CHAPTER

7 THE VIRTUALIZATION LAYER—PERFORMANCE, PACKAGING, AND NFV

INTRODUCTION

Proponents of NFV see applications that stretch from the replacement of individual fixed network appliances (eg, firewall) through reimagination of whole service architectures (eg, vEPC and vIMS), to large swathes of the forwarding infrastructure (eg, vRouters). When you peel away the service state components (eg, bearer/tunnel management or subscriber management), which we can agree are well suited for processor offload, these applications are built around network I/O.

Many vendors of standalone network appliances were already moving from their dedicated chipsets to an Intel- or ARM-based architecture before NFV was in our lexicon. The fusion of datacenter orchestration methodologies (enabling elasticity and management paradigms for this transformation), more aggressive marketing by manufacturers like Intel (and the optimizations of their driver architecture and core technologies to make them more suited to network I/O applications) accelerated an existing trend and expanded the scale of potential applications to Service Provider scale (clusters of machines capable of handling hundreds of gigabits per second of throughput).

In this chapter, we will look at the evolution of virtualization techniques and software network I/O acceleration to satisfy these application requirements. We will also look at how the constant evolution in the compute component of the (ETSI-labeled) "NFVI" might affect the current aggregated, VM-centric NFV model, our concept of Service Function Chaining and the potential economic assumptions behind the NFV proposition.

The original vision of the economics behind NFV was predicated on the use of "COTS" (commercial off-the-shelf) compute—bespoke hardware (general purpose processors[1]) and operating systems found in a data center environment. The fundamental attraction of a COTS-based solution is in the simplification of supply chain, inventory management, and operations gained from the use of common equipment and software configurations.

While the current discussion of NFV and the ETSI NFV architecture are predicated around an Intel model (it is no accident that Intel is a leading contributor to and proponent of the de facto NFV architecture), through the course of this and succeeding chapters it may be fair to ask if this is just a "first phase" assumption that enables conceptual adoption of NFV.

The pursuit of network IO performance on Intel x86 can introduce the use of specialized environments and toolkits (in comparison to other data center workloads). There is an implied

Network Function Virtualization. DOI: http://dx.doi.org/10.1016/B978-0-12-802119-4.00007-9

dependency on rapid architectural changes in the Intel Architecture (IA) that bring into question the definition of COTS and the economics of the model.

Also, in the current Intel-centric environment, a classic chase of performance bottlenecks in software has emerged.

Whenever the narrowest bottleneck is removed from a system, the next-less- narrow bottleneck will become the limiting factor.[2]

EVOLVING VIRTUALIZATION TECHNIQUES

The proposed ETSI NFV solution architecture optimizes limited, shared resources through virtualization.

There are numerous techniques to share the CPU resource, and they revolve around privilege (based on the x86 privilege four "ring" model—with Ring 0 being the most privileged and Ring 3 being the common privilege level for applications) and resource abstraction.

Virtualization started with emulation (in which all CPU instructions are translated from guest to host) and native virtualization (in which privileged commands are passed to the host kernel without translation but access to other peripherals is still emulated).

The hypervisor is the most common construct in machine virtualization today. There are two different types—Type 1 (eg, Xen, KVM, VMware ESX) in which the virtual machine manager (VMM[3]) runs on the host machine without a host OS (native), and Type 2 which runs on a host OS (hosted by, eg, VMware Virtual Workstation, VirtualBox, JVM).

Within these hypervisor types there are two techniques of virtualization: para-virtualization (or OS-assisted) and full virtualization. The mapping of a particular hypervisor to virtualization mode has gotten a bit murky with time (eg, VMware supports a para-virtual-like mode though it is commonly used in a full-virtualization mode).

Using KVM[4] as an example, with paravirtualization (Fig. 7.1), the paravirtualized network driver in the guest, shares virtual queues with QEMU[5] (a hardware emulator). A data path is created through the TAP interface to the guest VM via a virtual queue mechanism (signaling and notifications are a separate logical channel between virtio-net and KVM).

As a networking resource, virtio evolved as the cross-OS I/O device abstraction that replaced fully emulated network devices (eg, the e1000 device in Linux systems).

The original implementation of the driver (virtio[6]) placed the network stack into user space with QEMU handling the virtual queues, causing memory copies and context switches as packets traversed the kernel (KVM) to user (QEMU). Ultimately, vhost-net reduced this overhead by bypassing QEMU and pushing the stack and virtual queue system into the kernel.

While paravirtualization had some advantages, in that guest applications have the ability to make privileged calls to some hardware resources, the network I/O ran from an imbedded paravirtualized driver in the guest operating system that worked in conjunction with a host emulator (which would emulate the function of the shared NIC) via an I/O call to the hypervisor. Nonvirtualize-able instructions are replaced with "hypercalls" to the virtualization layer.

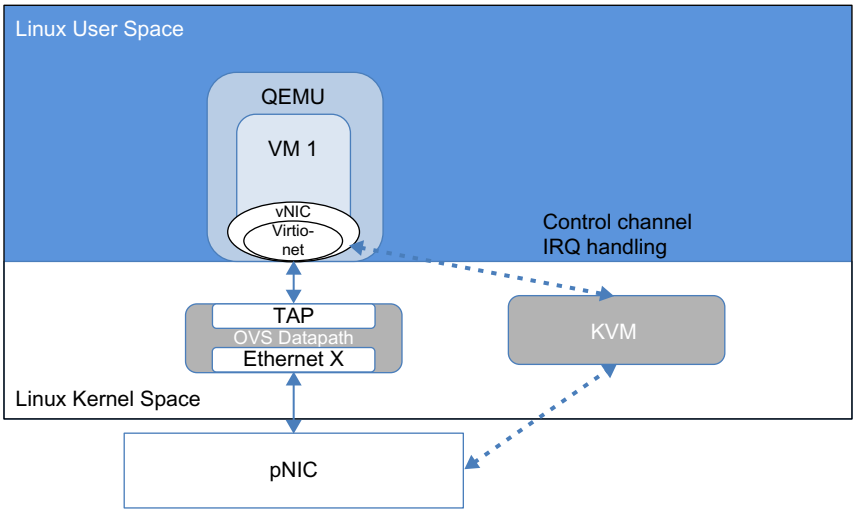

FIGURE 7.1

KVM execution and QEMU emulation.

The reduced performance and required modification of the guest OS (to imbed the driver) combination made paravirtualization less attractive. The original operator-driven NFV focus on performance and operational transparency could eliminate these techniques in favor of full virtualization.

In full virtualization, machine language codes of the guest are translated into machine language codes of the host by the VMM—full binary translation (depending on the implementation, some direct access is possible).[7] Attempts to run privileged commands (eg, accessing a device driver) trigger a trap for the event in the hypervisor. The guest runs unaltered (in Ring 1). As might be expected, translation has performance impacts.

Through its Virtual Machine Extensions (VMX)[8] for the x86 architecture, Intel enabled hardware-assisted virtualization (2005), allowing the VMM to run below Ring 0 (they extended the four "ring" privilege model to include VMX root and non-root modes). The result provides a combination of attributes from full and paravirtualization (the guest is generally unaltered and performance overhead of virtualization is minimized thanks to the hardware assist). This is the focus of NFV VM-centric operation today.

As Fig. 7.2 illustrates, the hypervisor provides abstractions between the physical NIC (pNIC) and virtual NIC (vNIC) associated with a specific virtual machine (VM). These typically connect through the virtual switch construct (vSwitch).[9] However, as we shall see with both PCI pass through and SR-IOV, the VNF can connect directly to the pNIC.

Memory is virtualized as a resource through the cooperative action of the Memory Management Unit (MMU) and Translation Look-aside Buffer[10] (TLB).

The VM is constructed[11] (in cooperation with the local host VMM) from virtual memory that maps back to host memory and virtual CPUs (vCPU) that map back to physical CPU resources (discrete cores or individual threads in a multicore socket).

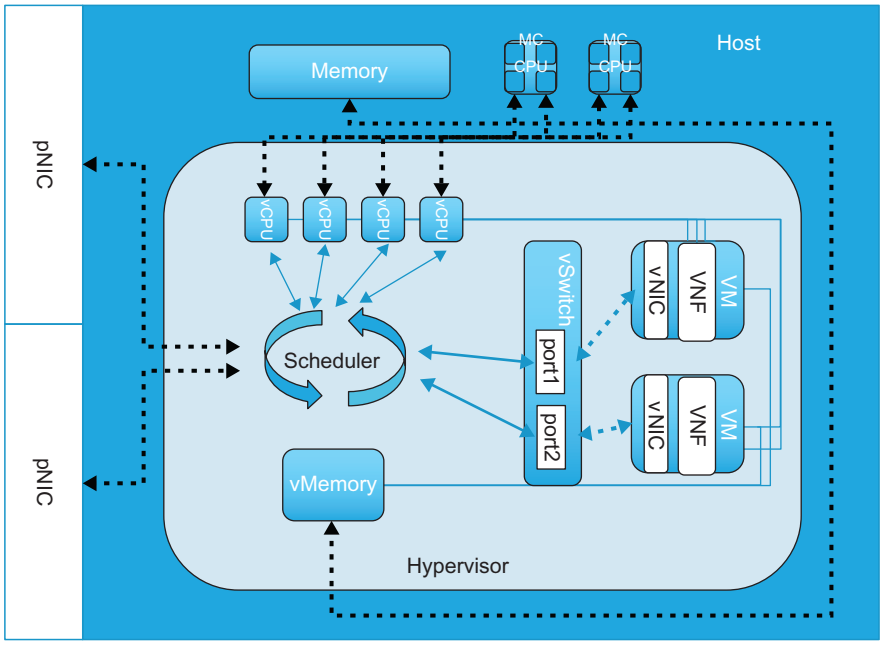

FIGURE 7.2

A hypervisor maps physical resources to virtual ones.

Abstractions of a physical resource into a shared virtual resource affect the memory/CPU overhead on the machine and the speed of orchestration. They may also impact network performance by nature of the pipelines we create between guests and these resources.

THE VM-CENTRIC MODEL

As we saw in Chapter 3, ETSI NFV ISG, the (Phase 1) ETSI model continues to evolve. While subsequent iterations of the work group have begun to discuss containers, the Phase 1 architecture is commonly modeled as "a VNF per VM."[12]

In ETSI NFV architecture it is suggested that all interfunction transfers happen through the hypervisor (through the vSwitch construct). That is, service definition implies a service chain comprised of multiple VNFs connected through the hypervisor—even if the functions are on the same host.

This model trades the ability to understand the service at a network level (providing a clean demarcation point for debugging)—for some performance and resource utilization overhead.

Several alternatives to the default vSwitch (OVS) have appeared that can satisfy this requirement. As a class, "virtual forwarders" have appeared that can run in their own container or VM (in host or guest) offering a range of functions from full stack (some VRF-aware) to simple patch panel between virtualized functions.

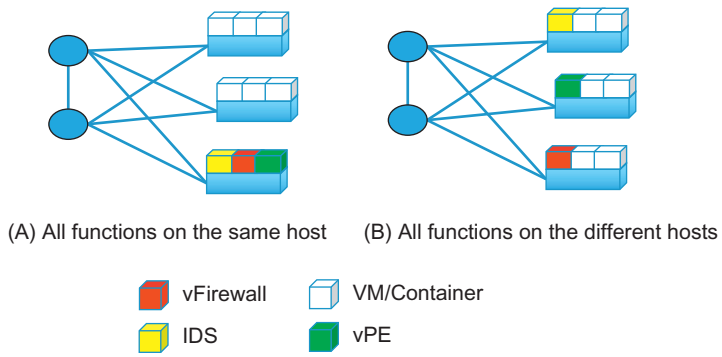

FIGURE 7.3

VM distribution where the VNFs comprising a service are (A) all on the same host and (B) different hosts.

Of course the need to transition through this demarcation point does incur some overhead. For example, in Fig. 7.3, given a service chain that was composed (vPE, vFirewall, vIDS), the most predictable performance scenario will be in "A"—where all the VMs are on the same host (assuming their resource usage patterns are not suboptimal in that configuration—they do not clash). This obviates the variability of network delays between the hosts in the chain and results in two pNIC to vNIC transfers of the packet (coming off the wire into vPE and returning from vIDS or vFW) and one or two transfers via the hypervisor (depending on whether the traffic was forwarded from the vFW to the vIDS).[13]

If, for some reason, the operator thought it better to separate the functions (eg, by type "Firewall," "PE," and "IDS"—as in "B"), the overhead would change to six pNIC-to-vNIC transfers—two at each host (and incur the variability of the intermediating network).

All of these transfers have costs that are part of the current performance focus in NFV studies.

The root of a recommendation that necessitates this overhead might lie in familiarity with the vNIC logic commonly used (at the time of ETSI's work) to connect to the local vSwitch (eg, the traditional tun/tap logic of Linux). Direct connection of VM-to-VM was uncommon[14] and there was an expectation that the vSwitch would connect the VM to the physical network.

The original ETSI recommendation that VM migration was necessary for NFV High Availability is also suspect. This may have been rooted in the familiarity of Telco personnel with VMware (vmotion is a common selling point) and their general lack of familiarity with massively scalable data center application designs (which have their own inherent high availability aspects without vmotion).

In reality, the application/VNF could manage its elasticity on its own or in conjunction with a VNFM.

That elasticity coupled with appropriate system and application designs that protect application state (for those functions that might be stateful) provide the availability model (examples of this approach appear in our chapter on SFC).

Further, VM migration can be unreliable, comes with many caveats, and has some additional expense (at the very least through the performance overhead or resource allocation that has to be planned for the move). And, not all applications are currently designed to work well with vmotion.

Various ETSI workgroups also did point out that the behaviors of one class of application may clash with those of another, so the idea of mixing and matching applications with dissimilar (noncomplimentary) profiles of CPU, disk or network I/O usage may complicate the ability to predict performance of a function (and thus the overall service). There is interesting support of these observations in early application-specific studies[15] (the performance and breadth of virtualization choices have been changing rapidly, quickly dating findings—a recurring theme here).

There may have also been some misconception that VM-based virtualization was required for large-scale orchestration (beyond tools like Puppet/Chef/Ansible). Work has already started in OpenStack to support orchestrated bare metal,[16] and container management environments also have solutions that integrate here.

Experience with the limitations and perceptions surrounding the VM-centric model are the keys to the ongoing evolution of NFV/SFC architectures—which for some applications may soon include containers, microVMs, or bare metal deployment.

The exploration of bare-metal compliments the original ETSI finding that a number of emerging applications for NFV do not require the "V" overhead, since they do not incorporate the per-user or per-organization customization requirements in rendering service (a driver of the ETSI NFV vision) and (due to their critical nature) are unlikely to be shared on a platform with other applications/machines—unless the conceptual revenue streams imagined in the ETSI PoCs and use cases via subletting NFV infrastructure to OTHER carriers (much like a mobile MVNO) bears fruit.[17]

For example, in the mobility architecture, the vGiLAN may be more amenable to a virtualized environment (where a service function may have organization-specific or user-specific instantiations) while the vEPC components (vMME, vSGW, and vPGW) may see no such benefit and thus be more suited to bare metal. In these cases the "v" for virtual is misleading. Simply by moving these functions onto COTS, we are optimizing the solution (without virtualization).

CONTAINERS—DO WE NEED HYPERVISORS?

The most compelling attributes of VMs are security (isolation), compatibility (host OS and VM OS are independent), and mobility (you can move the application while it runs—with a number of caveats).

However, several issues flow from the use of the hypervisor.

Resource scale is an obvious one—every VM has the overhead of a full OS. Without rigorous tuning, nondeterministic behaviors result as well—notably in the scheduling of vCPUs and traffic shaping on vNICs.

The hypervisor is also an additional solution part, and as the solution optimizes—as the industry has moved to solve application performance predictability problems by moving toward direct access to hardware (eg, working around CPU and memory affinity, IOMMU, and interrupt handling by introducing new hypervisors, soft forwarders, and acceleration techniques amongst them) the presence of the hypervisor can create a testing permutation problem in support and validation. An organization might eventually support multiple hypervisor/soft forwarder/OS variants, with differences in performance of and compatibility with various optimizations.

The guest OS(s) also created (yet another) security surface/frontier and maintenance object—the latter creating a perception that maintaining the guest OS presented an obstacle to agile development (application support programs within the VM might require maintenance to enable new features) and portability/mobility.

These issues led some to question the value of the hypervisor abstraction and to the rising exploration of alternative virtualization packaging in containers and microVMs—starting with Linux containers and UML[18] (LXC is a common management tool for Linux containers).

An IBM study in 2014 concluded that containers generally outperform virtual machines (in almost all cases), but both require significant tuning for I/O intensive applications.[19] It further suggests that given the present NUMA (nonuniform memory access) architecture typical of today's servers, that per-socket virtualization (ostensibly, containers) would be more efficient in spreading a workload than bare-metal deployment—spreading the workload across sockets (minimizing cross traffic).

From Fig. 7.4, the key differences between containers and VMs (as they pertain to NFV) are obvious.

A container is an OS kernel abstraction in which the applications share the same kernel (and potentially the bins/services and libs—a sandbox), and the VM is a hardware abstraction in which the applications run separate kernels.

While the VM emulates hardware (leveraging extensions on existing server processor architectures like Intel's VT-x) within the VM, the container does no device emulation and has no specific CPU requirements (easily supported across multiple processor architectures). The container concept does not require a specific CPU, but any given application is compiled for one. Containers replace the x86 whole machine environment with x86 instructions (without virtual memory, trap handling, device drivers, etc.) plus the Linux kernel interface.

Containers exist in Linux[20] user space and leverage dedicated namespaces (namespace separation) for their hostname, processes, networking, user, and IPC. Though they share a kernel, they can isolate resources (CPU, memory, and storage) using cgroups.[21]

In Fig. 7.5, the applications using a network namespace shared with the host appear as applications running natively on the host (physical connections are mapped directly to containers with access through Linux system calls rather than virtio), whereas the applications using separate namespaces appear as applications reachable on a subnet from the host (multiple bridges and virtual topologies are possible).

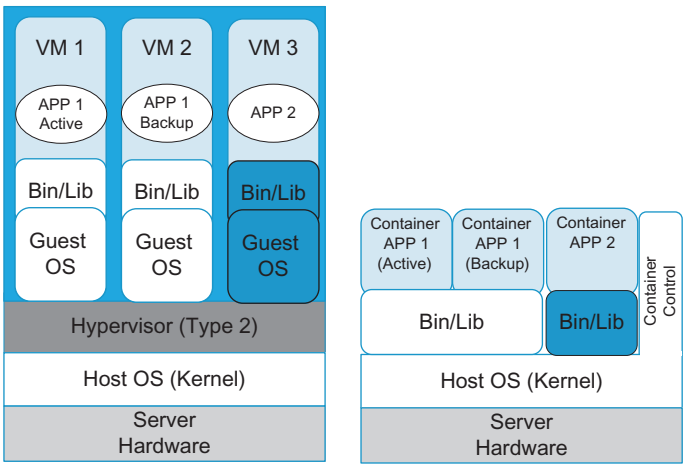

FIGURE 7.4

VMs and containers.

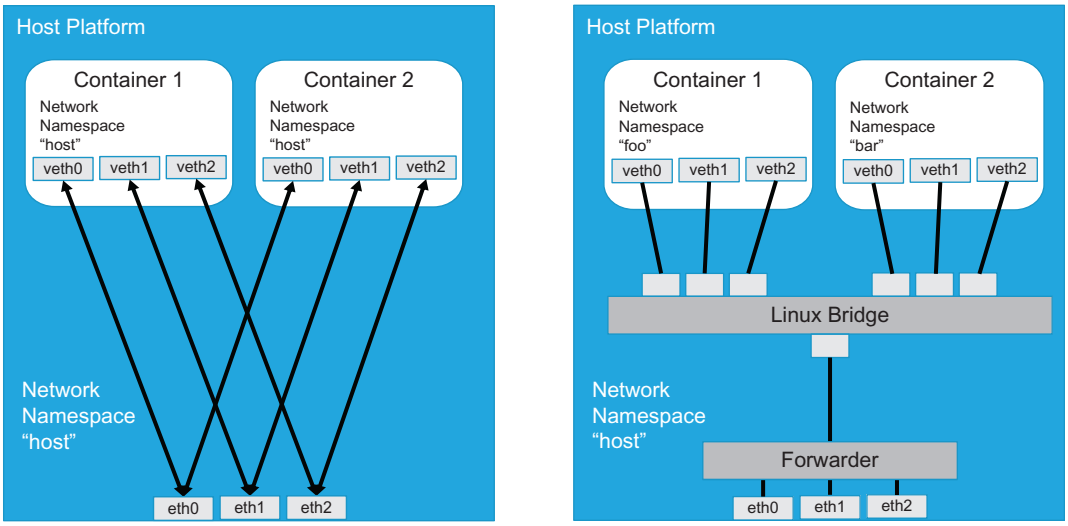

FIGURE 7.5

Linux container (LXC) networking using shared or discrete network namespaces. The host will control a (separate) management Ethernet interface (not shown).

Beyond the Linux container and UML environment, a number of new container runtimes have sprung up, led by Docker.

Docker[22] is built around either LXC or the more customizable libcontainer—so Linux containers are its root technology. It brings an even greater degree of freedom to containers in that it abstracts away host machine specific settings, provides an automatic build environment (supporting a broad range of tools, eg, rpm, make, devops—chef, puppet), provides some default kernel isolation to enhance container security, and has its own repository management tools to push, pull, and commit changes. The key benefit is the abstraction of machine specific settings enabling portable deployment (the same container can run on different machine configurations).

As a development environment, Docker lends itself well to component reuse through the definition and use of reference layers (layered filesystem images) on which the application sits. (These are shareable using copy-on-write mechanisms.)

The overall design[23] (common to Docker and Rocket) allows the developer to bundle an application crafted of these layers (all the environment variables, config, file system) into a container and manage the image and its layers using their public or private Index/Registry(s).

This extra layering/separation makes common provisioning and maintenance tasks a push of a (potentially) thinner application layer (in a largely-Docker environment, the reference layers should already be present on the host) and troubleshooting a simple diff of images to determine layer changes that might have caused problems.

The layering and accompanying automation also makes the applications within the container runtimes more agile around version dependencies between the layers that comprise critical support infrastructure within the container (moving the line for what is "shared" further into the container itself).

Docker's native container networking model uses NAT. It assigns a private range of IP addresses to the containers and maps them to an internal bridge. It uses its own ip table mechanism to manage this port mapping as well as external and intercontainer communication. General connectivity supported by libvirt is incorporated (eg, empty-network, veth, macvlan, etc.). Docker also works with Pipework,[24] a general container networking with DHCP support, external addressing, etc.

The move toward containers brings additional nuance to the orchestration (and networking) solution space, and because of this the market is still developing.

Several Docker networking startups have also sprung up—eg, Socketplane (which was acquired by Docker in 2015),[25] who are proposing an SDN integration solution for Open vSwitch and Docker, and Weaveworks[26] who offer container networking (a private shared virtual switch abstraction for all containers, regardless of location, with options for selective/controlled public connections) and service discovery.

Open source initiatives to improve container networking via OpenStack or Docker are beginning to crop up (eg, Calico which uses IPinIP tunneling for overlay construction and Bird BGP for route distribution—and does kernel based forwarding). fd.io provides a potential forwarding mechanism for containers in user space (fd.io is covered in greater depth later on in this book).

Kubernetes[27] (an open source project started by Google and recently endorsed by RedHat[28]) was an early original pairing with Docker to create an orchestration solution, but Docker had initiated its own solution (Machine/Swarm/Compose).[29] Kubernetes has a different approach to networking and a good comparison with Docker's default approach.[30]

Joyent[31] offers a provider-focused twist on Docker deployments that they claim addresses container security concerns (tenants attacking tenants) around shared Linux OS infrastructure through Linux Branded Zones (LXz). The offering markets against potentially unnecessary performance penalties in a hypervisor-based environment, and plays on some of the security concerns around running "default Linux infrastructure containers."

CoreOS[32] goes in a similar direction, offering a host OS comprised of minimal components (originally systemd, kernel, ssh, and Docker) as the basis for a container runtime environment and as an alternative to the "full Linux distribution plus containers" model. Its focus is the orchestration of Docker containers.

Perhaps spurred by the current opacity of Docker community governance, and Docker's rather monolithic structure, CoreOS (Fig. 7.6) claims that their individual components can run on other distributions than CoreOS (eg, Ubuntu) to provide their services.

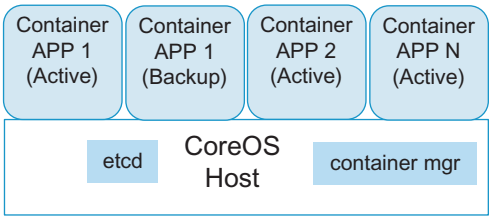

FIGURE 7.6

CoreOS. With the launch of Rocket, the container manager becomes more abstract (you could use either container runtime).

CoreOS emphasizes cluster management and a simplified "orchestration" stack built on their core components: etcd -> fleet -> (multiple) system. These supporting functions enable the containers, but are not specific to them:

- systemd provides init (initialization), resource management, ipc, cron (timer), config, logging—a very simple environment.
- etcd provides key-value store for shared configuration and service discovery. etcd also supports a VXLAN based overlay between containers (Flannel).
- Fleet enables the management of a fleet of system instances, much like Mesos (consensus algorithm is Raft, not Paxos).
- CoreOS has also announced a Kubernetes cluster management interface—Tectonic (https://coreos.com/blog/announcing-tectonic/).
- CoreUpdate provides a management dashboard.
- Flannel provides a simple overlay networking (a private mesh of directly addressed containers).

Another component, "locksmith," is the reboot manager and cluster lock manager (semaphore system) that works in coordination with the update engine to spawn/retire/update members.

Application versioning is managed automatically and includes supporting infrastructure like Java version (moved from the host OS to container) using a partition system to allow reversion.

CoreOS recently launched their own container runtime, Rocket, as a Docker alternative. With the launch of Rocket, CoreOS is attempting to standardize an "App Container Specification"—describing the container image (file assembly, image verification, and placement), container execution environment (cgroups, namespaces, and network), and image discovery behaviors (eg, name to URI mapping).

At this point, containers appear to offer some resource separation with a lower performance penalty[33] than the VM-centric approach—although larger security questions remain to be answered.

Like the early stages of the SDN movement, the container movement is currently volatile and mimics the SDN struggle between for-profit products, open source solutions and for-profit hybrids that enhance open source.

Some clarity as to what parts best remain true open source and the answers to the questions about governance around them may emerge from the opencontainer project in the Linux foundation.[34]

While it is easy to get lost in the orchestration business angle of the CoreOS/Docker entanglement, the underlying container is still a Linux variant (Debian in the case of Docker). That is to say, the surface of binaries and executables within the container, from a security perspective, is still relatively broad—even though the container is ostensibly providing a single service function (microservices).

And, while the distribution management scheme for a container runtime like Docker enables very incremental, layered maintenance, this may not be "best" from a security perspective. If the kernel is made small enough, pushing the (equivalent of a) whole container may be a security scheme easier to manage than allowing individual components to be dynamically altered via orchestration (since this may be the same mechanism an attacker might use, to load a new version of some binary). That is, treating the container as an "immutable" entity may be preferred.[35]

Intel professes to be working on a more secure Linux that leverages their VT technology.[36]

As confusing as the rise of container networking may appear, containers are one of the keys to the most reasonable vision of NFV—microservices. Microservices are collaborative, combinable atoms of functionality that can be reconfigured more nimbly into services. Think of them as the generation beyond "VNF per VM" architectural thinking and the deconstruction step beyond the faltering first step of NFV (simply virtualizing an existing physical function). There is an important tie between the orchestration of containers, the concept of microservices, and the thinking around PaaS coupling to orchestration, as discussed in Chapter 5, The NFV Infrastructure Management.

UNIKERNELS

That brings us to Unikernels, which attempt to address the concept removing any unnecessary binaries and executables—creating a very close linkage between the application and OS (in fact the application code, configuration, and OS runtime are compiled together into the Unikernel). The Unikernel created has a single virtual address space and requires no dynamic linking or initialization of user space applications.

Unikernels are a direct descendent of library operating systems (libOS) but are more feasible now that a hypervisor can provide hardware isolation and virtualization.

Unikernels are seen as a container substitute. A few of the touted benefits of using Unikernels include:

- Compiler driven performance optimizations can be driven by the presence of the configuration (your specific execution environment settings).
- Implementations use higher level language constructs for both the kernel and application that provide memory management advantages.

Unlike the microVM strategy which is focused more on the API exposure within the VM for security, the Unikernel strategy is particularly well adapted for cloud operations. The focus here is on binary size, speed (boot) enable immutable operation. Of course, there IS a potential bonus in security through a much smaller attack surface.

A nonexhaustive list of examples includes Click, OSv,[37] MirageOS,[38] and Unikernel Systems.[39]

The acquisition of Unikernel Systems by Docker implies a merging or hybridization of approach (Unikernels in containers) to realize the vision of "immutable infrastructure."[40]

HYBRID VIRTUALIZATION

In Chapter 2, Service Creation and Service Function Chaining, we also pointed to a hybrid mechanism that leverages either multilevel virtualization using containers and VMs or microVMs.

The VM and container combination allows an orchestrator to combine trusted containers within a secure trust boundary (the VM) to achieve greater scale at lower risk (enforcement extends to a harder resource allocation boundary as well).

The use of a microVM is actually a purposefully different example of this hybrid, and a great example can be found in vCPE application—using a low cost direct tunnel origination (customer site) and termination (operator site) into a microVM. This vCPE model was demonstrated by engineers at Cisco Systems in 2014.[41]

The fundamental thrust of the model is that a kernel-per-customer model of operation for the application (vCPE) underutilizes the kernel scalability. Further, the use of a microVM[42] is critical to the application as it isolates the CPE application from the host OS (the extra isolation provides additional protections against the effects of introduced malware through a form of permitted process awareness and self-regulation).

A multilevel virtualization is applied using a single Linux kernel instance as an outer level that is modified UML[43] featuring multiple subscribers/guests (up to 64 and still scaling in future releases) per kernel being run as a containers with up to 1024 network interfaces per kernel (capable of further scale as general purpose processing performance curve progresses). The inner level containers (also modified UML) run an instance of CPE firmware (eg, OpenWRT, OpenRG) in user space. This design reduces cache thrashing, decreases context switching, and improves memory footprint/efficiency.

New paravirtual network drivers published as open source for kvm/quemu and UML—particularly "raw" and "tap" bring the tunnels/vlans directly into the VM where they support direct encapsulation and decapsulation of overlays and L2VPNs on the vNIC, eliminating the need for virtual switching.[44] These connections are then passed as virtual Ethernets to the containers (the containers do not talk to each other so there is no common bridging or switching functionality required).

This is the first of two important features for NFV architecture discovered in the approach—the ability to hide the overlay network from the VM (there is no shared network state accessible from the containers via a virtual switch if they are compromised).

This architecture (Fig. 7.7) has a few noteworthy tricks including; multi-packet reads, a rewritten paravirtual interrupt controller (Edge vs Level virtual interrupts) that moves the interrupt handling for I/O from the host kernel to the VM, and multipacket transmit I/O events (decreases context switches and system calls per packet).

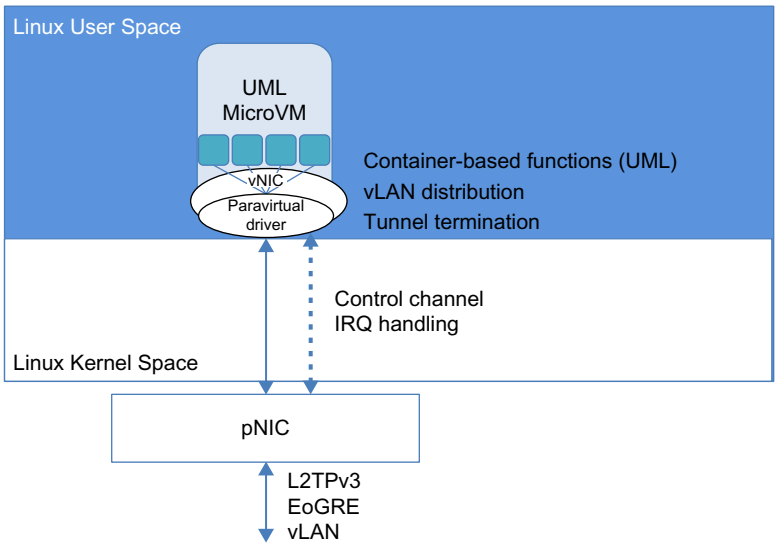

FIGURE 7.7

A UML and container hybrid.

One of the other benefits of using the UML kernel across multiple customers is realized in reusing contexts and timer events for forwarding and QoS, changing their performance/scale from linear to logarithmic.

This is the second important feature for NFV architecture discovered in the approach (and validated in other work[45])—for QoS to work, the VMM (or its equivalent) has to provide a high-performance timer (the developers also wrote and contributed a high-performance timer system using high-performance posix timers to UML).

Other enhancements improved event loop and interrupt controller behaviors that obviated the need for KVM or other alternatives for most use cases (although some of the enhancements to UML mentioned here were also submitted for KVM/qemu).

In its current instantiation, the architecture/model does not use poll-mode drivers (PMDs) (and thus does not disable power/thermal management), but it is adaptable to some of the evolving accelerators covered in the rest of the chapter; NIC or specialized hardware offloads, VM bypass, and kernel helpers like Netmap.

This sort of hybrid suggests that there may be some environments/applications, particularly in the broadband environment, where a hybrid VM/container or microVM/container approach is appropriate for performance, resource optimization, and security.

SECURITY TRADE-OFFS

The VM/hypervisor environment comes with a perception of relative security. "Relative" is used as a description because many security-minded professionals feel that all virtualization mechanisms are potentially fraught with problems—many of those concerns related to hypervisor virtualization are rooted in compliance (eg, user mounted VMs can be noncompliant with corporate security practices) and data security (eg, VMs can be copied, mistakenly orphaned leaving around information). Pervasive virtualization in cloud computing has provided more security experience and contributes to this "relative" sense of wellbeing (as does the burgeoning specialized security product and service industry around cloud and numerous best practice publications).

Hypervisors are small and relatively simple pieces of code and some see "relative" security in that simplicity. But vulnerabilities in the hypervisor can expose VM behaviors like memory use, system calls made, or disk activity that are valuable to an attacker.

Few casual users have knowledge of hypervisor compromises or bugs, yet these do occur[46] and are the potential Achilles' heel of the VM environment—a single potential point of failure that (if breached) can allow a tenant to compromise another tenant or the whole system.[47]

Hypervisors also allow machine-to-machine networking that does not go out onto the wire, creating a potential visibility problem unless instrumented.

"Relative" can also reflect the operator's knowledge of the level of trust amongst coresident virtualized applications. Unlike the typical cloud environment, the NFV environment may often have a single tenant (the provider/operator), and thus VMs may be relatively more secure in this environment than others.

Further, the fact that we are sharing resources between tenants opens the door to potential Denial of Service (DoS) or Distributed DoS attacks between tenants (in the case of NFV and the "single-tenant/operator" scenario, this argument would devolve into inadvertent incompatibilities and bugs).

There is some rigor required around some of the virtualization operation tooling as well (eg, using snapshots to restore virtualized instances can inadvertently revert the instance to a less secure profile).

There is some exposure through file sharing systems so sharing between hosts and guests needs to be minimized and all storage access needs some sort of protection starting with encryption of access.

Given all the above, are containers as "relatively" safe as VMs (or safer)?[48]

At their root, containers are not the exact same abstraction as VMs.

Containers appear to have some intrinsic security via namespace isolation for network interfaces, file systems, and host processes (between host and container), resource allocation control provided via cgroups (assuming cgroups themselves are secure constructs) and limited callable system capabilities.[49] A lot of the security hinges on limiting filesystem access, but they do not have the benefit of hardware virtualization safeguards.

Docker is not totally unaware of the concerns,[50] and has been trying to add security feature to later releases—including seccomp profiles (a level of process control), user namespaces (allowing more granular access policies), and authorization plug-ins (managing access to the Docker daemon and API).[51] (Keep in mind that Docker is not the only container type.)

In addition, we have already mentioned a potential architecture or deployment scenario that removes VM visibility into the overlay (via the vSwitch) for the container in a VM/container hybrid. We have also touched on the hybrid of containers and ukernels (immutable operation).

Securing Linux

Intrinsic protections can be enhanced via further restriction of root/privileged capabilities (eg, no CAP_SYS_NICE) and mandatory access control via LSM policies (eg, Simple Modified Access Control Kernel (SMACK)[52]—using labels attached to system objects, or SELinux[53]). Illustrated in Fig. 7.8, LSM provides kernel hooks that allow system calls to be evaluated against security policies. But the fact that multiple security schemes exist add even more of a burden in the continuing theme around choice and supportability in this chapter. These schemes have been historically challenging to operate, particularly the use of access-list (like) constraints in allowing the right level of security without constraining utility.

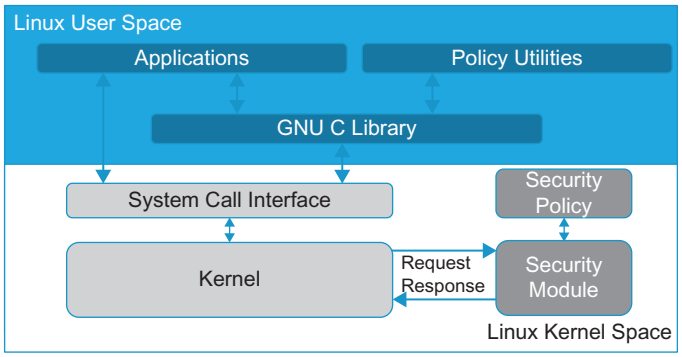

FIGURE 7.8

Linux Secure Module (LSM) framework.

These protections are equally applicable to VM environments and several vendors market "carrier-grade" or "hardened" Linux distributions that leverage these techniques, all of which should be evaluated against the previously-mentioned history.[54]

Just as with VMs, the container operation environment also requires security-focused vigilance. For example, there is a potential attack surface in the container image movement and verification (general operations for the environment).[55]

Not sharing

Beneath both hypervisor-based virtualization and containerization there exists the possibility of a class of bugs that allow for privilege escalation, allowing a malicious or suborned tenant to escape their private environment and gain access to the kernel or hypervisor, and from there other functions and possibly the rest of the network. This possibility feeds back into the "relative" security position of any shared host in the view of a security professional and potentially forward into the "reduced vulnerability surface" appeal of a minimized container-specific OS.

Looking forward, some of the networking optimization we need to discuss may also represent some tradeoff between security and performance. For example, the use of shared memory (as an IPC mechanism) has traditionally raised security concerns in the networking world. Many of the context-reduction techniques that adopt the use of shared memory (or, at the lowest level, shared processor cache) attempt to bound risk by limiting sharing to non-page-able memory and point-to-point (nonpooled) sharing.

NOT sharing might be the best advice.

That is, not allowing multiple tenants to share the same host. That does not mean the "end of virtualization" as a strategy for deploying NFV, but rather the adoption of the tenant segregation techniques employed by large cloud operators if your NFV solution is multitenant by design (eg, an NFVaaS infrastructure in which you allow resource slicing to enable multiple virtual providers).

SECURITY—THE LOWEST COMMON DENOMINATOR

Lying just below the surface of the argument around the relative security of the bare-metal, VM, container, or hybrid deployment model lurks a fundamental concern that everyone should have around NFV. This happens to exist for cloud computing environments as well. That is, whether shared computing systems and architectures can be *truly* secured or not, and whether or not virtualization, containerization or any other disaggregated system that runs on something other than a self-contained piece of hardware can be secured.

It has become clear that even on self-contained systems (ie, bare metal), that very low level components can be compromised including the BIOS, firmware, and hypervisor.[56]

Both Intel and the vendor community have responded by creating a trust model for cloud computing that starts with the hardware. The result will:

- Enable attestation as to the integrity of the hardware and images used.
- Enable compliance checking, policy, verification, and audit services.
- Enable hardware-based authentication of clients.
- Allow the provider to create trusted pools of compute that protect confidentiality and data integrity of the application.

Examples (nonexhaustive) include:

- Intel TXT, which is a hardware-based[57] solution that provides a traceable line of trust for the machine, its low level settings and the images loaded.
- Cavium provides a PCIE based "LiquidSecurity" Hardware Security Module to satisfy FIPS (Federal Information Processing Requirements) compliance and encryption needs (data protection) for government, finance, and other industries.

The ultimate solution to the problem of trust in virtualization will add even more complexity to the ETSI Management and Orchestration (MANO) components of the ETSI model particularly around resource management and VM placement policies, and change the fundamental requirements for NFVI.

CURRENT PACKET HANDLING

Processing packets in any network-centric software is a never-ending exercise in maximal forwarding performance optimization. This entails employing algorithms and "tricks" both in the firmware that executes on these devices and inherent in the hardware architecture. This task is made even more difficult when the hardware is not designed with these things in mind, as was the case with server network interface cards until a few years ago. Some of these "tricks" have unwanted side-effects, such as the use of poll-mode software drivers.

The IA uses optimizations that start with BIOS settings (NUMA, power mode, etc.), reaching through advanced NIC capabilities directly to the operation of the cache. For the best possible outcome, all of these optimizations (including their potential hardware dependencies[58]) are leveraged.

Using direct cache access (DCA is one of several accelerants that are part of Intel's I/O AT NIC technology), typical packet handling starts by directly writing packet info to the processor cache[59]:

- A processor RX descriptor write (a pointer to a filled buffer on a ring structure from the NIC, the size of which can also affect performance) is a trigger command on a DMA receive queue to begin the DMA transfer from a peripheral (NIC).
- The NIC reads a buffer (NIC descriptor read) and begins the DMA transfer; executing a header write, payload write, and status write all directly to the processor cache. This constantly churns the cache and the interplay introduces the need for coherency management (discussed later).
- The processor then does a complementing status read and header read from the cache. Then it does a payload read, which essentially is a zero copy operation.

If for some reason, there is no available cache line (ie, a cache miss) or the cache line is "dirty," indicating that contention or current use exists, the existing cache line will be "evicted" to clear the data for any current users. This leads to a subsequent access from system memory, which is a much slower proposition. This architecture is a bridge between the use of the cache and system memory. System memory is always updated regardless of the state of the cache. However, the trick is not to make the processor have to read from system memory very often because any such read incurs a decrease in performance.

The processor then makes a forwarding decision, implements features, does the header rewrite, outputs queue check, and adds this all to a TX descriptor (buffer). There really is no hardware transport management function (queue management) in this scheme. This would normally be handled by having an additional core resource assigned to a software forwarder handle this function.

Repeat. Repeat. Repeat.

Note that the provided acceleration (by reducing packet and packet descriptor access latency and memory bandwidth requirements) is just one part of the overall performance equation in getting packets from the NIC to the core and only applicable to receive side processing (transmit has a separate set of optimizations).

This description is for a single application on a host. Returning for a second to the NFV service-chaining model, it is easy to imagine the complications that may be introduced if and when more than one function in the chain is coresident on the host (and traffic has to traverse multiple VMs).

APPLICATION PROCESSING

The highest performing network applications will be micro-architecture sensitive (cache-friendly) and have few features (simple forwarding). Performance will be increasingly difficult to characterize as features are enabled (and additional CPU cycles consumed as we operate on the packet).

The performance required can be a bit daunting. To service a 10 Gbps connection, a service function would have to process approximately 15 million packets per second. While some vendors may claim that the average forwarding requirement is about 50 cycles per packet, that is largely applicable only to the most basic (and cacheable) L2 forwarding applications. Without some form of offload, prior knowledge that assists in prefetching[60] data or hardware acceleration, your mileage may vary.

In the compute/cache/memory sphere the application processing itself has a huge impact. Table 7.1 provides some back-of-napkin CPU cycle numbers for some common networking applications. (The point is not absolute accuracy, but relative number of cycles.) Your math may be aided by the knowledge that the generally available E5 family of processors is built for multiple applications and can clock in anywhere from 1.8 to 3.6 GHz. However, once you get into the 100s of cycles per packet, the ability to support 10 Gbps (per core) can be challenging.[61]

Table 7.1 Typical "First Generation" VNF Packet Processing Cycle Times. Firewall w/SSL Without Hardware Cryptography Offload

Application	Approximate cycles per packet[a]
L2 forwarding	Many tens (~70)
IP routing	Few hundreds (~175)
L2-L4 classification	Many hundreds (~700–800)
TCP session termination	Few thousands (~1500)
Stateful firewall	Many thousands (2000–3000)
Firewall w/SSL	Tens of thousands (~15,000–20,000)

[a] *YMMV, Your Mileage May Vary.*

BACKGROUND—CONTEXT SWITCH/DATA COPY REDUCTION

Setting aside the processor cycles required to actually act on a packet information (true compute—to actually perform the lookup and feature/application/function), a tax in the form of a context switch and data copying can be levied multiple times for every packet entering or leaving VNF, depending on how we have virtualized the environment.[62]

In general, a context switch will occur when we move between threads in a multitasking system, address spaces or protection domains (host-to-guest, kernel-to-user and the reverse or user to root and the reverse[63]), or handle external events (interrupts), exceptions, and system calls (these cause a trap to the kernel).

In the case of virtualized systems (particularly those virtualized with the Intel VMX technology), the major disruption occurs when a VM has to exit (VMExit). This occurs when the guest executes a privileged instruction, an I/O instruction, makes an access to the interrupt controller (APIC) or has to handle an external interrupt.

These actions are intercepted and lead to a save of the guest (VM) state into a control state or context memory, a load of host state from a similar memory, execution and then the reverse process. The operation can take many hundreds of CPU cycles. The TLB may be flushed and some indirect costs due to sharing of the cache may also be experienced.

Typically, when we virtualize a network device, the driver running in the VM puts a packet on a transmit ring and sets an I/O register (for this discussion, in the PCI address space). When the register is set, the software emulator receives a trap so that it can act on the contents. This trap or interrupt causes the context switch described above. This is just the first in a series of context switches and data copies on the way to the application in the guest OS (in a nonoptimized system).

By tracking the virtualized Task Priority Register (TPR), the VMM is supposed to know when it is safe to assert an interrupt. When a guest writes to this register, the VMM will cause a VMExit.

In general, optimizations attempt to bypass or augment the virtualization layer of the hypervisor and replace it with more sophisticated techniques. The performance gains in eliminating these perturbations can be greater than an order of magnitude in CPU cycle savings.

It would be preferable if we never had to do a data copy (possible, but with caveats). Zero-packet-copy claims rely a lot on the driver and on the implementation of some of the virtualization underpinnings. For example, the author of the hybrid virtualization project (described previously) found that virtio[64] in qemu/kvm 1.7.0 was not "precisely" zero copy because it creates a small fragment and then "pages in" the rest of the frame (from host to user space). The fragments persist and are returned by the same methodology to the host. If the driver does not fully support the fragmented frame API on the qemu side as the frame is being paged back, then the frame will be reassembled into a temporary buffer and passed as a single frame, not zero copy. In this release, no driver supported this paradigm in both directions (tap and hub have unidirectional support[65]).

This also meant that whatever you are running in user space that converses with kvm has to be able to operate on fragment lists shipped in the "iov" format. If not, a copy is required. The same limitation holds true for any application relying on legacy kvm kernel compatibility (eg, kernel defragmentation services).

It is important to note that the optimizations to eliminate packet copying and eliminate, coalesce or bypass the impacts of interrupt handling for network I/O are a continually moving target.

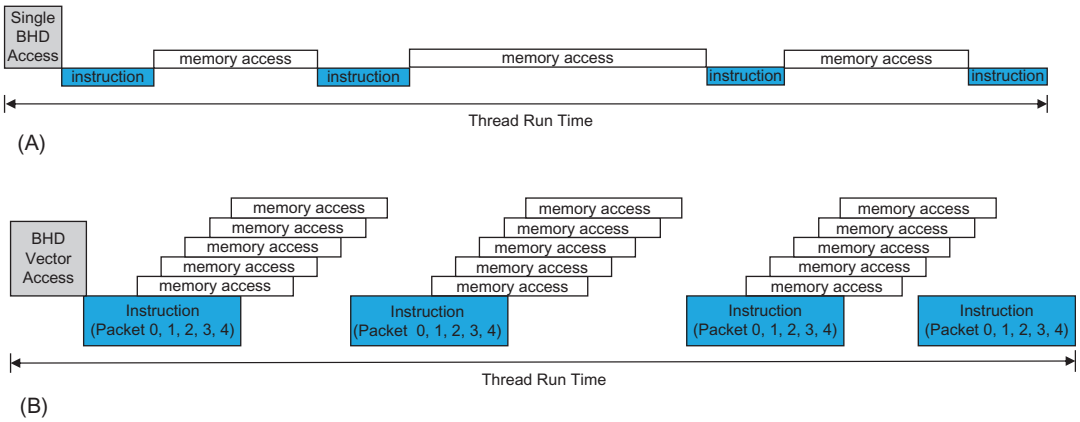

FIGURE 7.9

Vector processing amortizes load/use memory access latencies in (A) sequential code with nonoverlapping memory accesses and instructions by (B) loading a group/vector of packets and working on them simultaneously.

BACKGROUND—SCALAR VERSUS VECTORIZATION

Serialization of the packet transfers through context switches is an addressable problem. By putting multiple packets into a transmit buffer (safeguarding against triggering any traps) we can amortize the context switch over a number of packets. This switches the scalar operation to a vector basis, processing more data with fewer instructions and creating software parallelism (pipelining).

In Fig. 7.9, if the latency between the execution of the second and third group of instructions in the scalar example (A) is 100 ns, then the amortized value would be less than 20 ns in (B).[66] This is the root innovation behind some of the more advanced forwarders on the market. Efficiency improves with the size of the vector. In some cases, vendors have demonstrated a 256× improvement (essentially, the scale of the packet vector).

"Vectorization" can have a positive secondary effect on cache efficiency (see Chapter 8: NFV Infrastructure—Hardware Evolution and Testing).[67]

ONGOING—INTEL ADVANCEMENTS (AND ACADEMIC WORK)

Intel has been acutely aware of network performance optimization for some time and actively are promoting acceleration through performance studies (which upstream their findings into new versions of DPDK and platform enhancements), recommended best practices and exposure of architectural performance changes/enhancements as new hardware becomes available, through open source tooling to make management and orchestration of their multiple generations less complicated and optimize-able.

Acceleration for packet transfers from NIC to guest started with changes to the I/O driver, which evolved from native trap and emulation, to include support for pass-through operation (SRIOV or PCIE direct) and virtio based paravirtualization.

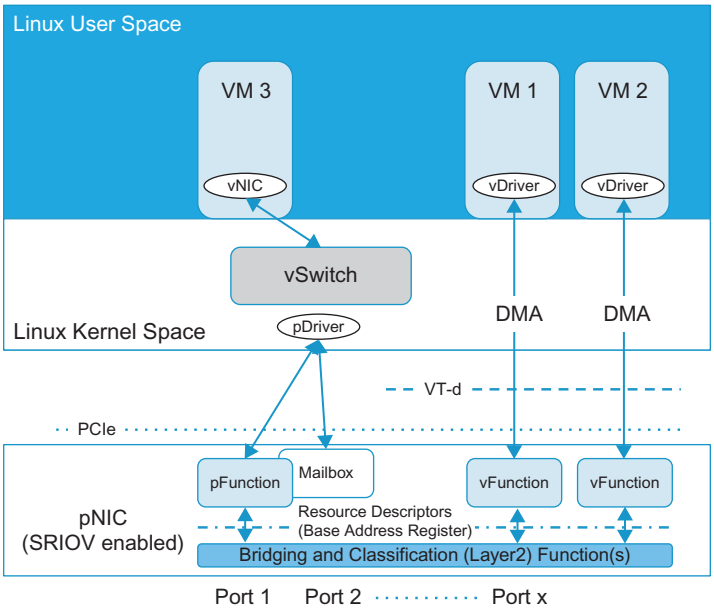

FIGURE 7.10

Intel VT-c and VT-d leveraged to support direct I/O into a VM (non-SRIOV supporting VM routes through kernel driver and hypervisor vSwitch).

Virtio-based solutions are evolving (recently from vhost-net to vhost-user) to shared-memory rings using large pages and the DPDK driver—bypassing the host kernel. The host stack is the last big bottleneck before application processing itself.

Intel offers a number of virtualization technologies in this area branded under the "VT-" umbrella (Fig. 7.10).

VT-x—added the previously-mentioned instructions specific to virtualization, starting with the VMX to the instruction set that enabled the VMM.

Intel VT FlexPriority support—removes the need for the VMM to track writes to the virtualized TPR, reducing the number of VM exits on context switches.

VT-d—(Directed I/O) allows the direct assignment of I/O devices to a specific guest via a dedicated area in system memory. VT-d performs address translation (and ensures separation of memory pages in different VMs) for DMA transactions between guest and host physical address. Smaller page sizes (4 KB) can result in inefficient mappings (lower coverage of the memory space in the TLB). A resulting miss can interrupt processing while the appropriate space is mapped and stored. VT-d addresses this with Extended Page Tables (EPTs) supported in the guest (1 GB). These techniques reduce copy-related behavior (staging of the data in separate host memory while performing the transfer) and are supported via a dedicated IOMMU.

VT-c—supports both VMDq (Virtual Machine Device Queues) and PCI-SIG SR-IOV (Single Root I/O Virtualization). VMDq is a NIC-based offload of some of the functionality common to the hypervisor (the hypervisor is a cooperative partner[68]) to enhance QoS and routing packets to VMs (queue mapping from hypervisor to NIC hardware queues).

SR-IOV—allows PCI devices that support these functions (eg, NICs and co-processors) to present themselves to the host as many virtual functions[69] that can be mapped directly to VMs (with varying bandwidth allocations). Its principal goal is:

to standardize on a way of bypassing the VMM's involvement in data movement by providing independent memory access, interrupts, and DMA streams for each virtual machine.[70]

SR-IOV allows sorting of the application traffic by tag, thus partitioning the resource and mapping them into the guests, eliminating copies that might have been incurred in the host to send/receive the packet. However, SR-IOV does not emulate the port or cable (the individual logical links owned by the applications running in the VM), so there is some management information loss in the vNIC emulation.

SR-IOV can be considered an improvement over its precursor (PCI Device Assignment), which lacked vfunction mapping and thus dedicated the whole device to a VM/guest.

This optimization still places a burden on the DMA engine/controller (it uses Direct Memory Access to move the data).[71] Additionally, SR-IOV places some constraints on the movement of traffic between VMs on the same host, and thus needs augmentation/combination with non SR-IOV data planes (virtio based or entities like a DPDK-enabled vSwitch). And SR-IOV requires driver support in the VNF, which may be considered an operational burden if the driver needs to be continually updated/enhanced.[72]

Intel has a progressive strategy for minimizing memory copies involved both with virtio and SR-IOV data planes that include improvements to routines (a vectorized version of memcopy called rte_memcopy[73]), the potential to offload copies to dedicated cores (which is aligned with their general exploration of "pipelining" solutions for higher throughput) and the development of new/stronger DMA engines.

Intel's DPDK libraries[74] (Fig. 7.11) enable a VM to take over network interfaces from the host OS using a PMD.

DPDK is a soft data plane that reduces context switches (PMD eliminates kernel interrupt context switching), reduces locks on queue structures (or optimizes them), and allows threads to be pinned to cores allowing packet processing to be isolated to a few cores. DPDK works well for fundamental frame movement but may be a more difficult programming model for complex features. To accelerate its use, Intel contributed an open source DPDK-enhanced version of Open vSwitch (initially their own version via 01.org but planned to converge with ovs.org at version 2.4.0—see drawing and explanation in Chapter 1: Network Function Virtualization). Communication is enabled through the use of shared memory ring structures between switch and VMs, its PMD, and the mapping of PCIe device base address space in host memory.

DPDK is the minimum/de facto optimization strategy for the VM and Intel oriented NFV architecture.

Intel's commitment to ongoing DPDK development should smooth out most of the negative observations around DPDK—adding interrupt support in addition to poll mode and changing threading operations (core mapping, thread type, and eliminating explicit declaration) to broaden DPDK adoption.

In fact, DPDK can be used in both the vSwitch data plane and/or the application data plane (examples of this arise in user space soft forwarders); other soft data plane proposals exist.[75]

These soft forwarders (with varying feature functionality) take similar advantage of DPDK and run in user space, relegating the vSwitch in the kernel to the role of patch panel—eliminating a great deal of overhead in exchange for the memory (normally a small footprint) and vCPU required to host its VM (illustrated in Fig. 7.12).

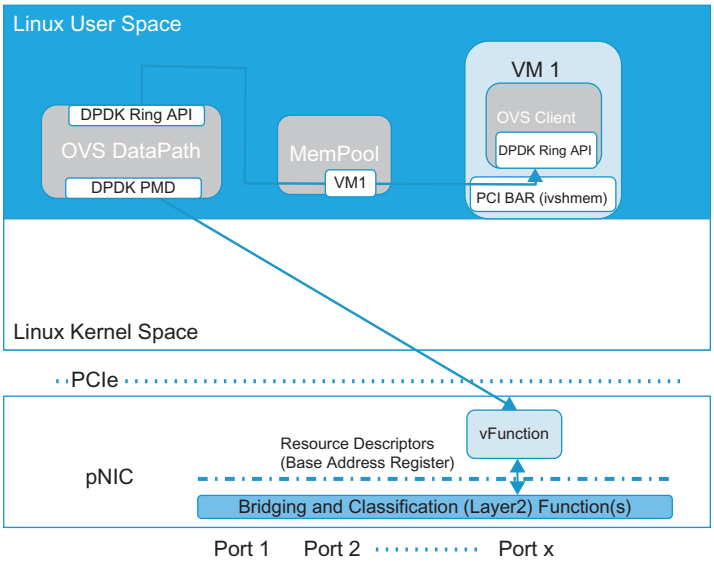

FIGURE 7.11

Intel's DPDK-enabled Open vSwitch, moves the switch construct to Host User Space.

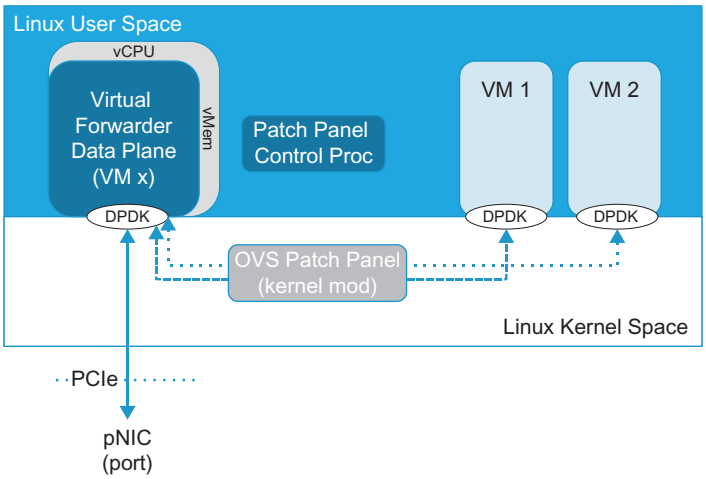

FIGURE 7.12

Alternative configuration of soft forwarder in user space and leveraging a thin driver in kernel (for the patch panel functionality) with a controller in user space.

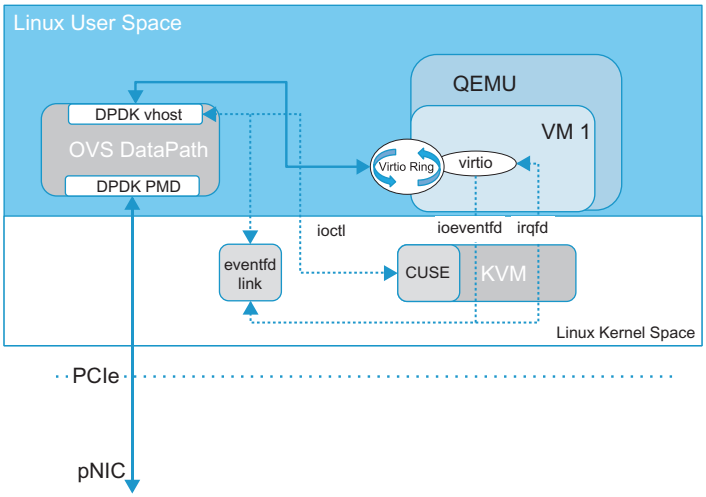

FIGURE 7.13

Intel's user space vhost implementation (errata—CUSE block represents the IO control routines for character devices).

In latter versions of DPDK (1.7), Intel attempted to accelerate the VM to VM and VM to physical connections through two mechanisms, both of which exploit a logical/virtual network device and user space forwarder: dpdkr/ivshmem (DPDK rings and shared memory) and us-vhost.

dpdkr/ivshmem uses host shared memory, making it less suitable for untrusted tenants. It also does not support live VM migration of the VNF but promises "zero copy" performance.

The us-vhost approach (Fig. 7.13) currently promises fewer data copies and context switches, but has the benefits of not requiring the VNF to be DPDK enabled (using the virtio driver—no new drivers required) and supports live migration.

The user space forwarder in both cases is the Intel provided DPDK enabled Open vSwitch, which uses a special netdev interface that bypasses the kernel. In the us-vhost case, the DPDK driver has capabilities similar to the kernel-based vhost-net (but in user space)—handling virtqueue kick events (on ioeventfd signals for guest-to-host and irqfd for host-to-guest) to put/pull data onto shared memory segments forged between the vSwitch and the guestOS (VM).

While the DPDK components include example applications, there is room for differentiation in custom forwarders (eg, feature completeness, performance at scale).

Intel typically publishes very high performance numbers on DPDK optimized platforms around low feature (L2 forwarding)—latest (version 1.7) claiming 59 Mpps on a core (at this rate, per packet time is 16 ns). While absolute numbers may be highly optimized, relative performance increases from release to release are generally believable and the CPU cycle reductions shown (greater than an order of magnitude over Linux stack) point to a reduction in context switching. Intel also introduced vectorization in version 1.3 (version 1.7 boasts intrinsic vector RX and TX support).

NETMAP AND VHOST-USER

There are other methodologies for addressing the VM to VM and VM to P-NIC transfer overhead than the DPDK methods (although they use similar constructs).

Netmap[77] is a memory buffer-based local network abstraction between applications. It consists of a kernel-based shared memory segment (ring) between the applications/guests (or application and host stack). Applications can pull/push packets onto/off of the ring without a context switch (it is explicit in the poll/select signaling). With netmap, a user space data plane can directly access memory/buffers (zero copy).

Synchronization is fairly simple. For the receive transaction, netmap uses a poll API and on transmit the writer will update the curser (new frames) and invoke ioctl to trigger a flush.

While it is possible for one application to inadvertently dirty the shared memory area, it is allocated from a restricted non-pageable area, and device registers and critical kernel memory are not exposed.

It is also possible to allocate separate pools for each virtual connection for further isolation and security. Netmap has a "pipe" mode that allows applications to directly connect via a shared memory ring (without Vale), using the kernel only for synchronization (again, via poll/ioctl).

Netmap can be used between applications (as pictured in Fig. 7.14), between the NIC and application/user space[78] (not pictured, we show DPDK as a common implementation in combination with Netmap) and between the application and the host stack (though for most networking applications bypassing the host stack is common).

Vale[80] (Fig. 7.15) is a complimentary system in the kernel that maps these netmap rings to each other. Vale scales the abstraction to multiple applications on the host.

Techniques like netmap can sometimes become dangerous in combination with test routines that are designed to exaggerate performance. An application distributed with netmap (pkt-gen) can generate 80 Mpps, but (typical of many speed-test applications) it can be used to set up a test that only manipulates rings of pointers to packets. Such unrealistic test scenarios can produce results unrelated to reality.

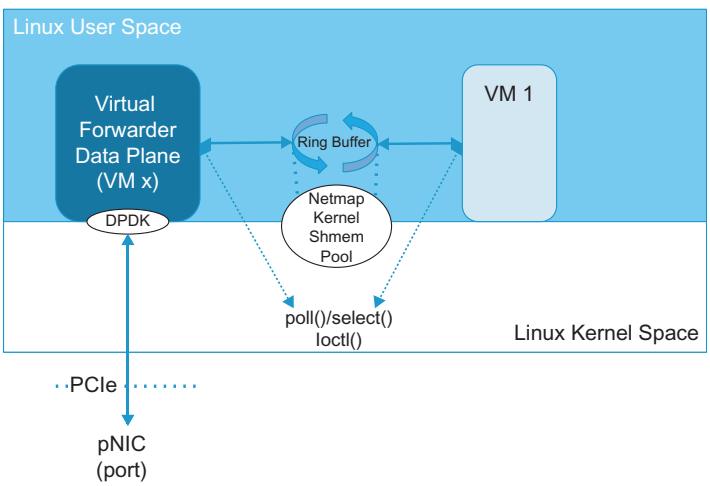

FIGURE 7.14

Netmap.[76]

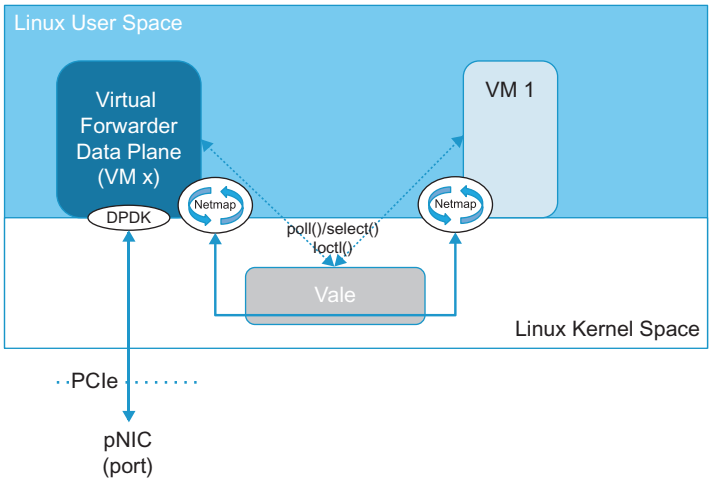

FIGURE 7.15

Vale.[79]

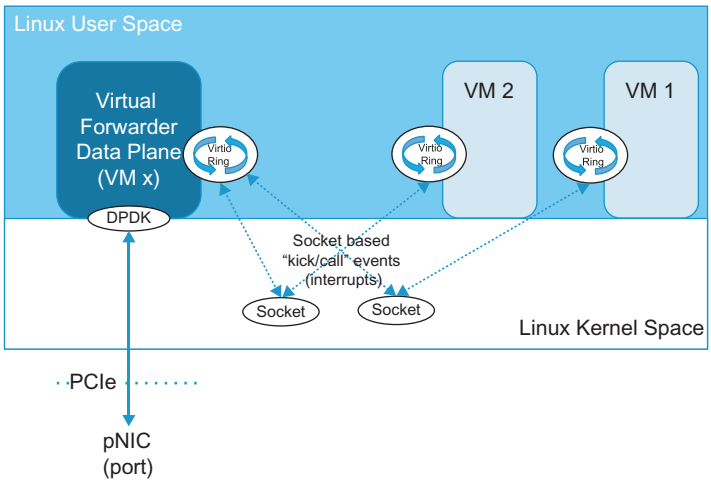

FIGURE 7.16

A virtual forwarder in user space leveraging DPDK and vhost-user.[81]

A generic forwarder in user space can also leverage user space vhost (vhost-user[82]) to forge user space shared memory rings between the guests and forwarder (as shown in Fig. 7.16), with the only remaining kernel based function being (socket) control, minimizing data copies.[83]

Using these methodologies in conjunction with a vector-based forwarder running in user space, we dramatically reduce the number of context switches to create a service chain (essentially one context switch for X-sized vector of packets traversing a virtual service chain on the box).

Software packet handling bottom line

The bottom line here is multipart:

There are virtualization connectivity techniques that can increase the performance of NIC to application I/O as well as techniques that can dramatically improve service chain (app to app) transfers within the same host. The latter is a consideration in light of the performance implications of the current focus on traversal through the hypervisor switch.

There are currently three types of virtual intraserver links involved in these optimizations: point-to-point (eg, vhost-user), vSwitch, and DMA (eg, NIC to VM). These are "bandwidth-less," but do consume shared resources (CPU cycles and DMA controller capability).

A promising combination is emerging that includes a NIC driven by DPDK, the VMs driven by vhost-user with no changes needed in guest (virtio net driver), and a vector-based virtual forwarder in host user mode. Performance of this model is expected to map well to containers.

DPDK is continually evolving with a roadmap already stretching beyond a 2.0 version.

Intel acceleration techniques like SR-IOV are not the only ones available, but tend to dominate as Intel presses their Open Network Platform (ONP) initiative. We focused on DPDK as part of an all-Intel reference platform. Techniques like Remote Data Memory Access (RDMA)[84] have been evolving in the High Performance Compute environment, but have yet to achieve the critical mass of SR-IOV (RDMA on DPDK is a potential future merge of technologies).

There are also implications to the uses of these technologies in service chaining. Whereas the switching from VNF to VNF using a vSwitch has platform and vSwitch bandwidth dependencies, switching to or between VNFs using direct-attachment techniques like SR-IOV can consume PCIe bandwidth and network controller resources as well.

Postoptimization, the main bottleneck shifts (yet again) to the guest Linux kernel and the service applications themselves.[85] The first order requirement of the functions would be to become vector-aware for data transfers.

Almost all of these techniques require support/awareness in orchestration (OpenStack), emulators (QEMU) and often library changes (eg, libvirt), creating dependencies of their own.

Finally, there are some applications that virtualization optimization techniques cannot completely address. A "true" Traffic Manager (verifiable QoS) is a (potentially) good example. Many VNF implementations that require "true" QoS will implement an additional core as a transport manager (TM)—creating a more sophisticated pipeline.

TURNKEY OPTIMIZATION (IS THERE AN EASY BUTTON?)

Given the possibilities for optimization and its ongoing architectural evolution (see Chapter 9: An NFV Future), Intel has cultivated an ecosystem of custom integrators,[86] some of which (notably 6Wind and Wind River) offer turnkey variants of the Linux OS with their own optimizations for the NFV VM environment.

Wind River[87] is an Intel company offering what their marketing calls "Open Virtualization Profile" that highly leverages Intel VT technologies (and DPDK) and purports an adaptive performance capability that allows the operator to prioritize throughput or latency.

They offer core pinning (for VMs) with some enhancements to enable dynamic core reassignment, NUMA awareness and isolation of potentially competing workloads, features that are pointed toward more intelligent orchestration. The environment also claims a greater degree of interrupt

control and latency reduction that helps with context switching (VM exits)—notably by the passing of virtual interrupts to VMs and supporting EPTs that allow the guest OS to handle its own page table modifications and faults (again, leveraging VT technology). In short, they have incorporated the entire suite of Intel enhancements on Intel hardware.

The Wind River solution includes a proprietary optimized vSwitch and Linux distribution. While guest VNFs can benefit from the tuning of the basic offering without modification, incorporation of their driver in the VNF is required for the full benefit of the solution.

Wind River has put forward an ETSI PoC for the mobile RAN that accentuated the combination of interrupt reduction, SR-IOV enablement (Intel Ethernet adapter) and QuickAssist coprocessor support (for signal processing acceleration).

Many proposed "carrier grade" enhancements to KVM are (rightly) upstreamed over time. Other "carrier grade" NFV enhancements dealing with VNF management functionality are packaged as OpenStack plug-ins. This seems to place some of the functionality typically associated with a VNFM product, into the VIM (in ETSI terms).[88]

The 6Wind[89] 6WindGate solution (Fig. 7.17) focuses on host-to-guest bandwidth, vSwitch performance, and poor performance of standard OS networking stacks. The solution leverages DPDK (on Intel) and a 6Wind network stack (adding telecom-specific encapsulation support—eg, PPP, GTP).

Some of the stack modules are "FastPath" modules that can run on dedicated cores (scaling with the addition of cores) as FastPath Dataplane (eg, for TCP session termination, they go to parallel DPDK cores). Outside of FastPath, 6Wind provides some applications they call Transport services (eg, Routing). These address perceived shortcomings in the default Linux stack and some of the application cycle overhead noted earlier.

6Wind also includes their own Open vSwitch module that can be run on multiple cores and their own shared memory driver that is incorporated in both host and guest.[90] The specialized vSwitch points out what 6Wind sees as a problem with the standard OVS—performance does not scale linearly with additional cores (while the 6Wind accelerated switch claims to do this) and is available as a standalone product (6Wind Virtual Accelerator). Wind River similarly has a specially adapted vSwitch component.

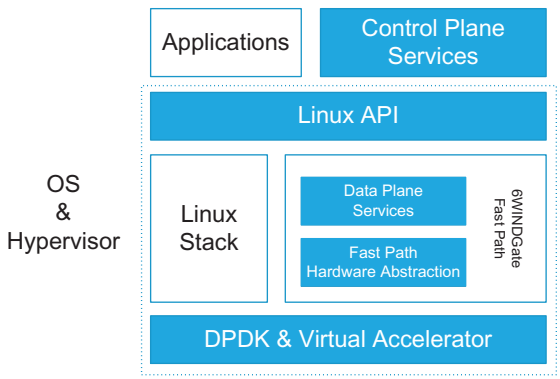

FIGURE 7.17

General depiction of 6WindGate architecture.

Turnkey environments offload the environment optimization problem (to some degree) to a vendor. Like the previously discussed optimizations, they can be architecture-specific (predominantly Intel). They do have their trade-offs:

- From a technology/business perspective, the operator depends on the vendor to follow the evolving software/hardware optimization curve (staying current)—potentially including support for evolving hardware that will get integrated into the architecture (looking forward to heterogeneous cores, etc.). This includes the support of containers and bare metal (which may create a tendency to mix solution pairings, as the optimized hardware virtualization provides little benefit).
- Generally, a high degree of vertical integration might limit the size of the ecosystem (third party products available for solutions).
- From a deployment/operations perspective, the operator may have to support an additional and specialized distribution of the Linux OS for network functions.
- From a development perspective, some turnkey environments may require augmentation of the development tool chain (eg, compiling a vendor's shared memory driver into your VNF/guest to take enable the advertised performance boost).

All of these need to be considered against the operational transparency NFV purports, and the significant amount of true open source that might eventually dominate the VIM and NFVI.

FD.IO (NONE OF THE ABOVE?)

By now you might have noticed a bit of a problem. Performance and functionality of virtual switches in the VM architecture have been bit slow to evolve and the forwarding paradigm for containers might be a bit immature. Proprietary solutions to the former seem to have fairly big trade-offs.

The Linux Foundation multiparty and multiproject fd.io (Fast Data) was launched to unify the approach to the data plane services. The scope of the project includes I/O, packet processing, and management agents.

As part of the project launch, Cisco Systems announced open sourcing of their Vector Packet Processor (VPP)[91] code, which may blunt the appeal of any proprietary accelerated vSwitch and provide a more suitable forwarder for the container environment in the short term and provide an extensible platform for other projects in the future.

VPP moves forwarding completely out of the kernel, for both containers and (because it can be configured to perform like a vSwitch) VMs. With a promised Neutron plug-in, it can readily replace OVS in OpenStack.[92]

Because of its independence from the kernel, the forwarder can be reset and return to forwarding in seconds without losing the electronic state of the device.

The basis of the VPP code has been used in some of Cisco's routing products (since 2002) and is arguably the progenitor of the vectorization techniques mentioned earlier. VPP acts as a user-space host router with stack, bypassing the need to use the host network stack in the kernel and unifies all IO (storage and peripherals) into a single platform.

A very high performer, the forwarder is capable of 14 Mpps/core and scales linearly with cores, supports extremely large tables and can run within an application container or in a container of its own.

VPP supports DPDK directly to (dVMQ) queues on the NIC as well as SR-IOV (though DPDK 2.0 is expected to replace SR-IOV over time).

Its modular code base allows developers to easily hang additional functionality off of the stack or support hardware accelerators (eg, encryption engines). VPP also supports flexible binding to a Data Plane Management agent, creating a great degree of deployment flexibility.

VPP has flexible plug-in support (Fig. 7.18). Plug-ins can alter the packet processing graph, introduce new graph nodes, and be added at runtime.

The project promises a CD/CI environment with a continuous performance test lab (to identify code breakage and performance degradation before patch review).

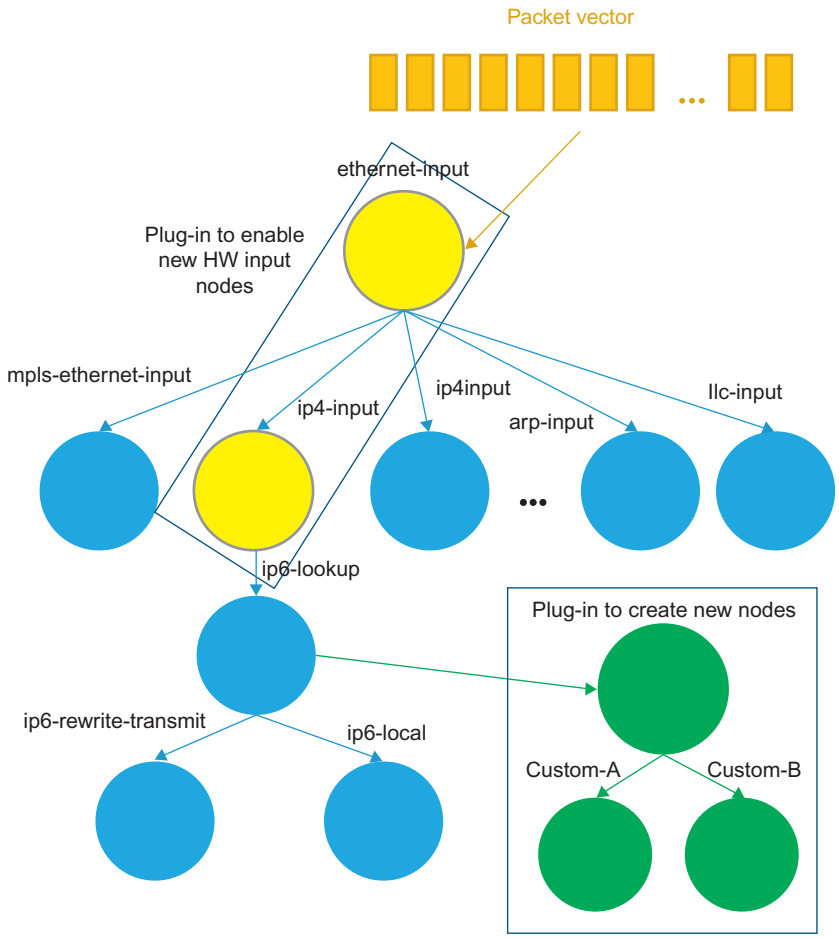

FIGURE 7.18

VPP supports a plug-in architecture.

CONCLUSIONS

It is hard to compress a discussion of virtualization performance issues and optimizations. This software-based component overview presents a portion of the future possibilities of high-performance NFV.

Awareness that performance can be an issue for NFV is good, but compulsion concerning performance details may be detrimental.

The focus on performance of a particular VNF or infrastructure component should not be a distraction from the broader and more long-term benefits of service creation.

Like the appliance market before it (which was already beginning to divert more towards generic processing platforms), the NFV infrastructure market will go through a continual innovation/optimization cycle. The network I/O performance of general compute architectures has progressed tremendously in the past two years and will continue to evolve. Vendors like Intel have a vested interest in raising awareness of these improvements and making them available as quickly and transparently as possible in orchestration.[93]

If innovation is a given and transparency inevitable, the NFV focus should be around integration and management of multiple generations of hardware capabilities.

The fluidity of NFV performance environment is typical of any early-stage phenomena like NFV. For now, some trends/patterns are emerging:

- *The demands of network I/O and attendant optimizations (pinning cores and mapping them to NICs, dedicating cores to virtual forwarders, dedicating cores to TMs) make it unlikely that virtual network appliances will be easily mixed with other applications in a general datacenter approach.*
- NFV systems will have multiple "generations," just like their integrated appliance predecessors (both in software and hardware). As we pursue performance optimization, we cannot forget the tie to the orchestration to properly align resources and avoid unnecessary bottlenecks in the overall solution architecture.
- Some of the more likely deployment architectures circulating for NFV in Telco environments show a "common forwarder" demarcation point between the service specific function and the network—a service-agnostic layer. This keeps "network operations" a traditionally separate entity (even in the virtual world) for debugging, performance monitoring, and provisioning. How well this approach integrates with evolving SFC/NSH (see Chapter 6: MANO: Management, Orchestration, OSS, and Service Assurance) remains to be seen.
- VNFs can approach the throughputs of dedicated network appliances, but (based on current trends) they will become hotter as they become more specialized for network I/O. Power efficiency will have to be taken into careful consideration at the solution architecture and design levels. The use of PMDs will have to accommodate some power savings potential.
- The VM-centric NFV architecture may not be the only packaging model going forward for all environments. MicroVMs, containers, and bare-metal all are emerging as alternative "packaging" models with varying risk/cost/benefit profiles. Security is only starting to emerge as a gating factor for shared infrastructure projects, and can be a key determinant in the packaging decision. "Compose-ability" of functions (a next generation potential for NFV enabled by SFC/NSH) may also be influenced by the packaging decision.

The potential shift in NFV from generic cloud compute toward network I/O optimized virtual appliances raises fundamental questions about some of the original premises behind NFV.

In particular, do the economics really work for NFV when the costs of specialization and power/Gbps are factored (now or in the future)? If not, can the overriding attraction of service creation, operational agility, and elasticity in NFV make the complications of specialization acceptable and keep NFV attractive (until the economics DO make sense)?

While it seems an assumption that the security risks associated with NFV are outweighed by the agility gains, the extent of that risk needs to be better understood and controlled. Security in an NFV environment should drive new opex spending, which has to be weighed into total cost in the balance of NFV.

There is an "easy button" being marketed, with a number of concessions to an entirely open and generic system. Organizational politics are often involved in the "easy button" decision. A traditional Data Center operator would forgo this level of customization and rely on the economies and availability of bulk compute services (and wait for these customizations to become part of the mainstream "COTS" purchase if they have long-term positive ROI). For a traditional Service Provider that offers wire line or wireless services, there may be political motivations to continue to own the delivery of services and deploy their own "service pods"—making the "easy button" attractive (reducing the need for inhouse virtualization expertise and giving the customer "one throat to choke" for NFV).

Additionally, the "carrier grade"-ing of solutions, while sympathetic to the concerns of traditional operators (particularly telcos), ignores the "pets" versus "cattle" thinking common in large-scale data centers. While some services might need some of the protections implied in "carrier grade," there is some substitution in terms when talking about software based services. VM "migration" for HA and "six 9s" may be replaced by the "service availability" and "resiliency" designed into today's web-scale applications instead of being baked into more expensive infrastructure.

The proliferation of NFVI-related optimization variables today may also be a short-term problem as the market determines the proper optimization tradeoffs going forward and/or as consumers opt again for the original NFV proposition (by deploying only what is commonly available or nonspecialized). This may sacrifice some short-term potential performance for longer-term operational simplicity.

The fd.io project may provide an open source consolidation of ideas and techniques that refocuses the disparate attempts to tune the data plane for the NFV environment.

In our next chapter, we will look at how hardware innovation (to solve the network I/O problem) exacerbates the potential complexity problems of NFV deployment and the need for standardized testing that is being driven by both hardware and software innovation cycles.

END NOTES

1. While general-purpose processors can include the Intel x86, ARM, and PowerPC architectures (and potentially the newer IBM/GlobalFoundaries OpenPower/Power8), the current de facto NFV target is the Intel x86. For the greater part of the performance chapters, the Intel XEON SandyBridge will be the exemplar because it is currently widely deployed.
2. *The microarchitecture of Intel, AMD and VIA CPUs*, by Abner Fog, Technical University of Denmark, 2013.

3. VMM—VMM is a layer of system software that enables multiple VMs to share system hardware. The applications can run without modification. The VMM runs in a special mode (VMX root is fully privileged while VMX nonroot creates the basis for the guest to run in Ring-0 deprivleged). The guest OS runs in Ring 0 deprivileged. Applications in the guest run in Ring 3 deprivileged.
4. http://www.linux-kvm.org, Kernel-based Virtual Machine.
5. QEMU is "a generic and open source machine emulator and virtualizer" (see http://wiki.qemu.org/Main_Page). QEMU can perform hardware-based virtualization.
6. http://docs.oasis-open.org/virtio/virtio/v1.0/csd01/virtio-v1.0-csd01.html.
7. There are other differences in the appearance of the memory space of the guest.
8. Introduced with the Xenon product family.
9. Note that scheduling happens outside the vSwitch context, which has implications on behaviors if the physical switch is oversubscribed (VM thinks it sent and is unaware of drops).
10. TLB is the physical-to-virtual memory mapper of the MMU.
11. Commonly by using an abstraction layer like *libvirt* (http://libvirt.org/).
12. The ETSI model also allows for "compound" VNFs, which may represent a complete service. Compound VNFs did not seem to provide the level of independence for application development and deployment that were goals of the ETSI architecture (as described by the end of Phase 1).
13. In some ways, this would mirror the Kubernetes distribution approach (pods) for containers.
14. While VM-to-VM connectivity of the time was not impossible (bypassing the vSwitch) it had accompanying marshaling and call overhead.
15. http://www.cc.gatech.edu/~qywang/papers/JackLiBigdata13.pdf is a good example of one of these studies. They have some interesting observations in Increasing Processing Parallelism and VM Placement (for Hadoop).
16. https://wiki.openstack.org/wiki/Ironic.
17. To realize some of these cross-operator use cases, federation of the NFVI and Orchestration systems might be required—an extremely "long pole" in the deployment of NFV.
18. http://user-mode-linux.sourceforge.net/.
19. *An Updated Performance Comparison of Virtual Machines and Linux Containers* Felter, Ferriera, Rajamony and Rubi <http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf>.
20. Non-Linux container environments do exist (eg, Solaris/Illumos zones, FreeBSD jails). The focus on Linux merely cuts down on an already large amount of environmental variance for our discussion.
21. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.
22. https://www.docker.com/.
23. This is simply and clearly explained in the blog, http://tuhrig.de/layering-of-docker-images/.
24. https://github.com/jpetazzo/pipework.
25. http://www.socketplane.io/.
26. http://weave.works/.
27. http://kubernetes.io/.
28. http://www.redhat.com/en/about/blog/red-hat-and-google-collaborate-kubernetes-manage-docker-containers-scale. There are numerous additional tools that integrate around Kubernetes (eg, Mesos and Marathon).
29. http://blog.docker.com/2014/12/docker-announces-orchestration-for-multi-container-distributed-apps/.
30. http://blog.kubernetes.io/2016/01/why-Kubernetes-doesnt-use-libnetwork.html.
31. https://www.joyent.com/company/press/joyent-announces-linux-branded-zones-and-extension-of-docker-engine-to-smartdatacenter. Joyent is based on Illumos/Smartos zones.
32. http://www.coreos.com.

33. There are some hidden benefits to the smaller footprint as well if management schemes (including HA) require "snapshots" of memory to be taken.
34. https://www.opencontainers.org/.
35. http://martinfowler.com/bliki/ImmutableServer.html.
36. https://clearlinux.org/features/clear-containers.
37. http://osv.io/.
38. https://mirage.io/.
39. http://unikernel.com/.
40. https://medium.com/@darrenrush/after-docker-unikernels-and-immutable-infrastructure-93d5a91c849e-.cg821bgpl.
41. http://www.cisco.com/web/HR/ciscoconnect/2014/pdfs/QvBN-TechnicalOverview.pdf.
42. A microVM uses hardware-based virtualization with a very limited privilege based on task-based security and introspection. For an example, see Bromium http://www.bromium.com/products/our-technology.html.
43. User Mode Linux allows the operator to run a software-based secure VM (Linux in Linux). It is a Type II VMM equivalent (with a paravirtualized relinked kernel).
44. http://git.qemu.org/?p = qemu.git;a = history;f = net/l2tpv3.c;h = 21d6119ed40eaa6ef38765c5a4a22b3b27dbda98;hb = HEAD.
45. http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4503740/pdf/pone.0130887.pdf.
46. A Xen hypervisor bug (disclosed in October 2014) required a number of reputable cloud computing providers to patch and reload systems (http://xenbits.xen.org/xsa/advisory-108.html). A KVM and Zen QEMU-related vulnerability (May 2015) https://access.redhat.com/articles/1444903 shows we may not be finished with possible escapes. Later failures of VMware's ESXi (printer hack), and other hypervisor failures have also paraded through the news recently.
47. http://www.bromium.com/sites/default/files/wp-bromium-breaking-hypervisors-wojtczuk.pdf is a good quick read to put the "relative" into your own view of virtualization security.
48. View http://seclists.org/oss-sec/2015/q2/389 and judge for yourself.
49. http://blog.docker.com/2013/08/containers-docker-how-secure-are-they/.
50. https://blog.docker.com/2013/08/containers-docker-how-secure-are-they/.
51. https://blog.docker.com/2016/02/docker-engine-1-10-security/.
52. https://www.kernel.org/doc/Documentation/security/Smack.txt.
53. http://selinuxproject.org/page/Main_Page.
54. The concept of Carrier Grade Linux has been around for some time and many of the protections/recommendations therein have been integrated into mainstream Linux versions (caveat emptor).
55. https://titanous.com/posts/docker-insecurity.
56. http://www.wired.com/2015/03/researchers-uncover-way-hack-bios-undermine-secure-operating-systems/.
57. It should be noted that Day Zero vulnerability in Intel architecture allowed a somewhat specialized exploit (the user needed access to Ring 0, which is privileged) of Ring 2 (System Management Mode) that would theoretically work around this level of security (http://www.bit-tech.net/news/hardware/2015/08/07/x86-security-flaw/1).
58. This is one of the funny paradoxes surrounding NFV. While most consumers want to use "open" and non-specialized hardware/software to avoid vendor lock-ins, they also seem to want the highest performance possible—which leads (at times) to deploying less-than-open systems.
59. http://web.stanford.edu/group/comparch/papers/huggahalli05.pdf provides a study of DCA with good descriptions.
60. Prefetch hinting was introduced (as part of DCA) to improve on write cache coherency transactions and reduce cache misses.

61. The approximate budget per packet would be less than 200 cycles for 10 Gbps. Naturally, moving to 40 or 100 Gbps drastically impacts that budget.
62. The simple packet processing routine described at the beginning of the section already contains some optimization (DCA).
63. There are OS dependencies here, so this is a generalization. The reaction to these perturbations can also be both OS- and architecture-dependent. For example, Intel provides both a hardware- and software-assisted functionality for domain crossing context switches, with the software-assisted method generally used to increase portability.
64. http://wiki.libvirt.org/page/Virtio.
65. The developer in this project was attempting to implement what was missing from qemu tuntap to see if they could improve this situation.
66. Locks are still required for multiprocessing.
67. The use of a packet vector reduces instruction cache thrashing.
68. VMDq works with VMware NetQueue and Microsoft Hyper-V Virtual Machine Queues. The combinations of hypervisor version and vendor NIC support should be researched if you want to use these features. Like SR-IOV, VMDq has resource limits—here, on the number of queues per port—that limit per-adapter scale for the function.
69. As a typical example, (in theory) a four port SR-IOV NIC can present four physical functions, each mapped to 256 virtual functions—for a total of 1024 virtual functions. In practice, since each device would consume system resources the supported numbers would be much lower.
70. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.
71. Some SR-IOV implementations do not allow local delivery and SR-IOV is not entirely secure without an IOMMU.
72. This can arguably be generalized as a concern for any direct-device-attachment strategy.
73. http://dpdk.org/doc/api/rte__memcpy_8h.html.
74. PMD, ring library, timer library, hash library, memory zone library, mempool library, etc.
75. NTT's LAGOPUS (lagopus.github.io) and Microsoft's PacketDirect.
76. This is a simplification, normally there are two separate unidirectional rings—a TX and RX ring.
77. http://info.iet.unipi.it/∼luigi/netmap/.
78. Solarflare open-on-load.
79. This drawing is also simplified as the rings are normally unidirectional pairs.
80. http://info.iet.unipi.it/∼luigi/vale/.
81. This drawing is also simplified as the rings are normally unidirectional pairs.
82. http://www.virtualopensystems.com/en/solutions/guides/snabbswitch-qemu/.
83. The fd.io forwarder (discussed later in this chapter) is a generic forwarder in user space that goes further and even eliminates this kernel dependency.
84. http://www.rdmaconsortium.org/.
85. Each new service component in the netmap/vhost-user paradigm(s) can be a potential bottleneck, but parallel chains are basically additive for throughput—eg, multiple chains of one component have relatively linear scale.
86. http://networkbuilders.intel.com/ecosystem.
87. http://www.windriver.com/.
88. The term "carrier grade" or "six 9s availability" panders to traditional Telco sensibilities. As covered elsewhere, high availability in the cloud is achieved differently than it was on the traditional Telco appliances.
89. http://www.6wind.com/.
90. 6Wind can support VNFs that do not integrate their shared memory driver (obviously at comparatively reduced performance).

91. http://fd.io/.
92. According to their launch materials, the difference between OVS and VPP can be viewed as VPP being a platform on top of which a set of switching functionality similar to OVS can be easily created.
93. To their credit, Intel has been pushing updates to OpenStack to make NUMA and PCIE locality control available to automation.