Enabling Fast, Dynamic Network Processing with ClickOS

Joao Martins[†], Mohamed Ahmed[†], Costin Raiciu[‡], Felipe Huici[†]

[†] NEC Europe Ltd. [‡] University Politehnica of Bucharest
firstname.lastname@neclab.eu, costin.raiciu@cs.pub.ro

ABSTRACT

Middleboxes are both crucial to today's networks and ubiquitous, but embed knowledge of today's protocols and applications to the detriment of those of tomorrow, making the network harder to evolve. SDNs seek to make it easier to extend the network with new functionality, but most of the research effort has focused on the network's control plane, that is, how packets are switched are routed through a SDN.

Given the pervasiveness and importance of middleboxes, we believe that a fully programmable network should also be able to dynamically instantiate and quickly move middlebox functionality. In this paper we shift focus towards making the data plane more programmable by introducing ClickOS, a tiny, Xen-based virtual machine that can run a wide range of middleboxes. ClickOS is small (5MB when running), can be instantiated in very small times (roughly 30 milliseconds) and can fill up a 10Gb pipe while concurrently running 128 vms on a low-cost commodity server.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management

General Terms

Performance

Keywords

Virtualization, Xen, middleboxes, SDN, NFV, ClickOS

1. INTRODUCTION

We are witnessing a revival of Internet architecture research that leverages software defined networking (SDN) to overcome the limitations of the current network, the biggest of which is ossification. To create an evolvable network, SDN proposes centralization of the control plane and commoditization of the data plane, thus removing the long-standing "you can't touch the core" deployment barrier. By creating an easily changeable software control plane, SDN allows flexible packet switching and routing, promising to accelerate the adoption of new Internet protocols (e.g., IPv6) or changes to existing ones.

Unfortunately, changing the way packets are switched may not be enough to evolve the Internet: there is a lot more functionality embedded in the network today—in the form of middleboxes—that also needs updating. Middleboxes are crucial to today's operational networks, performing 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 the performance of network applications (traffic accelerators, caches, proxies). Middleboxes are almost ubiquitous: one recent study [6] found that a third of access networks maintain TCP connection state, performing various flow processing functionality. Another study of enterprise networks concludes that there are as many middleboxes deployed in networks as routers and switches [23].

Given the pervasiveness and importance of middleboxes, an evolvable Internet must be able to dynamically instantiate, upgrade and quickly shift middlebox functionality. In this paper we switch focus towards making the *data* plane of SDNs more programmable, that is, the actual processing that packets go through as they pass through networks.

Middleboxes are, for the most part, costly, purpose-built hardware devices that are difficult to configure or upgrade, much like routers. The obvious solution to evolvable middleboxes is to instantiate them as software on commodity hardware, perhaps in the cloud [26, 23]. For cost-effectiveness and scalability reasons, software middleboxes will run inside virtual machines, allowing them to be consolidated on a single physical machine when traffic is low, and migrated to idle machines when traffic ramps up.

What is the right programming interface for software middleboxes? Before we jump to an answer, let's consider the requirements of a good solution:

- **Fast instantiation**: the system should be able to instantiate middlebox processing quickly and where needed, in order to match (possibly rapidly) changing SDNs.
- **Small Footprint**: ideally the system should be able to host a large number of middleboxes on the same server in order to reduce purchasing and operational costs.
- **Isolation**: in a world where slices of networks are given to different entities and users, it becomes increasingly important that users' middleboxes that happen

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotSDN'13, August 16, 2013, Hong Kong, China.
Copyright 20XX ACM 978-1-4503-2178-5/13/08 ...$15.00.

to run on common hardware do not affect each other, both from a security and performance point of view.
- **Performance**: in order to have a chance at competing with hardware offerings, the system should provide a high-performance data plane, including driver support for the latest network devices.
- **Flexibility**: the system should be able to easily perform a wide range of middlebox functionality and be extensible.

The default solution is to run middleboxes inside Linux VMs (or other commodity OSes). This solution provides isolation and performance, but has a large memory footprint and relatively slow instantiation times (roughly 5 seconds in some of our Linux-based tests). Typically, such software middleboxes are developed using general purpose programming languages, e.g. C or python.

Can we do better? We observe that middlebox processing is not general purpose computation: it works at packet-level and applies a relatively small menu of changes to packets. The Click modular router [8] is a good way to program middlebox functionality: users can combine existing stock elements or write new ones to quickly create complex processing configurations. To date, a wide range of networking processing has been implemented in Click [3].

Click offers flexibility and performance—but it lacks isolation, and has a large memory footprint as it runs in the Linux kernel. The coupling of Click with the Linux kernel is not fundamental: we propose to break it and run Click as a standalone virtual machine.

In this paper we introduce ClickOS, a Xen-based tiny virtual machine that runs Click. Through optimizations to the virtual machine itself and Xen's underlying networking system, ClickOS achieves the criteria above: it can be quickly instantiated (boot times are in the ballpark of as little as 30 milliseconds), it has a small footprint (the compressed image is 1.4MB and 5MB when running), and can process a 10Gb pipe worth of traffic. In addition, it benefits from the inherent isolation provided by Xen and the flexibility afforded by the Click modular software.

The rest of this paper is dedicated to explaining ClickOS and providing a performance evaluation of it. It is organized as follows. Section 2 gives an overview of ClickOS. Section 3 presents results concerning ClickOS' fast boot times and small footprint, as well as an evaluation of the system's data plane performance. Section 4 discusses related work. Finally, section 5 discusses the wider implications of ClickOS on middlebox processing.

2. CLICKOS ARCHITECTURE

We use Xen to create a scalable and easily programmable architecture. The system runs a set of ClickOS virtual machines (vms), each composed of Click version 2.0.1 running on top of MiniOS, a minimalistic OS provided with the Xen sources (figure 1). Xen was chosen because it uses paravirtualization to run slightly modified operating-systems as guests, offering better performance compared to full virtualization solutions (where the guest OS is unmodified) [1]. We motivate using MiniOS in section 2.1.

To run Click, users provide a configuration, essentially a text file specifying a graph of inter-connected elements. Once running, they can access read/write *handlers*, internal variables that can change the state of an element at runtime (e.g., the *AverageCounter* element has a read handler

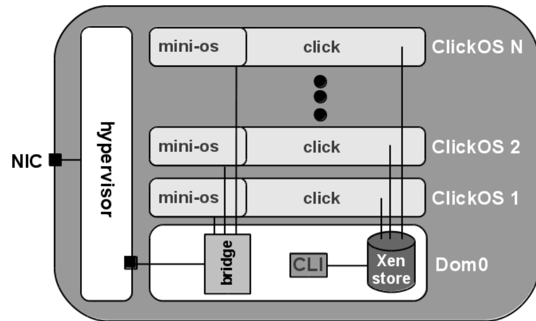

Figure 1: ClickOS architecture overview.

to get the number of packets seen so far, and a write handler to reset that count). Click relies on the `/proc` filesystem or sockets to provide these mechanisms; because these do not exist in ClickOS, we must provide an equivalent way of implementing them.

The ClickOS control plane in charge of handling all of these operations consists of three parts. First, a C-based CLI takes care of, among other tasks, creating and destroying ClickOS guest domains. When a guest domain boots, a MiniOS control thread is created (the second part of the control plane). This thread adds an entry to the Xen store, a `/proc`-like database shared between dom0 and all running guest domains (figure 1). The control thread then watches for changes to the entry. When a Click configuration string is written to it, it takes care of creating a new thread and running a Click instance within it, meaning that several Click instances can run within a single ClickOS domain.

The third part of the control plane consists of a new Click element called *ClickOSControl*. It talks, on one end, to all elements in a given configuration and to the Xen store on the other end. The CLI then provides users with an interface to read and write to element handlers via the Xen store and *ClickOSControl*. All these operations on the ClickOS side of things are executed in the control thread mentioned above.

2.1 Building ClickOS

Xen is split into a privileged domain called dom0 that (among other tasks) controls the hypervisor and hosts device drivers[1]; and guest domains, the users' virtual machines (also knows as domUs).

To achieve good performance, Click should run in the operating system kernel, but this makes crashing the whole system very easy - so Click must necessarily run inside a virtual machine. Using a full blown Linux virtual machine to run Click, as done today, is rather heavyweight, leading to large memory footprints and long boot-up times.

We observe that many of the services provided by a Linux kernel [2] are not needed to run Click, and neither is the userspace API. First, a ClickOS instance will run a single configuration belonging to a single user: there is no need for multiple user support. In fact, there is no need for user-space programs; removing the user-space/kernel space separation

[1]Strictly speaking, the device drivers are hosted in the driver domain, but in practice dom0 frequently also acts as the driver domain.

[2]The same observations apply for other commodity operating systems, not just Linux.

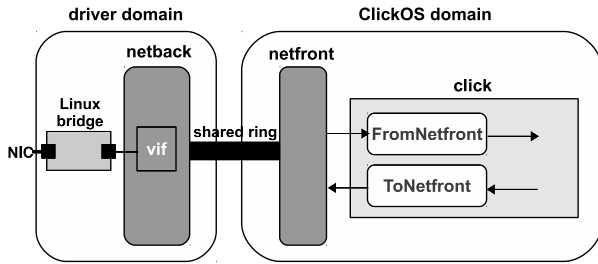

Figure 2: Basic ClickOS networking in Xen.

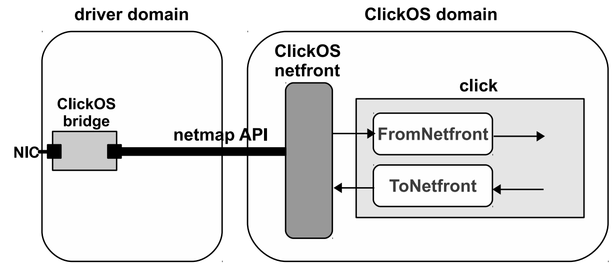

Figure 3: Optimized networking: modified netfront driver, revamped network back-end using the netmap API, and an improved VALE-based software switch.

increases performance (e.g., no system calls) and simplifies the kernel considerably.

In addition, Click does not need multiple memory address spaces: a single configuration runs in the same address space, as elements pass pointers to each other to perform their functionality; this implies that support for multiple processes is not needed either. Threads are needed, though, as different processing chains may need to run in parallel.

On the I/O side, Click needs access to network interfaces. These can be easily supported with a generic driver, leaving the complexity of providing drivers for different hardware in dom0. Filesystems are rarely used, and other I/O devices (e.g., usb, video output) even less. In fact, removing most of the I/O supports deprecates a big part of the Linux kernel. Finally, a networking stack may be needed for TCP/IP connectivity.

As it turns out, the Xen sources come with MiniOS, a minimalistic, para-virtualized OS that provides all the functionality needed by Click without any of the additional "clutter" included in traditional operating systems. As a result, we build ClickOS as a combination of Click and MiniOS.

MiniOS implements the basics needed to operate in a Xen environment: grant tables for sharing memory with other domains including dom0, a netfront driver for packet I/O, event channels (Xen interrupts) and the Xen store driver. In addition, MiniOS has a single address space, no kernel/user separation, and runs a co-operative scheduler, reducing context switch costs.

In order to build and link Click and MiniOS we first needed to have a Linux independent `c++` cross-compiler (Click is written in `c++`). To do so, we built a new toolchain (gcc, ld, ar, etc) that uses the platform independent `newlibc` library instead of `glibc`.

In addition, we had to adapt certain parts of Click to work in a MiniOS environment instead of a Linux or FreeBSD one. These include the creation of the *ClickOSControl* element previously mentioned for dealing with Click element handlers, as well as new Click elements that act as network devices that can talk to the MiniOS netfront driver (see next section).

2.2 ClickOS Networking

Xen has a split network driver model, whereby a *netback* driver running in a driver domain talks to hardware devices and exports a common, ring-based API; and a *netfront* driver running in a guest domain (e.g., ClickOS) talks to the netback driver via shared memory (i.e., the ring). This split model allows guest domains to have access to hardware devices without having to themselves host their drivers.

Under a typical Xen set-up, a network card is linked to a virtual network device called a *vif* via the Linux bridge module or in later Xen versions Open vSwitch (figure 2). When a packet is received, it is forwarded to the *vif* whose MAC address matches that of the packet's destination. The device then takes the packet and queues it at the netback driver. At a later point in time, one of the netback driver threads picks up the packet and puts it on the shared ring, notifying the netfront driver in the process; packets sent out follow a similar path in the opposite direction.

In order to interact with the netfront driver, we created two new Click elements, *FromNetfront* and *ToNetfront*. The first of these takes care of initializing a network device and, each time it is scheduled, retrieves *burst* number of packets from the netfront driver, where burst is a configuration parameter. The *ToNetfront* element is pretty straightforward, simply calling the netfront's transmit function.

Without optimizations, this network data path performs rather poorly, in the range of only 8 Kp/s for maximum-sized packets, and even with a Linux-based vm we experienced 2.9 Gb/s, confirming the figure given in [18]. To push this up to the 10Gb supported by our cards, we had to undertake a number of improvements [3].

First, we optimized the netfront driver by introducing two mechanisms: we changed the driver's receive function to poll for packets from the MiniOS thread running Click, rather than be interrupt driven, and we added a burst parameter to process packets in batches. Second, we re-used the grants that receive buffers are given and keep them for the lifetime of the network device (a grant is Xen's way of allowing two domains to share memory).

The next bottleneck was the netback driver and the ring-based API. Optimizing this required an overhaul of the Xen backplane. In greater detail, we got rid of the vifs and replaced the netback driver with a netmap-based [20] one that directly maps the network buffers of each port of the back-end software switch onto a vm's local memory[4]. This provides us a much more direct network path between the NIC and the vm and thus better performance.

The final major bottleneck was the software switch (Open vSwitch in recent Xen releases), which has been shown to yield poor performance, at most 300 Kp/s for the kernel ver-

[3]Note: line rate is roughly 810 Kp/s for maximum-sized packets and 14.8 Mp/s for minimum-sized ones.

[4]We chose netmap since it is able to handle packet transfers at rates of 10Gb/s and higher while consuming relatively few CPU cycles.

sion [21]. We replaced this switch with VALE [10], a netmap-based switch designed for high-speed rates, and adapted it to be able to interact with the ClickOS netfront driver.

3. EVALUATION

We now present results evaluating various aspects of ClickOS. We perform all tests on a couple of x86 commodity servers, each with two quad-core Intel Xeon E5620 2.4GHz processors (with hyper-threading disabled), 24GB of memory and an Intel x520-t 10Gb adapter. One server acts as a packet generator and sink, and the other runs Xen 4.1.2 and the ClickOS vms. The servers are connected using direct cabling.

ClickOS is compiled with most of the available Click elements (216/282), many of the remaining ones requiring a file system to work (we are in the process of porting a simple file system to increase the number of available elements). The amount and variety of elements means that ClickOS can support a wide range of middleboxes, including firewalls, proxies, load balancers and NATs, to name a few. It is also relatively easy to extend this functionality: adding a few new elements we were able to create a carrier-grade NAT, a software BRAS and a simple IDS.

3.1 Middlebox Instantiation

While flexibility is important, it is imperative that middleboxes can be instantiated quickly. In order to set up and run a ClickOS middlebox, several steps take place. First, the virtual machine is created, which includes reading its configuration, its image file, writing a (large) number of entries to the Xen store (e.g., the id of the vm, addresses for memory allocations, etc) and creating the vm itself. Second, we "attach" a virtual network device to the vm, which adds more entries to the Xen store, and connect the device to the software switch.

Once MiniOS boots, the ClickOS control thread is created. When this thread receives a new Click configuration, it starts a new Click thread, which in turn populates the Xen store with entries used to support element handlers, initializes the network device that had been previously created, and finally sets the middlebox running.

When we first started measuring the ClickOS start-up time we were getting results in the range of several seconds, but a quick investigation showed that that was due to the Xen store residing in an NFS-mounted drive; moving it to a RAM disk lowered this timing to just over a second. The second optimization was moving from the `xm` toolstack [5] to the newer `xl` one (`xm` was Python-based and made use of slow xml-rpc calls to carry out its operations). This change, along with switching to the newer, more efficient `oxenstore` resulted in a reduction of start-up times from roughly 0.86 to 0.21 seconds.

The final improvements had to do with trimming unnecessary operations from the MiniOS and Click start-up process, which brought the total time down to about 70 msecs. At this point, we discovered that reading the compressed virtual machine image was slow; providing an uncompressed image instead cut down the overall start-up time to about 30 msecs. Table 1 gives a detailed breakdown of the process up to the creation of the ClickOS control thread (we call this the ClickOS boot time); creating a network device and starting the Click middlebox (the start-up time) consumes an additional 8 msecs or so, roughly 30 msecs in total.

description	function	time
issue create hypercall	libxl__domain_make2	5.244
paravirt. bootloader	libxl_run_bootloader	0.049
prepare domain boot	libxl__build2_pre	0.089
parse, allocate and boot vm image	xc_dom_allocate	0.016
	xc_dom_kernel_path	0.047
	xc_dom_ramdisk	0.001
	xc_dom_boot_xen_init	0.011
	xc_dom_parse_image	0.286
	xc_dom_mem_init	0.007
	xc_dom_boot_mem_init	0.650
	xc_dom_build_image	7.091
	xc_dom_boot_image	0.707
write xen store entries, notify xen store daemon	libxl__build2_post	2.202
init console	init_console_info	0.004
	libxl__need_xenpv_qemu	0.006
	libxl__device_console_add	4.371
	TOTAL	**20.789**

Table 1: Costs of creating a ClickOS virtual machine and booting it up, in milliseconds.

So far the discussion has focused on a single ClickOS vm, but how long would it take to actually create many of them? Figure 4 shows boot times for 400 ClickOS vms on a single server. The first vm takes the roughly 21 msecs described in the table above, ramping up to about 200 msecs for the 400th vm. This shows ClickOS' ability to quickly instantiate processing even in the presence of several existing vms.

We further measured how long it takes to install a middlebox configuration in one of these ClickOS vms. Figure 5 shows such timings when different number of vms already exist in the system. Again, these costs are small: about 7 msecs when 64 vms are already running, all the way up to 21 msecs for 400 vms.

3.2 Networking Performance

The path that packets follow from a ClickOS vm through the Xen networking system and eventually to the NIC is complex and involves a number of components (recall figure 2). The first throughput measurements we conducted were rather disappointing: only about 1% of our 10Gb card for maximum-sized packets.

In order to identify where the bottlenecks were, we began with a single ClickOS vm with one CPU core assigned to it running a simple Click configuration [6] that creates packets as fast as possible and drops them. The first issue was with Click's packet generation: after several modifications to `InfiniteSource` that we leave out due to space restrictions, we were able to bump its rate from 8.4 Mp/s to 13.2 Mp/s, almost enough to saturate a 10Gb link for all packet sizes. This allowed us to investigate other problems with the

[5] The toolstack is the user interface used to control Xen; this includes vm creation.

[6] `InfiniteSource`→`EtherEncap`→`Discard`

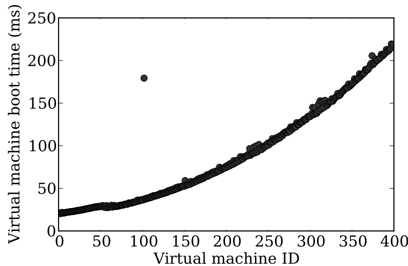

Figure 4: Time to create and boot 400 ClickOS virtual machines on a single server.

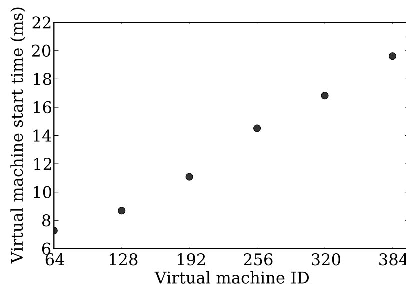

Figure 5: Time to instantiate processing in a ClickOS vm once other vms are running.

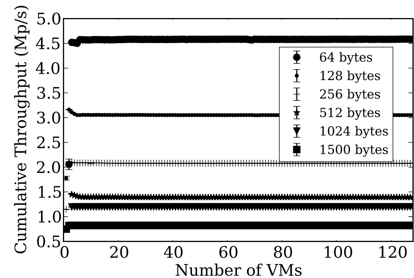

Figure 6: Aggregated throughput when 128 ClickOS virtual machines are active.

knowledge that the bottlenecks would not come from Click and the configuration.

Next we added a `ToNetfront` element to the previous configuration in order to test the netfront driver's performance; the throughput was a dismal 8 Kp/s for maximum-sized packets. A number of modifications (e.g., turning the driver from an interrupt-driven one to a polling one and re-using the grants that Xen uses to allow domains to share memory and thus packets) resulted in a significant improvement, up to about 360 Kp/s.

The next component in the transmit path is the netback driver. As a quick test, we replaced it with a minimalistic version that does nothing more than take packets from the netfront driver (via the Xen ring API), count them and drop them, without ever interacting with the vif. This set-up resulted in a rate of 950 Kp/s for maximum-sized packets (greater than line rate) but only 1.5 Mp/s out of a possible 14.8 Mp/s for minimum-sized ones.

At this point the remaining bottleneck was coming from the drivers' reliance on the Xen ring API, including their use of slow interrupts (known as events) to synchronize access to the packets. To push rates further up, we overhauled the Xen network backplane (figure 3), getting rid of the netback driver and replacing the software switch with a fast one based on VALE.

These modifications resulted in a significant performance improvement (figure 6). The graph shows, for various packet sizes, an increasing number of ClickOS vms residing on the same server and sending packets through a shared 10Gb NIC onto another server which measures the aggregate throughput. In this experiment we used all 8 CPU cores in our server, assigning 3 of them to the driver domain that hosts the ClickOS switch and assigning the remaining 5 cores to the vms in a round-robin fashion. As can be seen, our low-cost server can run as many as 128 ClickOS vms who in turn are able to fill up the 10Gb pipe for most packet sizes. In a separate test not shown due to space constraints, we confirmed that all vms were (roughly) equally contributing to this throughput. Further tests when receiving packets yielded line rate for all packet sizes of size 256 bytes and greater, and up to 4.8 Mp/s for minimum-sized ones.

4. RELATED WORK

In order to create a tiny system for network processing we could have relied on any number of existing minimalistic OSes [14, 28, 15], but they neither provide good device driver support nor the advantages of a fully virtualized system such as isolation and migration. Instead, we build ClickOS on top of MiniOS, a minimalistic OS aimed at creating small Xen-based vms.

Many virtualization technologies besides Xen exist [7, 25, 17]. We settled on Xen because it enables us to have excellent driver support (through its Linux-based dom0 domain and its split-driver model) while still allowing us to run a tiny virtual machine (a combination of MiniOS and Click). OpenVZ, for instance, does not support running an OS other than Linux. KVM supports MINIX, but the latest version is marked as crashing. The work in [24] introduces a new thin virtualization system aimed at increasing the overall security of a virtualized system.

Beyond minimalistic OSes and hypervisors, previous work has looked into optimizing the performance of Xen's data plane. One of the techniques consists of reducing the cost of packet copies by having the NIC directly place packets in guest memory [19, 22, 13]; we use this general concept in conjunction with the netmap framework [20] in order to speed up Xen's underlying network system. Some of the general optimization techniques used in the netmap framework and our netback and netfront drivers such as batching and polling previously appeared in other works such as RouteBricks [4].

Another optimization [19, 22] consists of making efficient use of memory grants (Xen's mechanism for allowing domains to share memory); we apply a similar approach to optimize MiniOS' netfront driver. The work in [9] optimizes Xen's scheduler and the Linux bridge that Xen relies on to direct packets to guest domains. We go one step beyond, entirely replacing the Linux bridge (or in the latest releases of Xen Open vSwitch [16]) with a modified version of the fast VALE switch [10]. One final technique available in Xen and other virtualization systems is passthrough, where a vm is given direct access to the NIC in order to improve networking performance. The problem is that this technique forces vms to host device drivers, binds the device to a single vm, and complicates vm migration.

There are a few other projects that have looked into creating small virtual machines. The Denali [27] isolation kernel was able to run large numbers of concurrent virtual machines, but had limited support for device drivers and guest OSes. The work in [12, 11] is similar to ours in that they also create tiny, Xen-based virtual machines, though their focus is on extending the Objective Caml language to generate different kinds of vms (e.g., they evaluate a vm with SQLite running in it) that are defined at compile-time. In-

stead, ClickOS can change configuration at run-time and the work further focuses on optimizing its boot times and networking performance. The work in [5] is similar, creating small, Erlang-based virtual machines on Xen.

With regards to commercial offerings, Cisco developed a single-tenant virtualized router that can run on VMware ESXi or Citrix XenServer [2]. However, it is not clear how extensible it is (it is based on IOS), how large its images are, how it performs nor how much it costs. Vyatta [26] has open-source software that can run on a number of virtualization platforms and that implements middlebox functionality. Unlike ClickOS, it is based on Debian, and so its images are large.

5. CONCLUSION

We presented ClickOS, a tiny, Xen-based virtual machine that can instantiate middlebox processing in milliseconds while achieving high performance, allowing for a truly programmable SDN data plane. Beyond the preliminary throughput experiments presented in the paper, we are in the process of optimizing and testing ClickOS' network performance when carrying out middlebox processing.

One of the main contributions of ClickOS is that it allows consolidation of very large number of vms in a single server: hundreds in our tests, and potentially even thousands since we can quickly put idle vms to sleep; contrast this with anecdotal evidence of only 10-30 vms in current deployments. We take the view that ClickOS might enable scenarios not possible today: per-customer firewalls, dynamically instantiated load balancers to cope with load, per-flow IDSes, and on-the-fly customizable software BRASes (Broadband Remote Access Server) providing on-demand premium services, to name a few. We believe ClickOS to be a step towards a much more dynamic, programmable SDN data plane.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] 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.

[2] 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.

[3] Click Modular Router. Click Elements. `http://read.cs.ucla.edu/click/click`, March 2013.

[4] 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.

[5] Erlang on Xen. Erlang on Xen. `http://erlangonxen.org/`, July 2012.

[6] 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.

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

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

[9] G. Liao, D. Guo, L. Bhuyan, and S. R. King. Software techniques to improve virtualized i/o performance on multi-core systems. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 161–170, New York, NY, USA, 2008. ACM.

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

[11] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. S. T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[12] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft. Turning down the lamp: software specialisation for the cloud. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[13] K. Mansley, G. Law, D. Riddoch, G. Barzini, N. Turton, and S. Pope. Getting 10 gb/s from xen: safe and fast device access from unprivileged domains. In *Proceedings of the 2007 conference on Parallel processing*, Euro-Par'07, pages 224–233, Berlin, Heidelberg, 2008. Springer-Verlag.

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

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

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

[17] OpenVZ. Welcome to OpenVZ Wiki. `http://wiki.openvz.org/Main_Page`, July 2012.

[18] 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 *Proc. ACM VEE, 2009*, VEE '09, 2009.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] U. Steinberg and B. Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

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

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

[27] 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.

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