-boot VMs make it possible to offer personalized processing (e.g., firewalls) to a large number of users with comparatively little hardware. Boot times in the order of milliseconds allow fast scaling of processing dynamically (e.g., in response to a flash crowd) as well as migration with negligible down-time. Finally, ClickOS could help with testing and deployment of new features by directing subsets of flows to VMs running experimental code; issues with the features would then only affect a small part of the traffic, and even VMs crashing would not represent a major problem since they could be re-instantiated in milliseconds.

Acknowledgments

The authors are in debt to Adam Greenhalgh for initial ideas and work towards ClickOS. The research leading to these results was partly funded by the EU FP7 CHANGE project (257322).

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *USENIX Conference*, pages 93–112, 1986.

[2] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon. Software techniques for avoiding hardware virtualization exits. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 35–35, Berkeley, CA, USA, 2012. USENIX Association.

[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 *Proc. ACM SOSP, 2003*, New York, NY, USA, 2003. ACM.

[4] A. Cardigliano, L. Deri, J. Gasparakis, and F. Fusco. vpf_ring: towards wire-speed network monitoring using virtual machines. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 533–548, New York, NY, USA, 2011. ACM.

[5] Cisco. Cisco Cloud Services Router 1000v Data Sheet. http://www.cisco.com/en/US/prod/collateral/routers/ps12558/ps12559/data_sheet_c78-705395.html, July 2012.

[6] L. Deri. Direct NIC Access. http://www.ntop.org/products/pf_ring/dna/, December 2011.

[7] 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 *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. ACM.

[8] ETSI. Leading operators create ETSI standards group for network functions virtualization. http://www.etsi.org/index.php/news-events/news/644-2013-01-isg-nfv-created, September 2013.

[9] 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.

[10] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 1–12, New York, NY, USA, 2012. ACM.

[11] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[12] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend tcp? In *Proc. ACM IMC*, 2011.

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

[14] Intel. Intel Virtualization Technology for Connectivity. http://www.intel.com/content/www/us/en/network-adapters/virtualization.html, September 2013.

[15] M. C. Kaushik Kumar Ram, Alan L. Cox and S. Rixner. Hyper-switch: A scalable software virtual switching architecture. In *Proc. of USENIX Annual Technical Conference*, 2013.

[16] A. Kivity, Y. Kamay, K. Laor, U. Lublin, and A. Liguori. Kvm: The linux virtual machine monitor. In *Proc. of the Linux Symposium*, 2007.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems, August 2000*, 2000.

[18] Luigi Rizzo. VALE, a Virtual Local Ethernet. http://info.iet.unipi.it/~luigi/vale/, July 2012.

[19] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, Mar. 2013.

[20] J. Martins, M. Ahmed, C. Raiciu, and F. Huici. Enabling fast, dynamic network processing with clickos. In *HotSDN*, pages 67–72, 2013.

[21] Microsoft Corporation. Microsoft Hyper-V Server 2012. http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx, September 2013.

[22] Minix3. Minix3. http://www.minix3.org/, July 2012.

[23] MIT Parallel and Distributed Operating Systems Group. MIT Exokernel Operating System. http://pdos.csail.mit.edu/exo.html, March 2013.

[24] Nadav HarEl and Abel Gordon and Alex Landau and Muli Ben-Yehuda and Avishay Traeger and Razya Ladelsky. Efcient and Scalable Paravirtual I/O System. In *Proc. of USENIX Annual Technical Conference*, 2013.

[25] N.Bonelli, A. D. Pietro, S. Giordano, and G. Procissi. On multi–gigabit packet capturing with multi–core commodity hardware. In *Passive and Active Measurement conference (PAM)*, 2012.

[26] Open vSwitch. Production Quality, Multilayer Open Virtual Switch. http://openvswitch.org/, March 2013.

[27] Openvz.org. OpenVZ Linux Containers. http://openvz.org/Main_Page, September 2013.

[28] K. K. Ram, J. R. Santos, Y. Turner, A. L. Cox, and S. Rixner. Achieving 10 gb/s using safe and transparent network interface virtualization. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 61–70, New York, NY, USA, 2009. ACM.

[29] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. USENIX Annual Technical Conference*, 2012.

[30] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In A. G. Greenberg and K. Sohraby, editors, *INFOCOM*, pages 2471–2479. IEEE, 2012.

[31] L. Rizzo and G. Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 61–72, New York, NY, USA, 2012. ACM.

[32] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet i/o in virtual machines. In *Proc. ACM/IEEE ANCS*, 2013.

[33] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[34] P. M. Santiago del Rio, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil. Wire-speed statistical classification of network traffic on commodity hardware. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, IMC '12, pages 65–72, New York, NY, USA, 2012. ACM.

[35] J. R. Santos, Y. Turner, G. Janakiraman, and I. Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[36] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[37] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratsanamy, and V. Sekarl. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. ACM SIGCOMM*, 2012.

[38] Sourceware.org. The Newlib Homepage. http://sourceware.org/newlib/, September 2013.

[39] Swig.org. Simplified Wrapper and Interface Generator. http://swig.org, September 2013.

[40] VMware. VMware Virtualization Software for Desktops, Servers and Virtual Machines for Public and Private Cloud Solutions. http://www.vmware.com, July 2012.

[41] Vyatta. The Open Source Networking Community. http://www.vyatta.org/, July 2012.

[42] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, Dec. 2002.

[43] Wikipedia. L4 microkernel family. http://en.wikipedia.org/wiki/L4_microkernel_family, July 2012.

[44] Wikipedia. FreeBSD Jail. http://en.wikipedia.org/wiki/FreeBSD_jail, September 2013.

[45] Wikipedia. Solaris Containers. http://en.wikipedia.org/wiki/Solaris_Containers, September 2013.

[46] Xen Blog. Xen Network: The Future Plan. http://blog.xen.org/index.php/2013/06/28/xen-network-the-future-plan/, September 2013.

[47] C. Xu, S. Gamage, H. Lu, R. Kompella, and D. Xu. vturbo: Accelerating virtual machine i/o processing using designated turbo-sliced core. In *Proc. of USENIX Annual Technical Conference*, 2013.