Software Defining System Devices with the 'Banana' Double-Split Driver Model

Dan Williams
IBM Watson Research
Yorktown Heights, NY

Hani Jamjoom
IBM Watson Research
Yorktown Heights, NY

Hakim Weatherspoon
Cornell University
Ithaca, NY

Abstract

This paper introduces a software defined device driver layer that enables new ways of wiring devices within and across cloud environments. It builds on the split driver model, which is used in paravirtualization (e.g., Xen) to multiplex hardware devices across all VMs. In our approach, called the *Banana Double-Split Driver Model*, the back-end portion of the driver is resplit and rewired such that it can be connected to a different back-end driver on another hypervisor. Hypervisors supporting Banana cooperate with each other to (1) expose a consistent interface to rewire the back-end drivers, (2) allow different types of connections (e.g., tunnels, RDMA, etc.) to coexist and be hot-swapped to optimize for placement, proximity, and hardware, and (3) migrate back-end connections between hypervisors to maintain connectivity irrespective of physical location. We have implemented an early prototype of Banana for network devices. We show how network devices can be split, rewired, and *live* migrated across cloud providers with as low as 1.4 sec of downtime, while fully maintaining the logical topology between application components.

1 Introduction

The flexibility and agility of the cloud stems from decoupling the functionality of devices from their physical hardware. However, for system devices the decoupling is incomplete. Virtual devices often hold dependencies on physical components, limiting the flexibility of cloud resource management. For example, they may depend on the presence of a physical device (e.g., a GPU, FPGA, etc.) or a device-related configuration (e.g., VLAN configuration, firewall rules, etc.). While device-specific mechanisms to complete the decoupling exist (e.g., virtual networking for NICs [4]), we aim to design a generic, software-defined mechanism for device decoupling.

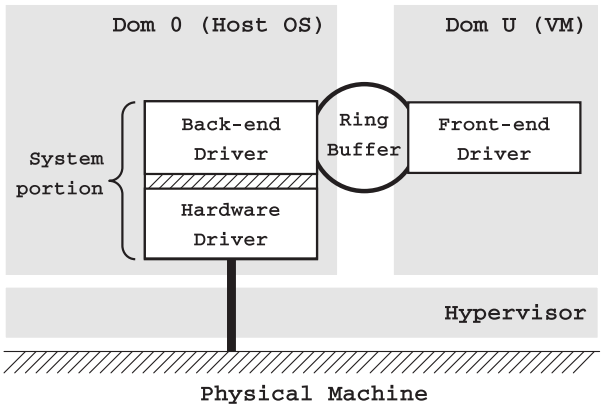

Figure 1: Xen's split driver model

Specifically, we reexamine the architecture of the split driver model that was popularized by Xen paravirtualization [1]. As shown in Figure 1, the split driver model consists of two parts: the front-end and back-end drivers. The two components are connected through a ring buffer, a bidirectional data pipe. The front-end driver is accessed by the virtual machine (running in Domain U in Xen). The back-end driver is managed by the host OS (running in Domain 0 in Xen) and interfaces with the underlying hardware drivers.

The split driver model enables multiplexing all access to the underlying physical hardware, but only partially decouples the underlying hardware from the end hosts. It does not provide location independence, limiting potential optimizations based on VM placement, proximity, or available hardware. This is because back-end drivers are coupled with the underlying hardware drivers in an ad-hoc fashion.

This paper proposes the *Banana Double-Split Driver Model* (Banana, for short). Banana extends the split driver model by further splitting the back-end driver into

two logical parts: the *Corm* and the *Spike*.[1] The Corm multiplexes access to the underlying hardware, while the Spike talks to the guest OSs. These two subcomponents expose a standard interface (*endpoints*) by which a Banana controller can *wire* or connect together Spikes and Corms. The Banana controller is software defined, allowing the external definition and on-the-fly reconfiguration of both (1) the pairing of the Spike and Corm and (2) the wire format connecting them. The wire format can be switched between local memory copies or a variety of network connections (i.e., OpenFlow, VPN, STT, VXLAN, RDMA, etc.), depending on the physical location of Corms and Spikes.

As a first step, we provide an alternate approach to virtualizing NICs in Xen [1]. NICs are a natural starting point as the resulting double split communication core creates a good reference architecture for other devices like GPUs and FPGAs. We focus on (1) how wires between Banana Spikes and Corms can be stretched across clouds, and (2) how, using a simple *connect*/*disconnect* primitive, Banana controllers can easily switch Spike/Corm mappings. Specifically, leveraging the Xen-Blanket [16] augmented with Banana, we performed cross-cloud live migration of VMs from our private cloud to Amazon EC2. The entire live migration process involved no changes to VMs or their network configurations and incurred as low as 1.4 s of downtime.

2 Going Bananas

Banana supports a double-split architecture, illustrated in Figure 2. This section describes three design elements: (1) the splitting of the back-end—system—portion of device drivers, (2) the wiring of double-split device driver components, and (3) the management of wires (connections) during VM provisioning and during VM migration.

2.1 Double Splitting Drivers

Banana modifies what we refer to as the *system portion* of a split device driver. The system portion contains the back-end driver (in paravirtualization), the hardware driver, and all of the ad-hoc code that connects the two. The system portion as a whole traditionally implements two primary features: (1) it multiplexes the underlying hardware, and (2) it interfaces with the virtualized guests (illustrated in Figure 1). In Xen, the hardware is multiplexed in an ad-hoc manner depending on device type. For example, the network may be multiplexed using a bridge or virtual switch, whereas a disk may be multiplexed using the filesystem. The interface with the guests is via ring buffers between the host OS (running in Domain 0) and guest OSs (each running in Domain U).

[1]The "corm" in a banana plant refers to the parts that are above ground (similar to a tree trunk). The "spike" is inflorescence in which the banana fruit grows.

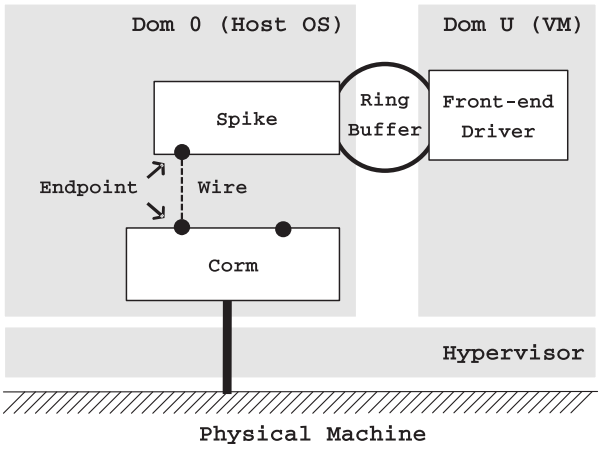

Figure 2: Banana Double Split Driver Model

Banana cleanly demarcates these two functionalities into two logical components: the *Corm* and the *Spike* (Figure 2). The Banana Corm essentially maps to the portion of the driver that multiplexes the underlying hardware. As such, there is typically one Corm per physical device; it is tied to a physical machine. The Banana Spike corresponds to the portion that talks to guest OSs. Unlike existing implementations, the Corm and Spike are not hardwired together nor do they have to run on the same physical host. Instead, the two components expose *endpoints*, which resemble virtual NICs, that can be wired together such that one or more Spikes from different physical machines can communicate with the Corm on a given physical machine.

Consider a base system in which both the Spike and Corm of the network driver reside on the same physical machine. Here, an endpoint on the Spike represents an interface to one of the VMs and an endpoint on the Corm represents an interface to the underlying network. As described later, the wiring between Corms and Spikes implies that VMs can be moved around irrespective of physical location.

2.2 Wires

Wires are essentially connections between two *endpoints*, each bound to a Corm or a Spike. The binding between endpoints is configured by an *endpoint controller* residing in the hypervisor. Each endpoint has a unique identifier. Consequently, Banana does not restrict how wires (connections) are implemented as long as the endpoint controller is able to properly associate connections

with endpoints. Furthermore, the choice of wires can depend on the location of the endpoints. If, for example, two endpoints reside in different data centers, the wire can be a VPN tunnel.

Wires are inherently decentralized. For example, endpoints could be globally addressable within a Banana network through a hierarchical naming convention rooted at hypervisor identity. Each endpoint could have a *home hypervisor*[2] that keeps track of the current location of the endpoint, while the endpoint controller in each hypervisor responds to queries pertaining to its active endpoints. The binding between endpoints is maintained even if the corresponding component is migrated to another hypervisor. On migration, the configuration of endpoints is updated to ensure the wire topology is maintained (described next).

2.3 Wire Management

Banana exposes a simple interface made up of two operations, `connect` and `disconnect`, which create and destroy wires between the Spikes and Corms. These interfaces augment the typical VM creation and destruction interfaces exposed by hypervisors.

To maintain the logical wire topology as VMs migrate, the endpoint controller on each hypervisor is integrated with its live migration mechanism. During migration, the source hypervisor copies the affected endpoint configurations to the destination hypervisor as part of the VM metadata. The wire format need not remain the same: for example, if the wire crosses administrative domains (e.g, different clouds) a VPN may be used in place of a simple encapsulation method (e.g., VXLAN).

3 Implementation

We have implemented Banana in Xen, initially focusing on modifying its virtual NICs to support the Banana Double-Split Driver model. We also augment existing hypervisor live migration mechanisms to also migrate wires and endpoints.

Endpoints. We kept the endpoint implementation separate from both the back-end and hardware drivers. Endpoints appear to the system as independent network devices, just like the vNICs and hardware devices. Figure 3 depicts how endpoint devices are connected to the back-end and hardware devices to form the Banana Spike and Corm. To form the Spike, we use a dedicated software network bridge, called the endpoint bridge, to connect the endpoint network device to the back-end driver. To form the Corm, we connect one or more endpoint devices to the virtual switch to which the hardware device is connected.

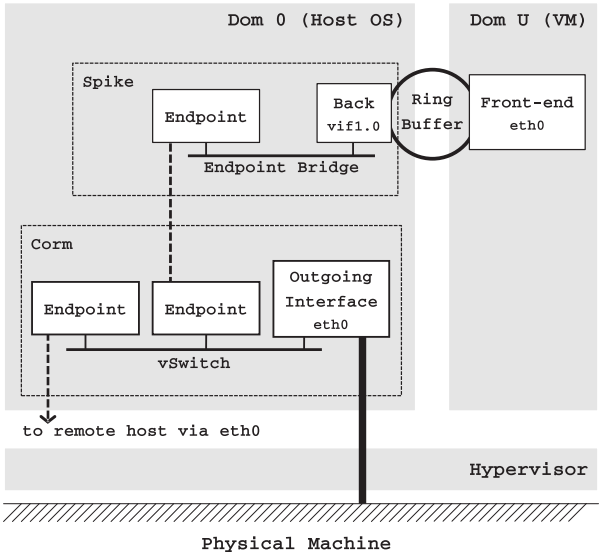

Figure 3: Virtual Ethernet in Banana

Types of Wires. We implemented three types of wires: native, encapsulating, and tunneling wires. Native wires directly bridge spikes and corms that are colocated on the same physical machine. Encapsulating wires implement VXLAN. We implemented the entire endpoint in a Domain 0 kernel module to reduce the number of context switches required when compared to a userspace implementation, such as `vtun`, which uses the TAP/TUN mechanism. Inside the kernel module, packets traveling across a wire between Spikes and Corms are intercepted and encapsulated in UDP packets, then sent through Domain 0's external interface. Finally, the tunneling wire is OpenVPN-based, which can be used to tunnel across different clouds.

Switching Wires and Corms. We implemented an interface in the `/proc` filesystem in the endpoint controller kernel module, through which endpoints are controlled. This interface provides a convenient mechanism to create and destroy wires as needed. It is also used by the local migration process to update endpoints during a VM migration.

Migrating Endpoints. We modified the local Xen live VM migration process [3] in two ways to include Banana migration.[3] First, when the destination creates the empty

[2]The home hypervisor is akin to the Local Mobility Anchor in Proxy Mobile IP (PMIP) [14], and could be implemented robustly through consistent hashing.

[3]During live migration, the source hypervisor instructs the destination hypervisor to create an empty VM container. Then, the source iteratively copies memory to the destination, while the migrating VM continues to run. At some point, the source stops the VM and copies the final memory pages and other execution state to the destination.

	Downtime	Duration
Banana	1.3 [0.5]	20.13 [0.1]
No Banana (nested)	1.0 [0.5]	20.04 [0.1]
No Banana (non-nested)	0.7 [0.4]	19.86 [0.2]

Table 1: Mean [w/standard deviation] downtime (s) and total duration (s) for local live VM migration

	Downtime	Duration
Local to Local	1.3 [0.5]	20.13 [0.1]
EC2 to EC2	1.9 [0.3]	10.69 [0.6]
Local to EC2	2.8 [1.2]	162.25 [150.0]

Table 2: Mean [w/standard deviation] downtime (s) and total duration (s) for live VM migration with Banana

VM container (during the pre-copy phase), it also recreates the vNICs and endpoints to form Banana Spikes for each interface of the VM. The newly created Spikes at the destination are wired to the same Corms that the corresponding Spikes at the source were wired to. Second, during the stop-and-copy phase, all Corms for which a Spike is attached are rewired to the new Spikes at the destination by signaling the appropriate endpoint controllers to update relevant endpoint configurations. Finally, after migration completes, the source deletes the original Spikes.

4 Banana in Action

We evaluate the efficacy of Banana in the context of our implementation of the Banana NIC. Specifically, we demonstrate how stretching Banana wires and switching Corms can simplify cross-cloud *live* migration of VMs from our local environment to Amazon EC2. Live VM migration across clouds typically require complex setup of gateways and VPNs to extend layer 2 domains. Banana wires abstract away this complexity in a uniform manner.

While we experimented with complex application setups, due to space considerations, we focus on two VMs connected through a virtual switch, with one VM continuously receiving `netperf` UDP throughput benchmark traffic from the other VM.[4] Initially, both VMs are colocated. We use resources at our local institutions as well as from Amazon EC2. Since we do not have access to the underlying hypervisor in Amazon EC2, we leverage the Xen-Blanket [16] for nested virtualization. For consistency, we use the Xen-Blanket in both setups. In either case, the Xen-Blanket Domain 0 is configured with 8 VCPUs and 4 GB of memory. All guests (DomUs) are paravirtualized instances configured with 4 VCPUs and 2 GB of memory.

Cross-cloud Live Migration. The performance time of live migration of a VM (receiving `netperf` UDP traffic) between two machines in our local setup is shown in Table 1. We examine the components of the migration overhead by comparing against VM migration without Banana, in which vNICs are directly bridged onto the physical network. Since this is on our local setup, we also show results for migration without nested virtualization (non-nested). We find a 43% increase in downtime and an 18% increase in total duration due to nesting. The added task of migrating Banana endpoints introduces an additional 30% increase in downtime, but a negligible increase in total duration. We note that live migration without Banana is restricted to a single physical network subnet because the vNICs are bridged onto the physical network.

Table 2 quantifies the performance of live migration across clouds using Banana. We compare the performance of single-cloud live migration within our local (nested) setup (*Local to Local*) and within Amazon EC2 (*EC2 to EC2*) to multi-cloud migration between the two (*Local to EC2*). Within one cloud, local or EC2, the latency between the instances is within 0.3 ms, whereas across clouds it is about 10 ms. VPN overhead limits throughput across clouds to approximately 230 Mbps. The 10 Gbps network between our EC2 instances leads to significantly reduced total migration time when compared to local; however, the downtime was comparable. Live migration of a VM (receiving `netperf` UDP traffic) between our local nested setup and Amazon EC2 has a downtime of 2.8 s and a total duration of 162.25 s on average, but variance is high: the duration ranged from 48 s to 8 min. For an idle VM, the performance of the network between machines has little effect: the downtime during live migration between two local machines and from local to EC2 is 1.4 s on average.

We ran two more experiments,[5] shown in Figure 4 and Figure 6, to identify the throughput and latency over time for the test deployment as the recipient VM and then the sender VM were live migrated to Amazon EC2, respectively. As expected, significant degradation in the throughput and latency occurs when the VMs are not colocated on the same cloud.

Banana Wire Performance. We examine the overhead of our VXLAN encapsulation for Banana wires, highlighting the effect of the in-kernel implementation of Banana endpoints. Figure 5 compares the performance

[4] We use `netperf` with 1400-byte packets and UDP_STREAM (for throughput) and UDP_RR (for latency) modes throughout.

[5] We could not measure both the throughput and latency from a single experiment using `netperf`.

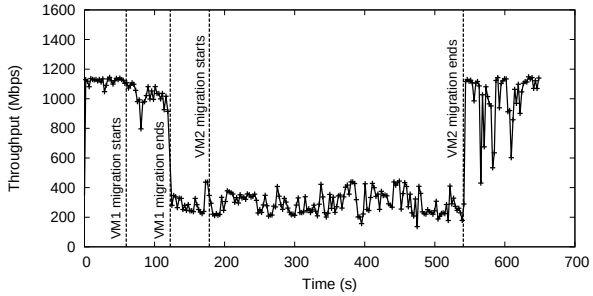

Figure 4: Throughput over time between two VMs migrating to Amazon EC2

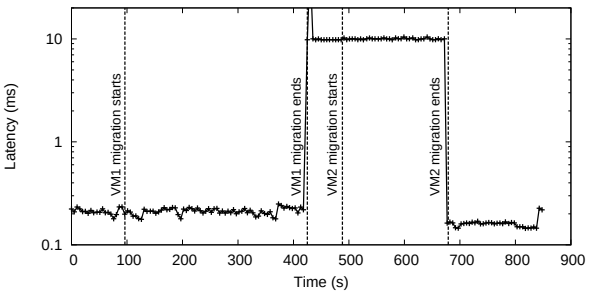

Figure 6: Latency over time between two VMs migrating to Amazon EC2

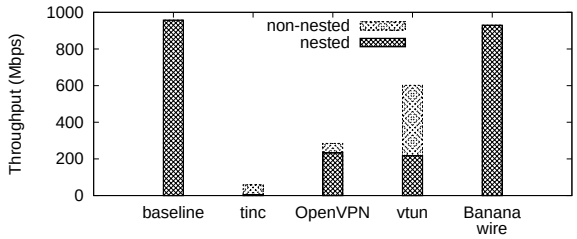

Figure 5: Tunnel throughput comparison

of Banana wires against various other software tunnel packages on our local 1 Gbps setup. We measure the UDP performance of a VM sending data to another VM through each tunnel. The baseline is a measurement of the direct (non-tunneled) link between the VMs. The kernel-module approach of Banana wires pays off especially for nested environments: while Banana wires show an increase in throughput by a factor of 1.55 over the popular open-source tunneling software `vtun` in a non-nested scenario, they show a factor of 3.28 improvement in a nested environment. This can be attributed to the relatively high penalty for context switches experienced in the Xen-Blanket [16]. These results suggest that, while any tunnel implementation can be used for wires in Banana, it should be optimized for the environment.

5 Related Work

Banana drivers provide a cut point for device drivers designed to provide location independence. There has been work on general frameworks to virtualize devices that focus on block devices. Block tap drivers [15] provide a different cut point in the block device I/O path where arbitrary functionality, including Banana, can be implemented in userspace.

Netchannel [10] enables live migration of device state, remote access for devices, and hot swapping, or switching out the physical device that a virtual device maps to. While a general framework, Netchannel primarily addresses I/O block devices. While we have so far focused on network devices, we envision Banana to benefit from the exploration of block devices in Netchannel. However, whereas Netchannel emphasizes continuous access to devices, we are interested in developing a clean and general wiring interface to achieve location independence.

More specific to devices, Nomad [7] discusses the migration of high-speed network devices that are mapped directly into guests, bypassing the hypervisor. USB/IP [6] achieves location independence for peripherals by virtualizing devices within an OS driver. Neither approach is general for a wide class of virtualized devices.

The focus of Banana on network devices was inspired by emerging virtual networking architectures, such as VL2 [4] and NetLord [12], as well as software-defined network environments [5, 9, 11, 13] that aim to enable location independence for the network. Banana complements these efforts by introducing location independence in the VM driver stack.

Banana can be directly applied to create overlays of virtual network components, such as VIOLIN [8] or VINI [2]. In these systems, the overlay can benefit from the Banana architecture, creating new opportunities for rewiring VM topologies and migrating various components without disruption.

6 Conclusion

This paper presented Banana, an architecture in which device drivers can be virtualized. It builds on the current split driver model that is found in paravirtualization to allow the back-end portion to be arbitrarily rewired. We have demonstrated the efficacy of our approach in creating virtual networks that can migrate between clouds by only modifying Xen's virtual NIC device driver. We are actively modifying other device drivers with the hope of creating a robust device virtualization layer that can unify ongoing approaches in storage, GPUs, and any other system resource.

7 Acknowledgments

This work was partially funded and supported by an IBM Faculty Award received by Hakim Weatherspoon, DARPA (numbers D11AP00266 and FA8750-11-2-0256), US National Science Foundation CAREER (number 1053757), NSF TRUST (number 0424422), NSF FIA (number 1040689), NSF EAGER (number. 1151268), and NSF CiC (number. 1047540). We thank Zhefu Jiang for work on an earlier version of this paper and the anonymous reviewers for their comments.

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 *Proc. of ACM SOSP* (Bolton Landing, NY, Oct. 2003).

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

[3] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proc. of USENIX NSDI* (May 2005).

[4] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM* (Spain, Aug. 2009).

[5] GUDE, N., KOPONEN, T., PETTIT, J., PFAFF, B., CASADO, M., MCKEOWN, N., AND SHENKER, S. NOX: Towards an Operating System for Networks. *ACM SIGCOMM CCR 38*, 3 (July 2008).

[6] HIROFUCHI, T., KAWAI, E., FUJIKAWA, K., AND SUNAHARA, H. USB/IP: a peripheral bus extension for device sharing over IP network. In *Proc. of USENIX Annual Technical Conf.* (Anaheim, CA, Apr. 2005).

[7] HUANG, W., LIU, J., KOOP, M., ABALI, B., AND PANDA, D. Nomad: migrating os-bypass networks in virtual machines. In *Proc. of ACM VEE* (San Diego, CA, June 2007).

[8] JIANG, X., AND XU, D. Violin: virtual internetworking on overlay infrastructure. In *Proc. International conference on Parallel and Distributed Processing and Applications* (Hong Kong, China, Dec. 2004).

[9] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., , HAMA, T., AND SHENKER, S. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of USENIX OSDI* (Vancouver, Canada, Oct. 2010).

[10] KUMAR, S., AND SCHWAN, K. Netchannel: a VMM-level mechanism for continuous, transparentdevice access during VM migration. In *Proc. of ACM VEE* (Seattle, WA, USA, Mar. 2008).

[11] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR 38*, 2 (Apr. 2008).

[12] MUDIGONDA, J., YALAGANDULA, P., MOGUL, J., STIEKES, B., AND POUFFARY, Y. NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *Proc. of ACM SIGCOMM* (Toronto, Canada, Aug. 2011).

[13] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending Networking Into the Virtualization Layer. In *Proc. of ACM HotNets* (New York, NY, Oct. 2009).

[14] WAKIKAWA, R., AND GUNDAVELLI, S. IPv4 Support for Proxy Mobile IPv6. RFC 5844, May 2010.

[15] WARFIELD, A., HAND, S., FRASER, K., AND DEEGAN, T. Facilitating the development of soft devices. In *Proc. of USENIX Annual Technical Conf.* (Anaheim, CA, Apr. 2005).

[16] WILLIAMS, D., JAMJOOM, H., AND WEATHERSPOON, H. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proc. of ACM EuroSys* (Bern, Switzerland, Apr. 2012).