Virtualizing the Data Plane Through Source Code Merging

Eric Keller
Princeton University, Princeton, NJ, USA
ekeller@princeton.edu

Evan Green
Princeton University, Princeton, NJ, USA
eagreen@princeton.edu

ABSTRACT

Virtualization is a key technology that enables multiple research groups to test new protocols simultaneously on the same physical network and also allows service providers to incrementally add new services. In this paper we focus on virtualization of the data plane, allowing for customized packet handling in each virtual network.

Much work has been done on virtualization technology. However, this has been focused on the user application experience or on a fixed networking stack. Rather than running custom data planes in user space or running separate guest operating systems, both of which come at a performance hit, we propose running a single kernel-level custom data-plane by synthesizing the configuration of the per-virtual-network data planes.

In this paper we present this idea using Click, where packet processing is specified as an interconnection of fixed networking tasks. We then demonstrate the idea using an unvirtualized Linux kernel as the target platform, showing how we provided isolation between the customized data plane.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design; C.2.6 [**Computer Communication Networks**]: Internetworking

General Terms

Design, Experimentation, Performance, Languages

Keywords

Virtualization, architecture, routing, virtual router

1. INTRODUCTION

Virtualization provides a means to run multiple virtual networks on a shared physical infrastructure. Each virtual network is logically separated and can be tuned to the specific needs of the applications running on it. Each application can run its own protocols and services, ideally, without disturbance from traffic on other virtual networks. This is a key technology for future experimental platforms and service deployment.

In this paper we focus on enabling the customization of data plane functionality. The challenge of this is the need to support (i) customization of the packet-handling functionality, (ii) packet forwarding at high rates, and (iii) multiple virtual networks running in parallel. Furthering point (iii), not only is it necessary to have a mechanism to share the physical machine, but it is required that each virtual network be isolated from one another. In this way, one experiment or service cannot interfere with the network of an unrelated experiment.

Existing virtualization techniques focus primarily on separation of execution environments, e.g. separating user environments through containers [17] or running separate guest operating systems [1]. However, this amount of flexibility and isolation comes at a price. First, the techniques are limited to software and therefore cannot be applied to specialized devices, such as FPGAs or network processors. Second, networking has higher demands, both in throughput and latency. Adding an extra layer, as virtualization technologies do, adds a significant amount of unnecessary overhead.

There has also been work on virtualizing fixed networking stacks [15] [5]. Here, each of the data structures in, for example, the Linux networking stack are duplicated and isolated in a container. This approach provides little in terms of flexibility as it is limited to what the fixed network stack provides, e.g. IPv4 forwarding.

These approaches sacrifice performance or flexibility in order to enable virtual data planes. Instead, we argue that networking is a specific task and does not require general virtualization to enable running multiple data planes on a shared resource.

We propose that when using a language where packet handling is specifies as a graph of common networking functions interconnected to indicate packet flow, source code based virtualization is an approach that does not concede either flexibility or performance. By this, we mean that the functionality of the data plane can be *specified* for each virtual network. Then at the source code level, the data planes can be combined. From the source code, important meaning can be inferred, including which networking specific operations are being performed and in what order.

Alone, this is not sufficient, but it does lower the barrier for providing an isolated environment for an unvirtualized resource such as the operating system kernel or programmable hardware. This is because we can inspect the

PRESTO'08, August 22, 2008, Seattle, Washington, USA.

source code and ensure that the virtual networks are not interfering with each other or doing anything illegal. Further, by using a language based on specifying a graph of common functions as the source code, we restrict functionality to what is needed in networking and therefore do not have to provide protection against general code from interfering with another virtual network or the entire system.

In this paper we present this idea using the Click modular router. Click is a software architecture for building flexible and configurable routers. It uses a textual language to specify how packets flow between a set of networking tasks, called elements. When Click is executed in the kernel using polling it achieves packet forwarding rates comparable to native Linux. Both [10] and the more recent [9] show that Click's IPv4 forwarding rate is actually greater than Linux, so it is reasonable to assume in kernel mode Click is high performance.

As a simple example to illustrate how virtualization can be simplified and kept lightweight, consider a situation where only a predefined set of existing Click elements are allowed to be used in the custom data planes. For this, it can be seen that virtualization can be reduced to simple resource accounting and ensuring the traffic goes through the elements for the correct virtual network. The virtualization layer only has to allocate each of the virtual networks a share of the physical resources and provide a run-time accounting mechanism.

We demonstrate the idea of source code based virtualizating by allowing multiple Click configurations to run in a single Click instance in the Linux kernel. First, we provide a coordinating process that combines each of the Click configurations. This coordinating process also performs a set of checks on each configuration, ensuring each configuration is legal. Second, we use Linux-VServer [17], a container based virtualization layer, to provide resource accounting and limiting.

The paper is organized as follows. In section 2 we discuss the architecture of the system and the general idea of source code based virtualization. In section 3 we discuss our prototype implementation based on the Click modular router and Linux-VServer. Then, in section 4 we discuss two challenges that arise in our implementation from executing in the Linux kernel. In particular, we discuss how these challenges relate to the issue of safety. We then discuss alternate techniques in section 5. We wrap up with a discussion of future work in section 6 and conclusions in section 7.

2. LIGHTWEIGHT VIRTUALIZATION

In this paper we argue that a lightweight mechanism, combining source code, can be used for supporting virtual networking. In particular, enabling multiple concurrent customized data planes to run together on a shared platform.

From the virtual network's perspective a single node in its network would appear as shown in Figure 1. This includes a user environment to run control software, a forwarding path for processing packets, and an interface between the control and forwarding for configuration. As the physical node is shared, to achieve isolation, each virtual network is allocated a share of the physical resources. This includes, for example, CPU cycles, memory, and bandwidth.

The packet processing is expressed using a language that specifies a graph of common networking functions interconnected to indicate packet flow. This graph specifies the complete packet processing functionality from input to output for that particular virtual network.

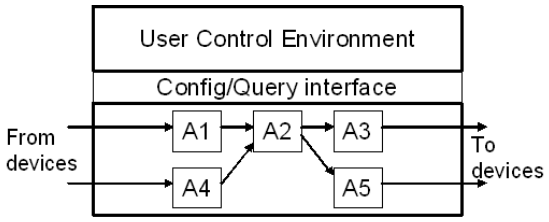

Figure 1: View of a single virtual network device.

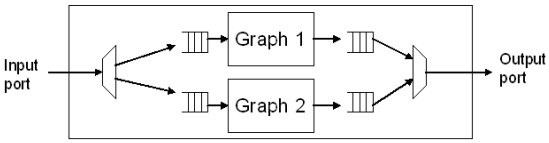

Figure 2: High-level view of a combination of two custom data planes

Considering this setup, the idea behind source code based virtualization is relatively simple. There are multiple virtual networks that are to be run on a single physical machine. Each virtual network's data plane functionality is specified using a language representing the graph. The graph for each virtual network is given to a central controlling process which then combines the graphs into a single graph.

Simply combining the graphs is not sufficient. Additional processing needs to be added by the coordinating process to direct packets from input of the master graph (i.e. a physical interface) to the input of the correct sub-graph. A high-level view of this can be seen in Figure 2 which shows the combination of two custom data planes from two virtual networks.

As each of the virtual networks have been allocated a particular share of resources, some run-time resource accounting and limiting is also needed. This is to ensure that one virtual network does not consume excess resources and exert influence on the performance of other virtual network. Each virtual network is allocated a share of the resources, such as CPU cycles, for the particular physical machine. This allocation is then used by a scheduler to determine whether the code for a given sub-graph can be run.

In this paper we assume that there exists a library of common packet processing functions that can be used. Each of the functions in this library would be considered safe. Meaning that it is bounded in terms of resource usage and does not access or corrupt memory that is not its own. Ideally, the collection of elements is complete in that new protocols can be built exclusively from this library. However, there will be cases when custom functionality will be needed. We discuss the problems related to this as it pertains to our particular implementation in Section 4.

Using source code based virtualization, the overhead is minimized by ensuring isolation at compile time where possible. The extra overhead comes from (i) sharing the network interfaces and (ii) performing the resource accounting, both of which can be minimal and will be needed for any virtualization solution. There are proposals to add functionality

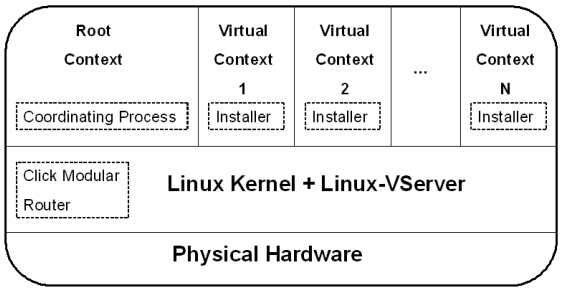

Figure 3: System Architecture

for performing the mux/demux functionality directly on the network interface card [19], removing that overhead completely.

3. PROTOTYPE IMPLEMENTATION

To realize the architecture discussed in Section 2 we use the Click modular router and Linux-VServer, as shown in Figure 3. Linux-VServer is realized as a patch to the Linux kernel, and forms our kernel environment. Running on top of the shared kernel are several contexts which corresponds to a collection of user space processes grouped together to provide namespace isolation and collectively have their resource usage accounted for. The user's working environment, which corresponds to the management interface of the virtual network device, is the user space context that Linux-VServer provides. In this use, Linux-VServer is providing the role of isolation of the user environments.

The rest of this section discusses the isolation of the custom, per virtual network, data planes. In Section 3.1 we discuss how we combine multiple Click configurations. Then, in Section 3.2 we discuss how we used Linux-VServer to provide us with the resource accounting that we require as part of the run-time management that source based virtualization needs.

3.1 Combining Graphs of Click Elements

We argue that source based virtualization is possible given a language which can specify packet processing functionality as a graph of common functions. For this, we chose to use Click [10]. Click is widely used to build custom data planes, both for researchers doing experiments [8][4] and for building real systems (e.g., deploying software routers and services) [13]. This makes it a natural choice for network virtualization.

In Click, a router is assembled by connecting together basic packet processing modules called elements. Elements include common processing tasks such as classification, lookup, and queueing. As an example, the following configuration describes a system that reads a packet from device eth1, passes the packet to the EtherMirror element which swaps the source and destination MAC addresses, puts the packet in a Queue, and then sends the packet out device eth1.

```
FromDevice(eth1) -> EtherMirror
                 -> Queue
                 -> ToDevice(eth1);
```

In our system, each virtual network can have a Click configuration loaded into the kernel. However, this is not immediately possible with the original implementation of Click. While it would be possible to modify the Click kernel module to accept multiple configurations through system calls, it is unnecessary. Instead we chose the less complex solution of adding an extra layer on top of Click, running in user space, that combines multiple graphs into a single master Click configuration.

To call the coordinating process and install a graph, an installer program running in each user context replaces the standard "click-install" program, as shown in Figure 3. It installs the Click configuration in the kernel using socket based communication with the coordinating process, passing the Click configuration for that virtual network. The coordinating process will then install the configuration.

In addition to combining multiple Click configurations into one, the coordinating process will also add elements to demultiplex and multiplex the traffic. In our setup, we assume physical interfaces can be shared. To enable classifying packets according to which virtual network it belongs to, we make use of generic routing encapsulation (GRE) tunnels. A GRE header is composed of a source and destination IP address along with a type field. The type indicates the inner packet type, for example IP, and the destination address is used for classification in our system. A system configuration file specifies the source and destination IP address of the GRE tunnels, the physical devices used, the virtual devices assigned to each virtual network, and the internal IP address of each of the virtual devices. From this, the coordinating process can add elements to the master Click configuration for the purpose of multiplexing and demultiplexing.

Referring back to Figure 2, showing the combined master graph, each sub-graph shown is a Click configuration in our implementation. On the ingress side, the master Click configuration will use classification based on the GRE header to direct the packet toward the correct portion of the graph. After classifying which virtual network a packet belongs in, elements the coordinating process added to the master Click graph will strip the GRE header and deliver the packet to the virtual network's Click elements.

At the egress side, packets from each of the virtual network's set of elements are encapsulated in a GRE header and multiplexed onto the physical devices. It is envisioned that traffic shaping elements could be added to the master Click configuration as a means of resource accounting the allocated bandwidth.

In addition to its basic responsibility of installing individual Click configurations, the coordinating process also provides some form of isolation. Click is a flexible language and as such can be used to specify a wide variety of configurations. The coordinating process verifies the validity of the Click configuration. It will ensure that the devices the virtual network is accessing are the virtual devices actually assigned to it. The naming of devices can be shared (e.g. two virtual networks can have eth1), but virtual networks cannot access devices that are not assigned to it (e.g. eth3, if it was not assigned an eth3). Through the use of the multiplexing/demultiplexing elements, previously discussed, the coordinating process performs the mapping between the virtual network's devices and the physical devices. Further checks may become necessary in the future, and having this coordinating process will enable these checks to be easily added later.

3.2 Resource Accounting with Linux-VServer

The basic mechanism for isolation of the actual packet processing is done through resource accounting. Through resource accounting each user context can be given guarantees on the amount of a particular resource, such as CPU cycles or memory, that it will receive. This enables a virtual network to be unaffected by the configuration of other virtual networks. Linux-VServer uses a token bucket extension to the Linux scheduler to count CPU cycles and limit each thread's ability to run if its context does not have enough tokens. The challenge here is to associate specific elements with a particular user context.

To achieve this, the Click configuration for each virtual network is run in a separate kernel thread. In Click, packet processing is essentially a series of function calls passing a packet down the pipeline. Upon hitting an element such as a Queue, the function calls return. It is impossible to associate an individual element with a particular thread. Instead, Click has particular elements that are considered schedulable. Such elements run at scheduled times and are the start of a series of function calls. Through the StaticThreadSched element available in Click, these elements can be assigned to a particular thread. The thread consists of the starting element plus the entire pipeline of elements up to the next Queue element.

Each of these threads now handles a single virtual network's Click configuration. As such, resources such as CPU cycles and memory used by this thread need to get accounted for and billed to each particular virtual network. To achieve this we created an element called VSStaticThreadSched. Similar to the StaticThreadSched element that assigns schedulable elements to threads, the VSStaticThreadSched element assigns a thread to a user context. The configuration string it takes in is a list of pairs consisting of the thread and context IDs.

Through the mechanism we provide, we are able to combine multiple Click configurations into a single graph, associating each of the original graphs with a particular virtual context. This allows Linux-VServer to account and control the resources each context can use to forward packets. Additionally, this accounting is system wide, meaning it is a single mechanism covering both user-space control and kernel mode packet forwarding.

4. KERNEL EXECUTION CHALLENGES

The ability to execute packet forwarding in kernel mode is beneficial given that packet forwarding is between six and ten times faster when compared to packet forwarding in user space. However, this capability does not come without challenges. In this section we discuss two issues that arise with execution in the kernel: unyielding threads and pointers. We then discuss the challenge of verifying the safety of elements, in particular as it relates to these two issues.

4.1 Unyielding Threads

The first issue found with executing in the kernel is that of unyielding threads. The Linux kernel is a cooperative environment where kernel threads yield to one another or where execution changes when a kernel thread blocks or returns from a system call. In our implementation we implemented each virtual network's configuration in its own kernel thread, with each thread allowed to execute a certain number of tasks before yielding. Here, a task corresponds to the execution of one or more elements, roughly a pipeline of elements between queues. A single long running task can result in both a short term disruption as well as possibly leading to long term unfairness.

The short term disruption comes from the simple fact that when sharing a single processor, one thread executing means another thread is not. Therefore a single long task for one virtual network can cause extra delay for the other virtual networks. However, this problem can be solved. From the guarantees made by the platform to the virtual networks in terms of minimum delay and jitter, a maximum allowable execution time for a single thread can be calculated. Also, as the elements in the provided library can be profiled to determine a bounded execution time, the execution time of each task can be calculated. From this, if a task is determined to be too long, the pipeline of elements forming that task can be broken up into two tasks using a Queue element. However, when an element in the task does not have known characteristics, e.g. a custom element, then we propose executing that portion of the pipeline in user space inside of a container, isolating it from the rest of the system. This solution also holds for the situation where a single element has a known long execution time and the pipeline cannot be broken up enough. In this case, the performance degradation from executing in user space is not as big of a concern given the forwarding rate with that element will be low anyway. An area for future consideration is to (i) determine a set of elements that form a complete library and (ii) determine a process to enable custom elements to be fully characterized so they can be used in kernel mode.

There are also long term issues with unyielding threads in our particular implementation. In particular, Linux-VServer makes use of a token bucket scheduler that adds tokens at a particular rate and consumes them when running. However, the token bucked does not go negative. Therefore, the time from when a particular context runs out of tokens until when it yields is not accounted for. This can lead to the context getting a share of the CPU in excess of its allotment. The solution to this is the same as the short term disruption. In particular, calculating the length of a particular task and either breaking it up if the execution time can be determined or executing in user space if it cannot.

4.2 Pointers

A second issue with executing in the kernel comes from the fact that custom elements are written in C++, a language with pointers. Click is a shared environment and because of this there are global variables accessible to elements that can affect the behavior of the entire router. For example, the router configuration is made globally available, which would allow any element to see the entire configuration, including the configuration of other virtual networks. While some system elements may require the use of these variables, in general they should not be accessed as it would enable a virtual network to interfere with, or observe, another virtual network.

In addition to the global state in Click, there is the system wide state made available by Linux. Kernel modules are given access to internal state and functions that affect the entire machine. As elements are run in the kernel, this is a concern.

One possible solution can involve compiler tricks. Here, before allowing an element to be added for use, the element

could be (1) compiled in user mode and (2) compiled using restricted versions of various header files. Compiling in user mode ensures that new Click elements are not calling kernel functions or accessing kernel data structures. Compiling with restricted header files ensures that Click global state cannot be accessed.

However, since the elements are written in C++, this approach is not sufficient. Pointers can allow elements to access and modify memory that belongs to other applications. Because this code is running in kernel mode, pointers allow an element to potentially corrupt data of another user, or damage the entire machine. Because of this, any custom elements are to be run within a container in user space to fully isolate it from the rest of the system.

Ideally, the elements that end up being executed in user space are not on the fast path of the network device. However, it is possible that they will be. As with the issue of unyielding thread, future work will be needed to allow custom elements to run in kernel mode.

4.3 Verifying Safety of Elements

While considering both of the issues discussed in the preceding sections, it is important to note that safety guarantees can be maintained. First, in situations where safety has the potential of being compromised, we trade the performance of executing in the kernel for the isolation of executing in a container in user space.

However, a goal is to make this an uncommon situation or one that does not hurt the system performance. This is where the use of a safe library becomes important. When using elements from the library, the execution can occur in the kernel and safety guarantees can be maintained. Elements in the library would have bounded execution time, leading to the ability to verify that a given task would yield in the maximum allowable amount of time. Path analysis with bounds on loops can be used in conjunction with models of the architecture to determine estimates of execution time [12]. In addition to executing in a bounded amount of time, elements in the library would also have been well tested to ensure that they are not accessing memory that is not their own or have any other bugs that can affect the system. Static analysis has been shown to be able to find many bugs in complex systems [7].

Currently, Click has a library of approximately 450 elements, some of which are commonly used and well behaved and many of which are experimental or not for packet processing in the kernel. As a prototype in our environment, this is sufficient. However, for real use, a more thorough analysis of the library will be needed. Additionally, while the current library of elements may not be complete in terms of being able to specify any protocol, it does include many common functions such as queueing and lookup. Additional elements that are parameterizable can be added to fill the gaps and make the library more complete.

5. RELATED WORK

There are other approaches to virtual networking in the context of the data plane. Three high level considerations for each of these are (i) flexibility, (ii) performance, (iii) isolation.

In [2], the authors present the VINI system, a virtual network infrastructure that allows experimentation in a realistic environment and with a high degree of control over network conditions. In an experiment the authors ran, Click was used as the forwarding plane within the user space environment of Linux-VServer, so therefore it suffers from poor performance due to being restricted to running in user-space.

OpenVZ [15] is a container based virtualization technology, meaning the kernel is shared and the user space environment is isolated. OpenVZ achieves good performance by using a virtualized networking stack allowing packets to be forwarded in the shared kernel. Through the use of NetNS [5], Linux-VServer has a similar mechanism as demonstrated with Trellis [3]. However, both of these systems are limited to the functionality provided by the Linux networking stack, namely IPv4 forwarding.

Another approach is to use a full or para virtual machine, such as Xen [1], and run Click in kernel mode in the guest operating system. This has the advantage of supporting custom data planes. Performance of this has not been evaluated, but it has been shown that using the guest operating system (domU) for forwarding in Xen is significantly slower (over 6x worse) than when doing the forwarding in the shared dom0 [6]. Doing the forwarding in dom0 is essentially the same as using OpenVZ, as this is then limited to the fixed forwarding capabilities of Linux. So it can be assumed that running Click in this environment would not be efficient. This needs to be evaluated further.

While originally meant as a way to protect a system against faulty drivers, the Nooks system [18] could be used as an approach to provide general virtualization of the kernel. For this, Click would need to be modified to be run as multiple kernel modules, and each module is a single Click configuration. The Nooks system would provide protection, but does not do resource accounting. It is something that should be looked into more.

Scout [14] is a communication oriented operating system that introduced the notion of the path abstraction. A path specifies the flow of data through communication modules from source I/O to sink I/O as well as the resources used. The Scout operation system then schedules based on paths, rather than threads. This provided an inspiration for the high level architecture of our system which has similar constructs. However, they were creating a specialized operating system customized for a given network devices, as opposed to sharing a general platform across multiple custom (virtual) network devices.

6. FUTURE WORK

There are two main directions for future work. First, there is more to be done on the issue of safety. In particular we have assumed an existence of a library of common function that can be used to construct custom data planes. Further work is needed to determine what would constitute a complete library, where custom elements will not be needed in most cases. To cover the situations when custom elements are needed, we intend to investigate ways to either automatically determine if an element is safe or monitor the element at run time through, for example, automatically adding extra checks into the code.

A second direction is to provide support for specialized devices, in particular we plan to look at FPGAs. As previous work has shown[11][16], it is possible to map Click configurations to FPGAs. We plan to investigate the issues with this in the context of source code based virtualization. In software, we talk about resource accounting in terms of

CPU cycles. In FPGAs, execution is parallel, so chip area is the limiting resource. Both environments require management over memory. There are additional complications as well. For example, "hot-swapping" a portion (one virtual network) of the master Click configuration is not trivial to do in an FPGA whereas in software it is trivial. Extending into architectures beyond software based routers has the potential for great performance benefit.

7. CONCLUSIONS

In this paper we presented our architecture for sharing kernel-mode Click as an example of using a source code based virtualization. Here, the source code is a language where packet handling is specifies as a graph of common networking functions interconnected to indicate packet flow. By using this language, allowing users to create custom data planes for programmable virtual networks can be very lightweight. The overhead includes the muxing/demuxing needed for sharing the network interfaces as well as performing resource accounting. Using Linux-VServer for resource accounting allowed for a system wide accounting mechanism to be used. By associating a thread with a virtual context, the CPU usage for packets traveling through the user's Click configuration can be counted against that users context.

We also discussed two challenges that arose from our implementation of executing in the Linux kernel and how they pertain to safety. In particular we discussed the problems of unyielding threads and usage of pointers and discussed potential solutions.

By using a language that specifies a graph of common functions, we are able to simplify the task of providing a virtual environment for the data plane and therefore make it lightweight. It is due to the unique nature of networking where the problem of virtualization can be constrained to enable simplified solutions. Through virtualization of the data plane, innovation in the network can occur.

8. ACKNOWLEDGMENTS

We would like to thank Michael Freedman, Jennifer Rexford, Andy Bavier and Sapan Bhatia for their advice and help throughout our work on this research.

9. 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[2] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and controlled network experimentation. In *Proc. ACM SIGCOMM*, Sept. 2006.

[3] S. Bhatia, M. Motiwala, W. Muhlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, , and J. Rexford. Hosting virtual networks on commodity hardware. Georgia Tech Computer Science Technical Report GT-CS-07-10, January 2008.

[4] J. Bicket, D. Aguayo, S. Biswas, and R. Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *MobiCom '05: Proceedings of the 11th annual international conference on Mobile computing and networking*, pages 31–42, New York, NY, USA, 2005. ACM.

[5] E. Biederman. Netns. https://lists.linux-foundation.org/pipermail/containers/2007-September/007097.html.

[6] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, L. Mathy, and T. Schooley. Evaluating xen for virtual routers. In *International Workshop on Performance Modelling and Evaluation in Computer and Telecommunications Networks (PMECT)*, 2007.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001.

[8] M. Handley, O. Hodson, and E. Kohler. XORP: An Open Platform for Network Research. In *First Workshop on Hot Topics in Networks*, Oct 2002.

[9] F. Huici. Measuring click's forwarding performance. Internal Technical Report University College London, September 2005.

[10] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[11] C. Kulkarni, G. Brebner, and G. Schelle. Mapping a domain specific language to a platform fpga. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 924–927, 2004.

[12] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.

[13] Mazu Networks, Cambridge, MA. *Mazu Profiler Datasheet*, 2007.

[14] A. B. Montz, D. Mosberger, S. W. O'Mally, L. L. Peterson, and T. A. Proebsting. Scout: a communications-oriented operating system. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, 1995.

[15] OpenVZ. *http://openvz.org*, 2007.

[16] G. Schelle and D. Grunwald. Cusp: a modular framework for high speed network applications on fpgas. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 246–257, 2005.

[17] S. Soltesz, H. Pötzl, M. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of EuroSys 2007*, Lisbon, Portugal, March 2007.

[18] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: an architecture for reliable device drivers. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 102–107, 2002.

[19] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.