Multi-Level Architecture for Data Plane Virtualization
COS461 Project Report

Eric Keller
ekeller@princeton.edu
Princeton University

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 custom data-plane by synthesizing the configuration of the per-virtual-network data planes. Then enable trade-offs between performance and isolation by partitioning the functionality of the custom data-plane across multiple targets.

In this paper we present this idea using Click, where packet processing is specified as an interconnection of fixed networking tasks. First, we target execution in an unvirtualized Linux kernel as the target platform, showing how we provided isolation between the customized data plane. Then, for added isolation, we execute inside of a container running in user-space. Finally, for increased performance and to demonstrate extending beyond software routers, we target execution on an FPGA.

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 [20] 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 [18] [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 specified 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 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 [12] and the more recent [11] 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 [20], a container based virtualization layer, to provide resource accounting and limiting.

We then demonstrate that source code based virtualization extends beyond "software routers" by providing an implementation of Click that can run on an FPGA. In this context, the coordinating process also provides the role of partitioning a single graph across multiple targets. In particular, targeting the kernel for light weight software processing, a container for increased isolation, and FPGA for increased performance.

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. Following that, in section 5 we discuss extending beyond commodity hardware, by discussing the integration of FPGA into Click. We then discuss alternate techniques in section 7. We wrap up with a discussion of future work in section 8 and conclusions in section 9.

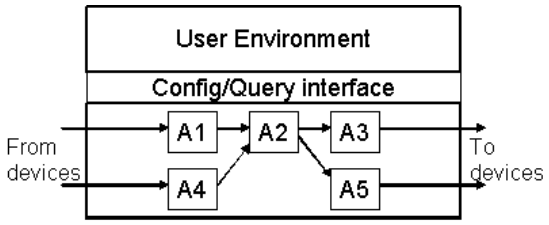

Figure 1: View of a single virtual network device.

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.

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.

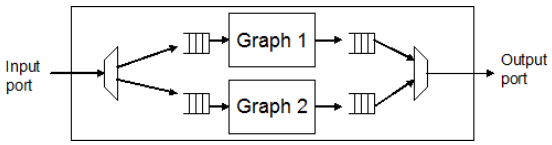

Figure 2: High-level view of a combination of two custom data planes

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 subgraph. 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 subgraph 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 for performing the mux/demux functionality directly on the network interface card [25], 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.

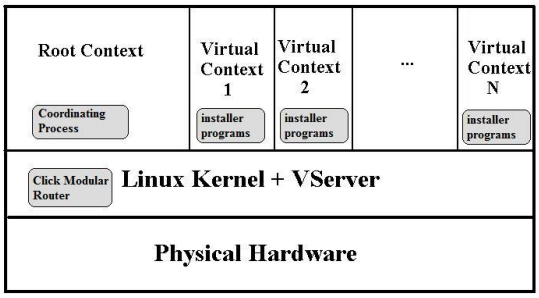

Figure 3: System Architecture

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 [12]. Click is widely used to build custom data planes, both for researchers doing experiments [9][4] and for building real systems (e.g., deploying software routers and services) [16]. 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, an element that 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 unyielding threads in out 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 [14]. 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 others which are experimental. 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 Extending Beyond Commodity Hardware

The previous sections focused on the issues concerning software routers. Namely, they discussed execution in kernel mode where possible and in a container in user space when not possible. This section then discusses moving the platform beyond commodity hardware. That is, moving beyond software routers.

There are many possible architectures that can be used to build network nodes. One example is a commodity hardware platform with a programmable network interface card (NIC). Here, the packet processing functionality of the NIC is capable of being modified since it is implemented using a specialized programmable device such as an FPGA or network processor. Another example is a pool of processing architecture, as proposed by Turner [22], where a number of processing elements of type general purpose processor, FPGA, or network processor are connected to each other and to interface cards via a crossbar switch. In either architecture, a data plane specified as a graph is partitioned across multiple processing elements. In the programmable NIC case, partitioning is across an FPGA and general purpose processor. The pool architecture is more general where partitioning can be across any number and combination of general purpose processors, FPGA, and network processors.

To support part of the graph representing the data plane functionality being mapped onto an FPGA, support for FPGAs was integrated into Click. Listed below are important considerations for the tool. Each will be described in more detail in following sections.

- **Connection to external environment:** what, for example, does eth1 mean when using FromDevice(eth1) and how does that get implemented on an FPGA?
- **Packet transfer:** how are packets transferred between elements?
- **Element specification and implementation:** how is the meta-data about the element described and how is the packet proccessing code described?
- **Run-time querying and configuration:** how can a software program interact with elements to get and set state, such as listing and adding/removing routing table entries.
- **Memory:** how can elements store internal state or buffer packets?

Before describing the components of the tool, it is useful to describe the use of the tool from the user's perspective. First the user runs the *click-fpga* tool. On the command line, the user specifies a board to use, e.g. NetFPGA[15], and the Click configuration file. The tool will create a new directory with all of the files needed to compile the design to an FPGA. This includes source files (e.g. Verilog) for each element, a top-level Verilog file, a constraints file describing the pins on the FPGA,

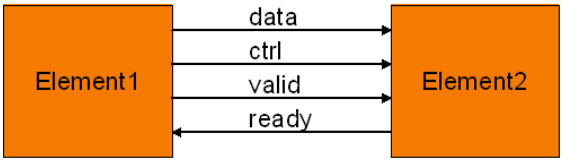

Figure 4: Connectivity between two elements using a common interface for packet transfer

and a script which calls each of the FPGA design tools, e.g. synthesize and place and route. This scripted functionality can be integrated into the *click-fpga* tool, but was kept separate for now.

Once a bitstream[1] has been created and downloaded to the FPGA, the user runs *click-fpga* again, but this time with a command line option *-r*, which indicates that the tool is the run-time program. Users can then telnet into the Click program and query/configure the elements.

5.1 Packet Transfer

The first aspect to consider is how packets are transferred between elements. In software, packets are passed between elements through function calls with a pointer to the packet as the argument. For Click in the FPGA, the Click configuration is implemented using a pipeline of the elements. Each of the elements will have a Verilog source and a pipeline is created by connecting the elements together using a common interface, as shown in Figure 4. Then, using the common interface, packets are streamed from one element to the next, word-by-word. While it is possible to pass a pointer in between modules, this would then require accessing the data through a shared bus. Instead, by streaming the packet, we are fully utilizing the capabilities of the FPGA making the processing more efficient.

5.2 Connection to external environment

Once the inter-element connectivity is considered, the next aspect of the tool is how a Click configuration is connected up to the external environment. That is, when an element, such as FromDevice, makes use of a device, such as eth1, that element is dependent on the execution environment.

In software, the operating system provides an interface to achieve this. For the FPGA, a description of the board is needed. This board description includes a list of available devices. For each of the devices, the board description includes a description of the extra logic needed (e.g. using an Ethernet device requires an Ethernet media access controller) and the pins of the FPGA that the device connects to. Also for each of the devices, there is a description of the ports available to use by the Click element. A port is a grouping of singles for the particular device. Using Ethernet as an example again, one port would be the ingress port that the FromDevice element would use, and the other port would be the egress port that the ToDevice element would use.

Using this description, the *click-fpga* tool can generate the system. It is important to note that by using this board description, the Click configuration becomes portable across boards, so long as each board has the ports that the configuration uses.

5.3 Element specification and instantiation

The *click-fpga* tool generates all of the necessary source to generate a bitstream for the FPGA. This includes Verilog describing each element. To accomplish this, the *click-fpga* tool requires (i) meta-data describing the element (e.g. how many ports), (ii) an interface to parameterize the element, and (iii) a mechanism to generate the Verilog for each element.

The *click-fpga* tool makes use of much of the existing infrastructure in the Click software. With Click as already exists, new (software) elements can be created by implementing a class in C++ that extends a base class called *Element*. The element then, through implementing various functions, specifies the meta-data, handles the parameterization, and specifies how the packet is processed.

Therefore, to satisfy the three requirements *click-fpga* has, only a small modification to Click's Element class was needed. The meta-data and parameterizing of the element are done using the existing Click mechanism. However, instead of implementing the function to handle the processing of the packet, an element targeting an FPGA implementation implements a function, *elaborate()*, that will generate the Verilog for that element. At the simplest level, this could simply be a single print statement printing an entire Verilog file. However, given the implementation in C++ and given the flexible parameterization that Click's configuration strings provide, more complex generation could exist. That is, the *elaborate()* function could specialize the generated Verilog based on how the element was parameterized.

5.4 Run-time querying and configuration

In addition to supporting the creating of a design in an FPGA, the *click-fpga* tool also provides run-time capabilities as well. In particular, it provides a mechanism to query and configure the state of elements at run-time, for

[1] A bitstream is the file which is downloaded to the FPGA to configure it.

example listing and adding routing table entries. Click has support for this in software through the use of the element's *handlers*. The *click-fpga* tool, again, makes use of the existing infrastructure and extends it slightly to support execution on an FPGA.

Handler's in Click are implemented as call back functions. Using an API provided by Click, an element can specify the name of a handler and which function to call when a user invokes that handler at run-time. With *click-fpga*, this mechanism was kept. However, instead of the call-back function returning the value of a variable in software it will get the value from the element on the FPGA.

Before considering what is required from the tool, it is useful to discuss how a request would be handled. As shown in Figure 5, a user would connect to the Click process using socket based communication (e.g. using telnet). The user would enter the command "READ elem.handler" where elem is the element name and handler is the handler name. Click will then call the call-back function specified for that particular handler. The handler will then make a request to read from the FPGA at a particular offset, where the offset is relative to zero for all elements. The offset will be translated to an absolute address based on the base address of the element, which is assigned by the tool at instantiation time. A char driver is used to pass the request to the FPGA, where the address is decoded and the request only passed to the correct element using the relative offset. The element, which specified Verilog code to handle requests, then returns the requested value.

From this, it can be seen that what is required is (i) a way for a board to specify how much address space is available, (ii) the ability for each element to request a block of address space, (iii) an interface for an element to make a read or write request at run-time, and (iv) a decoding mechanism to translate between absolute and relative addresses. The first three are provided within the *click-fpga* tool through interfaces in the *Board* and *Element* classes. For the decoding mechanism, in addition to interfaces in the *Element* class, the *click-fpga* tool will automatically generate the Verilog for the decoding logic. The base offset of each element is assigned to minimize the area the decoding logic will take. In particular it tries to minimize the number of bits of comparison that are needed. For example, if there are two element, each of which use less than half of the available address space, only a single bit is needed for decoding. On the other hand, if one of the two elements requires more than half of the address space, then additional bits will be needed.

Through the use of the handler functionality in Click and through the decoding mechanism, a run-time querying and configuring capability is part of *click-fpga*.

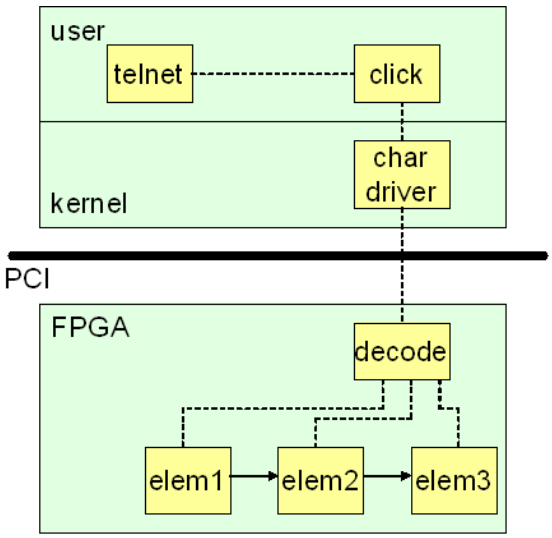

Figure 5: Handling run-time querying and configuration of elements on an FPGA

5.5 Memory

Memory is a key component of network systems. It is used to store data structures such as tables as well as provides storage for packets. As such, the design and use of memory is important. FPGAs have several options to consider, each listed below.

memory type	**notes**
flip-flops	used for single variables
distributed RAM	made from lookup tables, good for register files
block RAM	on-chip dual ported memories, 16Kbits in size, hundreds of BRAMs on bigger FPGAs
SRAM	off-chip memory with low access times (1 or 2 cycles)
DRAM	off-chip memory with long access times

Given these resources, Click elements then have the ability to use any of the on-chip memories within the element itself. External memories are currently treated as external devices. This means that an element that needs to use an external memory must request to use it and is granted exclusive access to a port of the memory controller[2]. This is not the final implementation as shared memory needs to be an option.

[2]the memory controller can be multiple ports, with a single port to memory

6 Results

To evaluate the platform, there are three main questions to consider. Each will be considered in subsequent subsections.

- Is multi-level the right approach?
- What is the overhead of virtualization?
- Are the virtual networks isolated in terms of resource usage?

6.1 Multi-Level

The main goal is to enable multiple custom data planes to run concurrently on a shared platform. This could be done in user-space making use of virtual machine technology, such as with Xen or Linux-VServer. We argue that this approach is not high performance enough. Additionally, we argue that specialized devices are integral parts of architectures for virtual networks. Given this, the first experiment was to determine the raw forwarding capability of the same design, IPv4 forwarding, on each of the target platforms - Click in user-space, Click in kernel mode, and Click on the FPGA.

To this, we created a five node star topology using the Emulab [24] testbed, as shown in figure 6. Each machine has a 3GHz processors, 2 GB of RAM, and 1 gigabit ethernet connectivity between each node and the router node. The router node in the software tests has four gigabit ethernet network cards. For the FPGA test, the router node has a NetFPGA card running a reference router developed by students at Stanford. Node n0 generates 64 byte packets at a specified rate, tagging each packet with the current time with nanosecond resolution. Each packet send by node n0 is destined to node n1. Node n1 then modifies the header to make the packet destined for n2, which modifies the header to be destined for n3, which then modifies the header to be destined for n0. Once the packet goes around from n0 to n1 to n2 to n3, going through the router each time, the packet is received by n0. Node n0 will subtract the time stored in the packet from the current time to get the delay.

To determine the peak rate that the router could handle, the send rate was increased until the packet loss reached a 0.1% loss rate. The results are shown in Figure 7. As can be seen, the forwarding rate for Click in the kernel is substantially higher than in user space and the FPGA is substantially greater than kernel[3]. This shows that the extra effort to support the kernel and FPGA are worth it.

[3]The FPGA results were limited by the maximum send rate that n0 could achieve in software, not by packet loss.

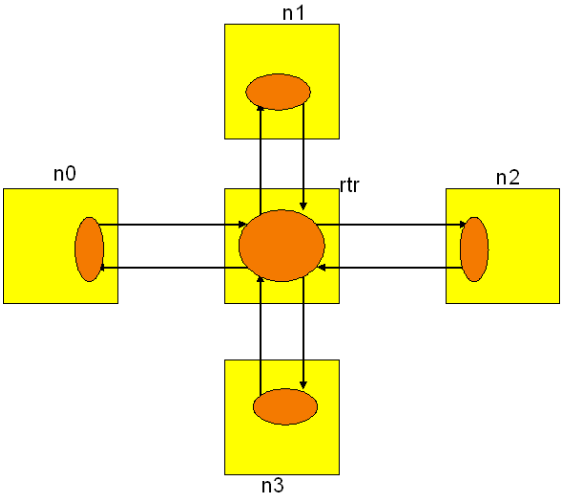

Figure 6: Five node star topology used for tests on Emulab

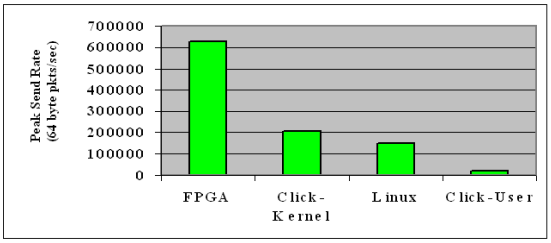

Figure 7: Performance comparison of each target platform

A second point to notice in Figure 7 is the comparison of Linux to Click in the kernel. While this is not a direct comparison, as Linux is a more complete and complex IPv4 forwarding solution, it does show that Click is at least comparable in performance. Therefore, using a modular solution, such as Click does not sacrifice performance as compared to a custom approach.

6.2 Overhead of Virtualization

The next issue to look at is what the overhead of virtualization is. First, in the case when going to user-space is necessary, there is overhead of running inside of a container (Linux-VServer) as opposed to just user space. Using the same setup on Emulab with a five node star topology, the IPv4 router running in user space and running inside of a container are shown in Figure 8. There is some overhead of executing inside of a container, but it is minimal.

The next source of overhead to look at is the act of us-

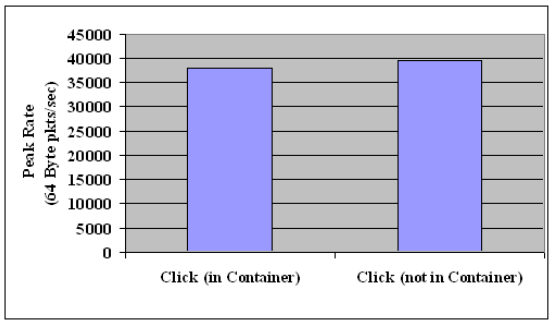

Figure 8: Overhead of running in a container

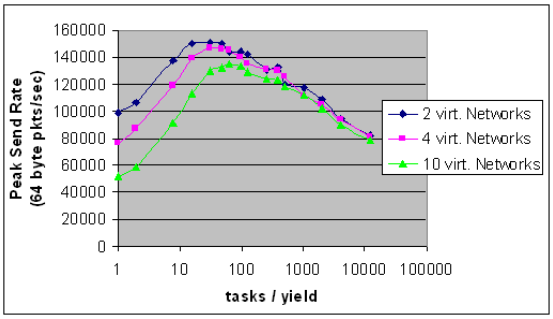

Figure 9: Analysis of finding the optimal number of tasks to run before yielding

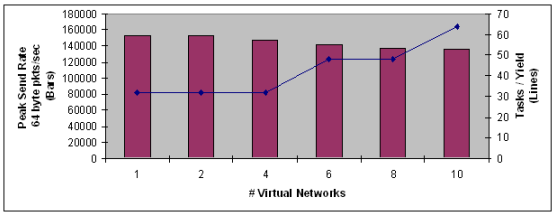

Figure 10: Aggregatge forwarding rate for each number of virtual routers given the optimal number of tasks to run before yielding

ing multiple threads. One key design decision was to run each virtual router's data plane in its own thread. To then examine the effect of this decision, we use a similar setup on Emulab with a five node star topology. Running on the router node was multiple virtual router data planes, each doing IPv4 forwarding. Traffic was distributed to each virutal router in a round robin manner. Therefore, there are N+1 threads running; one thread for each of the virtual routers, and one thread performing the I/O functionality. As each of the virtual routers are performing the same functionality, the aggregate throughput should stay constant as more virtual routers are added.

However, before considering that, there is the key question of how long each thread should be allowed to run before yielding. To determine this, the number of tasks each thread was allowed to run was varied for a given number of virtual routers. In our test, a task corresponds to forwarding a single packet in the virtual router threads and receiving or transmitting eight packets with the network card in the I/O thread. For each configuration, the peak forwarding rate was determined using the same metric as previous tests - the highest rate node n0 can send at with less than a 0.1% packet loss rate. As can be seen in Figure 9, there is an optimial value for the number of tasks to run before yielding. When the number of tasks run before yielding is low, there is high context switching overhead. When the number of tasks run before yeilding is high, the I/O thread gets starved and packets get dropped on the network card.

It can also be seen that the optimal value is different for different numbers of virtual routers. This comes from the fact that as there are more threads, the I/O thread gets swapped in less often and therefore needs to run for longer.

The optimal value of tasks to run, for between one and ten virtual routers, is shown in Figure 10 as the line in the graph. Then, using the optimal value of number of tasks to run before yielding, the peak rate is plotted with the bars. It can be seen that there is a slight downward trend, representing the overhead of more virtual routers sharing the same CPU. However, there is not a significant drop in forwarding rate from one virtual router to ten virtual routers, implying the overhead is minimal.

6.3 Resource Isolation

There final question to examine relates to the effectiveness of the resource isolation mechanism that was used, namely using Linux-VServer. In particular there are the issues related to the short term disruption due to a thread not yielding and the long term unfairness when a thread does not yield.

A short term disruption can be caused by a single long running task. Here, a task is a pipeline of elements, so the finest granularity is a single element, as the pipeline can be automatically broken up using queues between elements. We profile a few elements in the Click library to quantify this, finding the average number of cycles each takes to execute. In particular, some examples include Counter (700 cycles), CheckLength(400 cycles), HashSwitch(450 cycles) and RadixIPLookup with 167,000 table entries (1000 cycles). Additionally, the four port IPv4 router that has been used as an example throughout our results takes approximately 5400 cycles to execute.

The long term unfairness comes from when a thread is

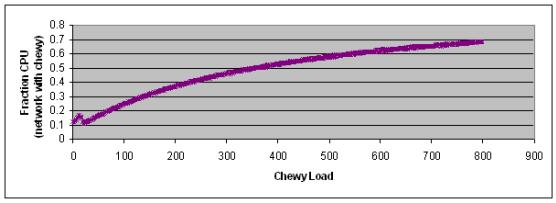

Figure 11: Long-term unfairness: the fraction of CPU that a thread limited to fifteen percent receives as a function of the per packet processing time

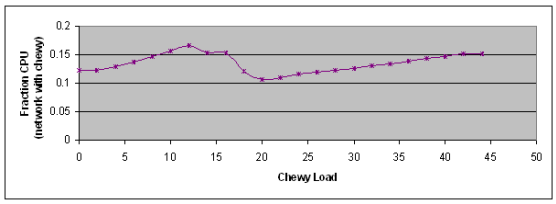

Figure 12: A zoomed in version of the long-term unfairness results, showing the limiting for run times under one token

executing and the context it is associated with runs out of tokens. From this point until when the thread yields, the CPU cycles are unaccounted, leading to the context getting more than its fair share. This can be seen in Figure 11 and a zoomed in version in Figure 12. To obtain these graphs, two threads were run concurrently doing IPv4 forwarding. In each case, an infinite source was used that generates packets at a rate as fast as is possible - as soon as one packet has been forwarded, another packet is generated to be forwarded. One of the threads was limited to fifteen percent of the CPU and included an element, chewy, that did nothing more than chew a parameterizable number of cycles. As the chewy load increases, the per packet load is greater, and therefore the thread runs longer. From the graph in Figure 11, one can see that above a chewy load of 45[4], the thread that was limited to fifteen percent gets more than fifteen percent of the CPU, its allocated share. A load of 45 corresponds to a per packet processing time of about 10,000 cycles on top of the 5,000 cycles to do the forwarding. As each thread forwards 256 packets before yielding, this works out to roughly 1 ms on a 3 GHz CPU. This is, as expected, the granularity of a token, confirming the assertion that not yielding after running out of tokens leads to unfairness.

[4]where a load of one corresponds to about 200 cycles

7 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 [18] 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 to say more definitively.

While originally meant as a way to protect a system against faulty drivers, the Nooks system [21] 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 [17] 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 gen-

eral platform across multiple custom (virtual) network devices.

In addition to approaches to support virtual networks, there have been two previous efforts to map Click to FPGAs. Cliff [13] mapped a Click configuration to an FPGA as a pipeline of implemented Verilog elements. Each element was assumed to exist as a Verilog module with each packet interface using a standard set of signals. From the graph specified in the Click configuration, Cliff then generated a top level Verilog file for the system. This approach was the starting point for our implementation, but it did not make use of Click's meta-data specification, did not support handlers, and did not generalize the interface to the external environment. CUSP [19] is another tool for mapping Click configurations onto an FPGA. CUSP focused on providing support for parallel execution. By annotating elements, written in an HDL, with information about what parts of a packet are read and written, the CUSP tool can determine if two elements can process the same packet concurrently. This is an optimization over Cliff and could be used within the *click-fpga* tool presented in this paper as well.

8 Future Work

There are several 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.

Another direction is to further examine support for FPGAs. We provided a first implementation, however there are several issues that were not considered. In particular, the ability to "hot-swap" is not trivial in FPGAs. There is support for partial reconfiguration as well as much prior work on run-time reconfiguration [8][10], however it is still an unsolved problem. However, just as virtualization is made simpler through the use of a language that specifies packet processing as a graph, so too could the task of "hot-swapping" in FPGAs. Additionally, further work is needed when considering resource accounting in the context of the FPGA implementation. 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, though they are quite different as with the FPGA we have the ability to modify the hardware implementation of the memory controller.

Further work is also needed on applications. In particular, one interesting possibility is the area of virtual router migration. The idea is to maintain the logical topology of routers while being able to change the physical machine the router is running on. A router includes both the control functionality as well as the packet processing functionality. In [23], the packet processing was the Linux networking stack and migration was done through the use of duplicating tables. It would be interesting to consider migration where the packet processing is specified as a Click configuration.

9 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, including increasing isolation by executing in a container in user space.

Finally, we discuss an implementation of Click on an FPGA as a demonstration of how using a language, where packet handling is specified as a graph of common networking functions interconnected to indicate packet flow, extends beyond commodity hardware.

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.

References

[1] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03:*

Proceedings of the nineteenth ACM symposium on Operating systems principles (2003).

[2] BAVIER, A., FEAMSTER, N., HUANG, M., PETERSON, L., AND REXFORD, J. In VINI Veritas: Realistic and controlled network experimentation. In *Proc. ACM SIGCOMM* (Sept. 2006).

[3] BHATIA, S., MOTIWALA, M., MUHLBAUER, W., VALANCIUS, V., BAVIER, A., FEAMSTER, N., PETERSON, L., , AND REXFORD, J. Hosting virtual networks on commodity hardware. Georgia Tech Computer Science Technical Report GT-CS-07-10, January 2008.

[4] BICKET, J., AGUAYO, D., BISWAS, S., AND MORRIS, R. 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* (New York, NY, USA, 2005), ACM, pp. 31–42.

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

[6] EGI, N., GREENHALGH, A., HANDLEY, M., HOERDT, M., MATHY, L., AND SCHOOLEY, T. Evaluating xen for virtual routers. In *International Workshop on Performance Modelling and Evaluation in Computer and Telecommunications Networks (PMECT)* (2007).

[7] ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. 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] GUCCIONE, S., LEVI, D., AND SUNDARARAJAN, P. Jbits: A java-based interface for reconfigurable computing. In *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)* (September 1999).

[9] HANDLEY, M., HODSON, O., AND KOHLER, E. XORP: An Open Platform for Network Research. In *First Workshop on Hot Topics in Networks* (Oct 2002).

[10] HORTA, E. L., LOCKWOOD, J. W., TAYLOR, D. E., AND PARLOUR, D. Dynamic hardware plugins in an fpga with partial runtime reconfiguration. In *Design Automation Conference (DAC)*.

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

[12] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (Aug. 2000), 263–297.

[13] KULKARNI, C., BREBNER, G., AND SCHELLE, G. Mapping a domain specific language to a platform fpga. In *DAC '04: Proceedings of the 41st annual conference on Design automation* (2004), pp. 924–927.

[14] LI, Y.-T. S., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *32nd ACM/IEEE Design Automation Conference* (June 1995), pp. 456–461.

[15] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., , AND LUO, J. Netfpga - an open platform for gigabit-rate network switching and routing, June 2007.

[16] MAZU NETWORKS. *Mazu Profiler Datasheet*. Cambridge, MA, 2007.

[17] MONTZ, A. B., MOSBERGER, D., O'MALLY, S. W., PETERSON, L. L., AND PROEBSTING, T. A. Scout: a communications-oriented operating system. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (1995).

[18] OPENVZ. *http://openvz.org*, 2007.

[19] SCHELLE, G., AND GRUNWALD, D. 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* (2005), pp. 246–257.

[20] SOLTESZ, S., PTZL, H., FIUCZYNSKI, M., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of EuroSys 2007* (Lisbon, Portugal, March 2007).

[21] SWIFT, M. M., MARTIN, S., LEVY, H. M., AND EGGERS, S. J. Nooks: an architecture for reliable device drivers. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop* (2002), pp. 102–107.

[22] TURNER, J. A proposed architecture for the geni backbone platform. GENI Design Document 06-09, March 2006.

[23] WANG, Y., KELLER, E., BISKEBORN, B., VAN DER MERWE, J., AND REXFORD, J. Virtual routers on the move: Live router migration as a network-management primitive. In *Proc. ACM SIGCOMM* (2008).

[24] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. pp. 255–270.

[25] WILLMANN, P., SHAFER, J., CARR, D., MENON, A., RIXNER, S., COX, A. L., AND ZWAENEPOEL, W. 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).