Server Virtualization

Dr. Rajiv Misra

Associate Professor

Dept. of Computer Science & Engg.

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss server virtualization and the need of routing and switching for physical and virtual machines.

- Then we will discuss the two methods of virtualization: **(i) Docker based and (ii) Linux container based** and also address the problem of networking VMs and its **hardware based approach: SR-IOV, single-root I/O virtualization and software based approach: Open vSwitch**

Cloud Computing depends on Server Virtualization

- **In the context of server virtualization, how do we network the large number of VMs that might reside on a single physical server?** Cloud computing depends heavily on server virtualization for several reasons:

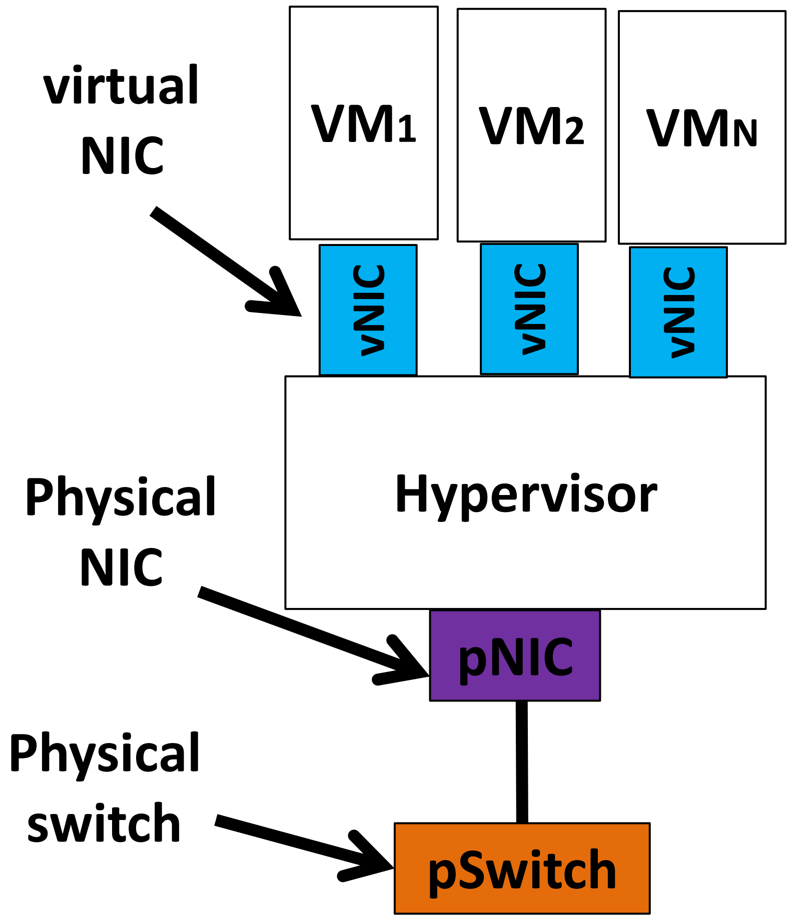
(i) **Sharing of physical infrastructure:** Virtual machines allow multiplexing of hardware with tens to 100s of VMs residing on the same physical server. Also, it allows rapid deployment of new services.

(ii) **Spinning up a virtual machine in seconds:** Spinning up a virtual machine might only need seconds compared to deploying an app on physical hardware, which can take much longer.

(ii) **Live VM migration:** Further, if a workload requires migration. For example, you do physical server requiring maintenance. This can be done quickly with virtual machines, which can be migrated to other servers without requiring interruption of service in many instances. Due to these advantages today, more endpoints on the network are virtual rather than physical.

Server Virtualization

- The physical hardware is managed by a Hypervisor. This could be **Xen, KVM, or VMWare's ESXI,** or multiple such alternatives.
- On top of the Hypervisor runs several VMs in user space.
- The hypervisor provides an emulated view of the hardware to the VMs, which the VMs treat as their substrate to run a guest OS on.
- **Among other hardware resources the network interface card is also virtualized in this manner.** The hypervisor managing the physical NIC, is exposing virtual network interfaces to the VMs. **The physical NIC also connects the server to the rest of the network.**



Networking of VMs inside the Hypervisor

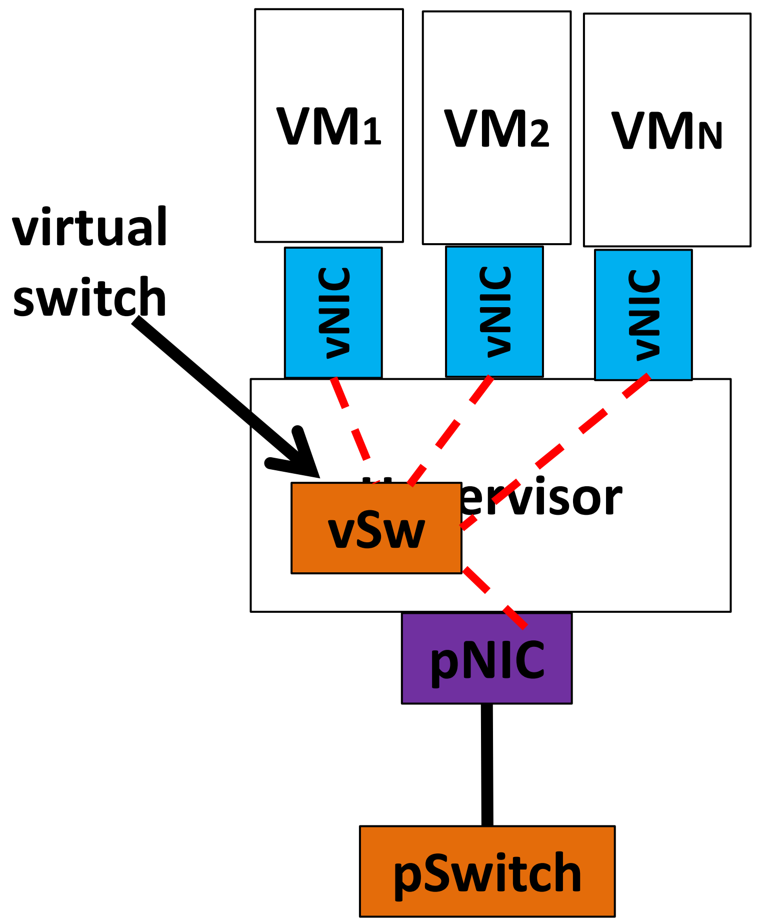
- **The hypervisor runs a virtual switch,** this can be a simple layer tool searching device, Operating in software inside the hypervisor.

- vSw is **connected to all the virtual NICs,** has them as the physical NIC, and moved packets between the VMs and the external network.

Alternate Methods of virtualization:

(i) Using Docker

(ii) Using Linux containers

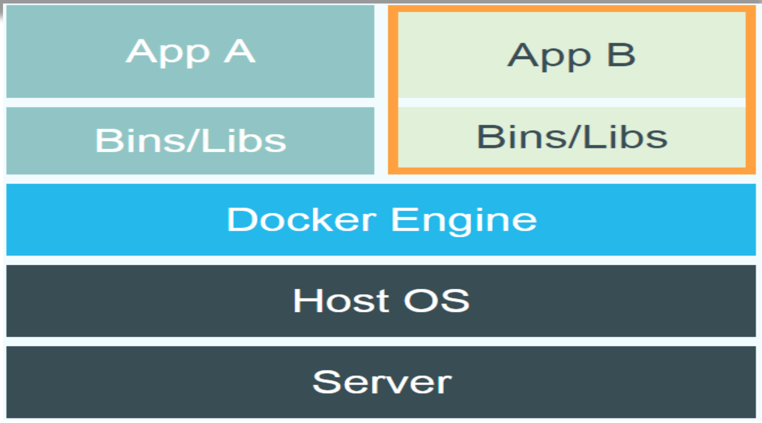


Docker

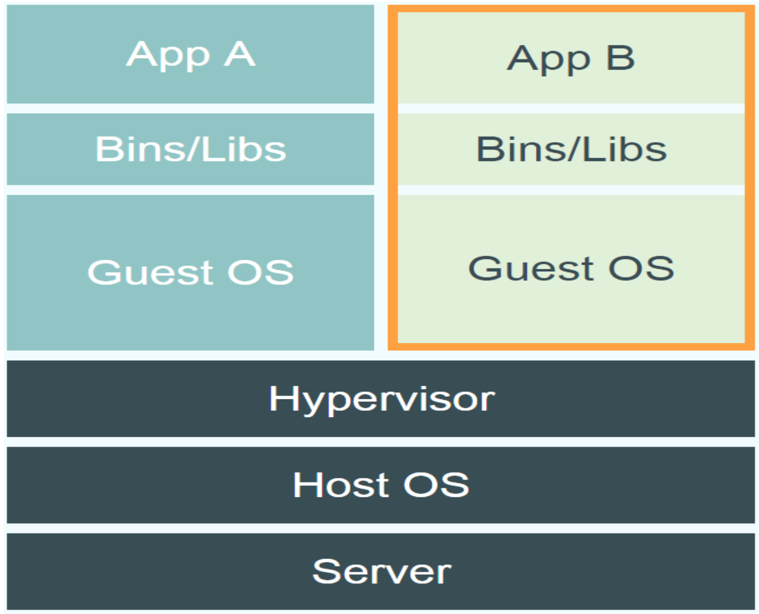
- **Docker is an open-source project that automates the deployment of applications inside software containers,** by providing an additional layer of abstraction and automation of OS–level virtualization on Linux.

- **The Docker Engine container comprises just the application and its dependencies.**

- **It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers.** Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.



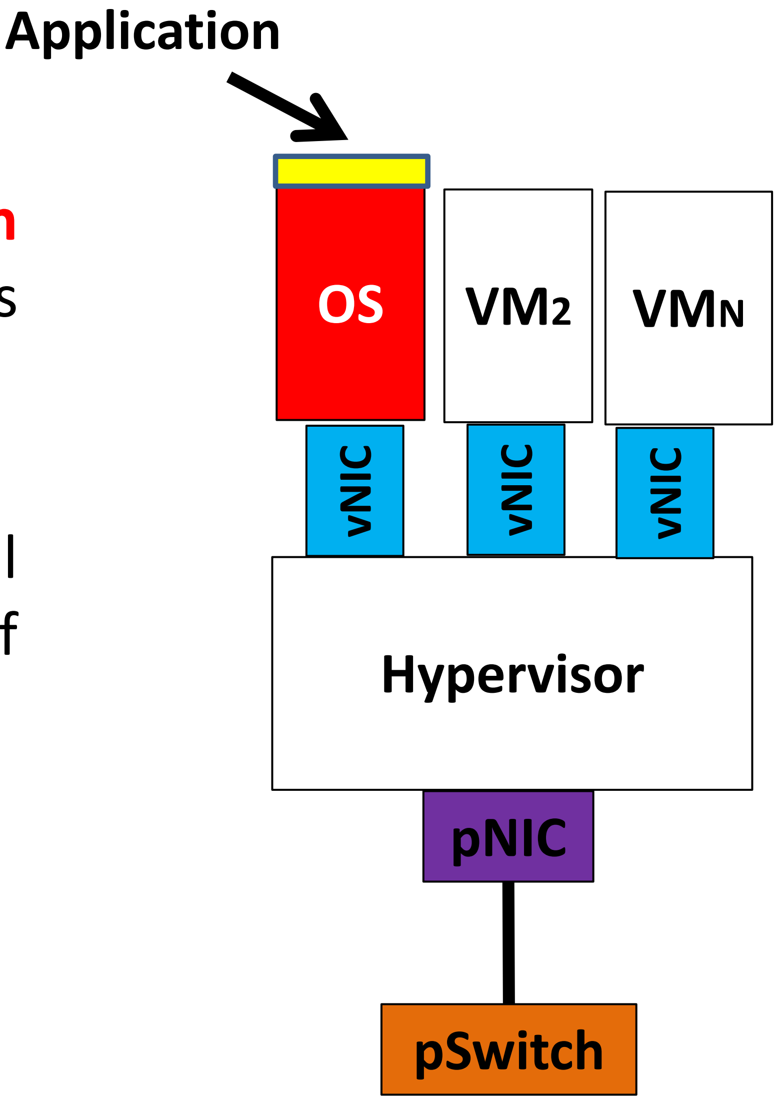
Docker



Virtual Machine

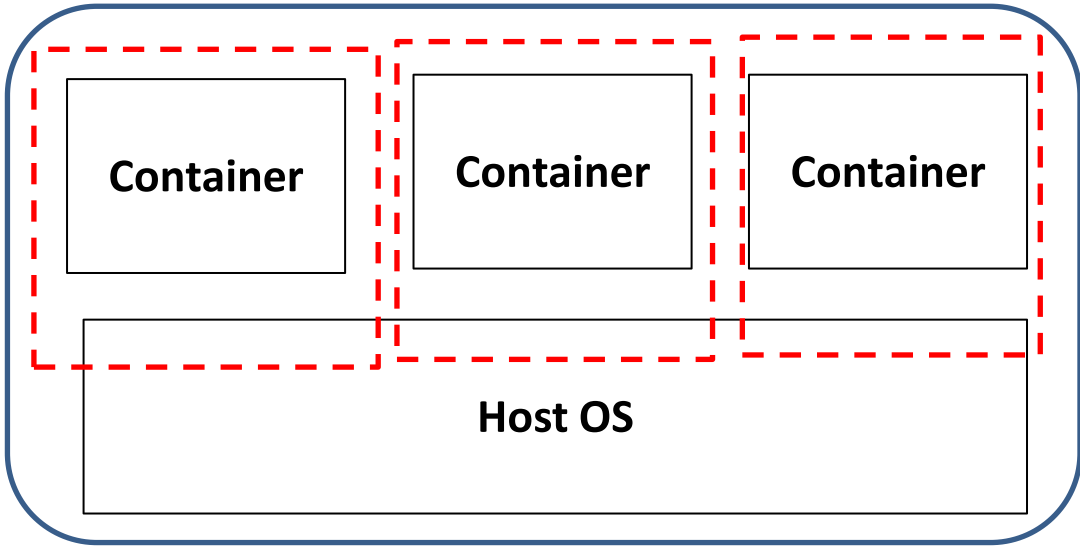
(i) Using VMs as Virtualization

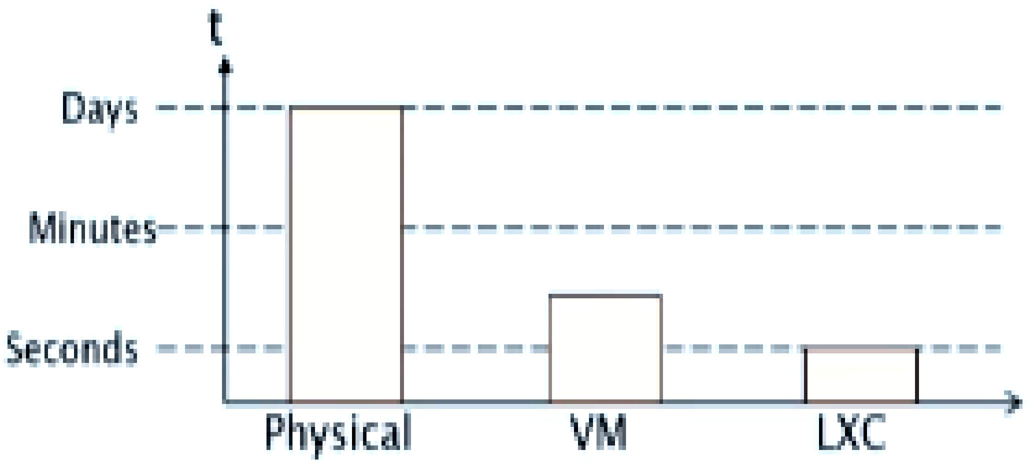
- In this model, **each VM runs its own entire guest OS.** The application runs as a process inside this guest OS.
- This means that even running a small application requires the overhead of running an entire guest OS.



Linux containers as Virtualization

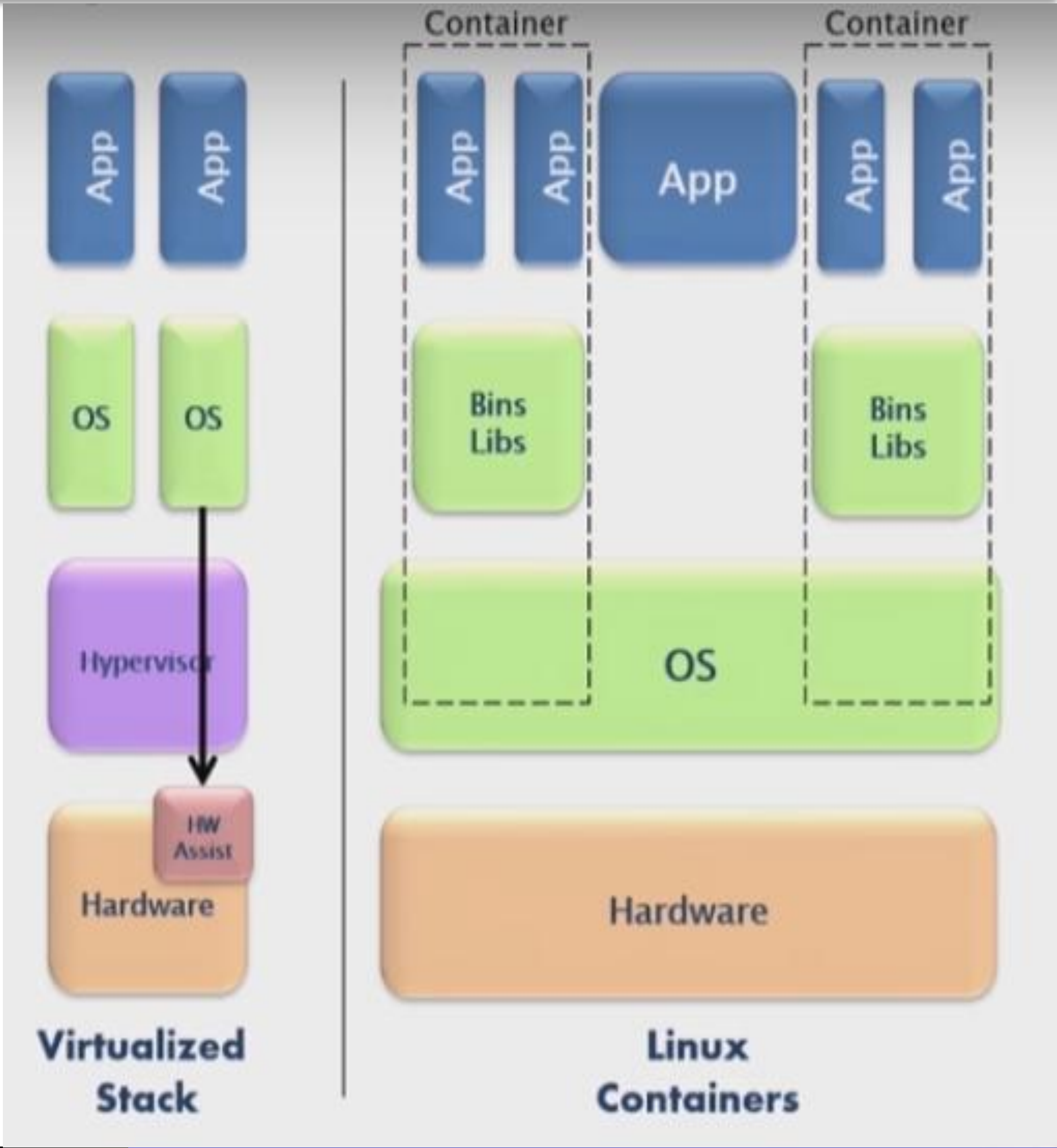
- Create an environment as close as possible to a standard Linux without separate kernel
- **Uses kernel features for separation**
- **Near bare metal (physical hardware) performance**
- **Fast provisioning times (Building up/start-up time)**





Linux containers

- Containers run in host systems kernel
- Separated with policies
- Can use apps in host system
- Can install additional libraries and apps
- Portable between OS variants supporting linux container
- No overhead with hypervisor and guest OS kernel

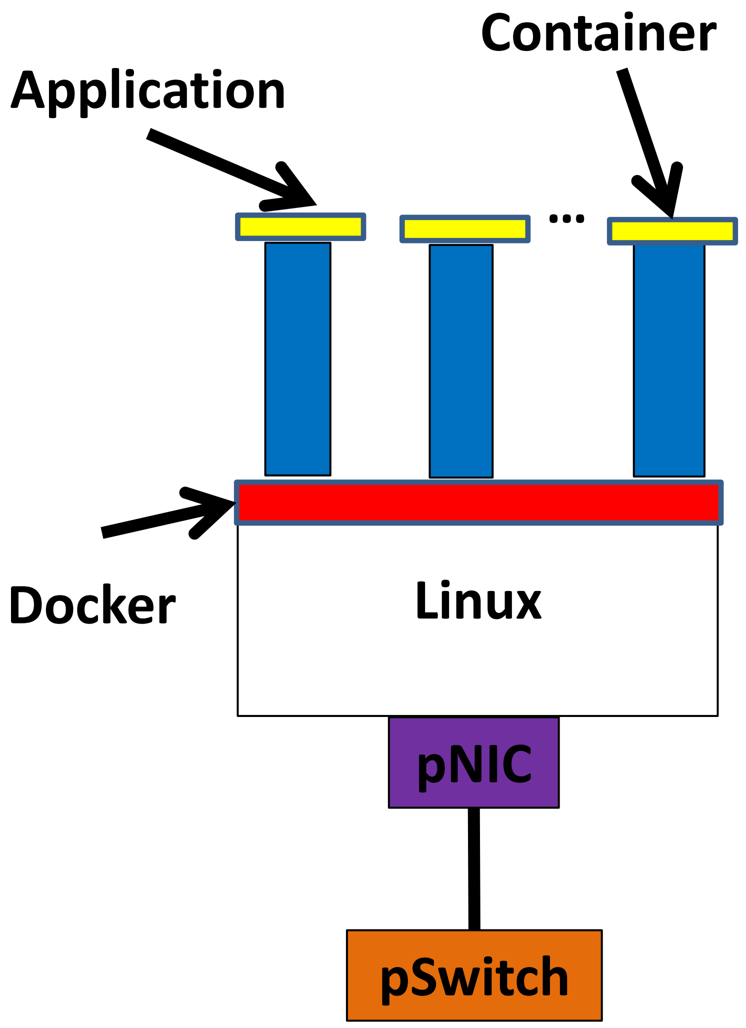


(ii) Using Linux containers

In this approach, an application together with its dependencies uses packages into a Linux container, which runs using the host machine's Linux stack and any shared resources.

Docker is simply a container manager for multiple such containers. Applications are isolated form each other by the use of separate namespaces. Resources in one application cannot be addressed by other applications. This yields isolation quite similar to VMs, but with a smaller footprint.

Further, containers can be brought up much faster than VMs. Hundreds of milliseconds as opposed to seconds or even tens of seconds with VMs.

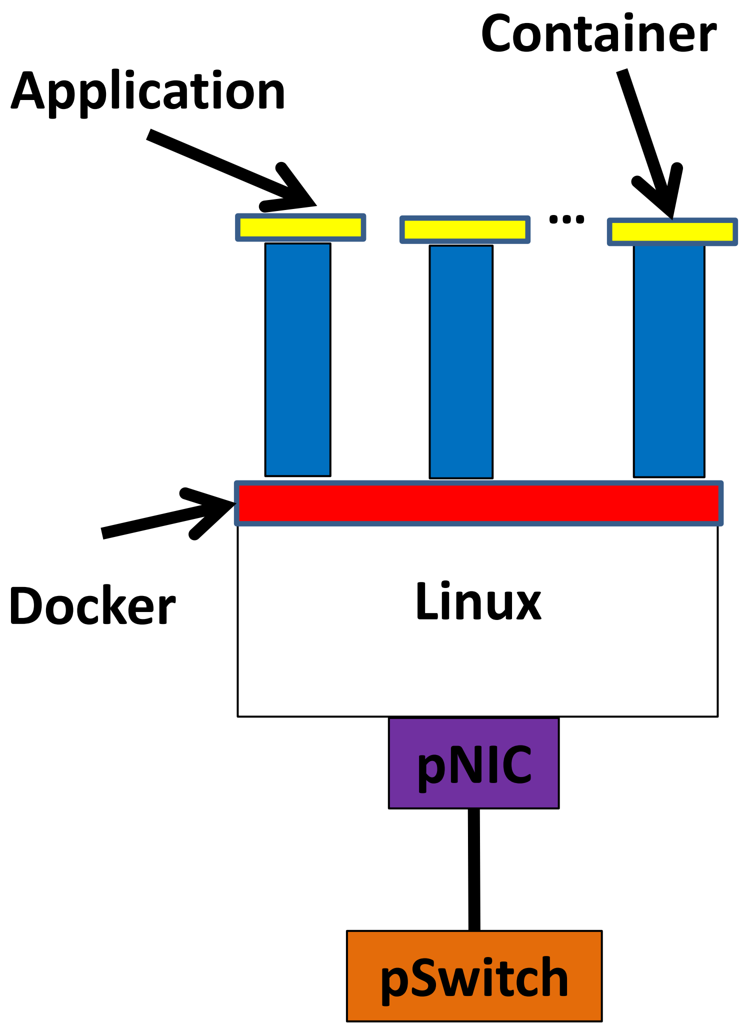


Networking with Docker

- **Each container is assigned a virtual interface.** Docker contains a virtual ethernet bridge connecting these multiple virtual interfaces and the physical NIC.

- **Configuring Docker and the environment variables decide what connectivity is provided.** Which machines can talk to each other, which machines can talk to the external network, and so on.

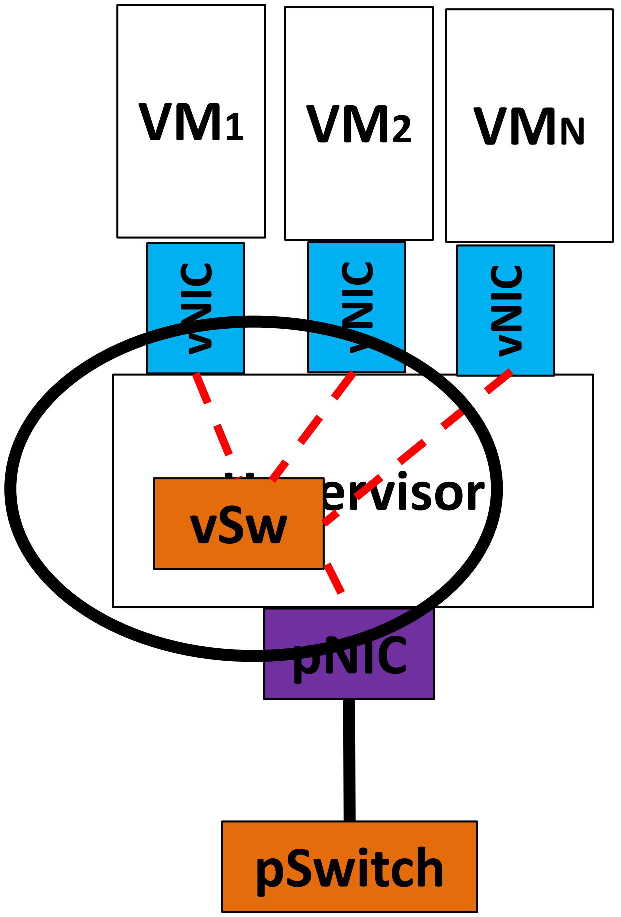
- **External network connectivity is provided through a NAT, that is a network address translator.**



Improving networking performance

- The **hypervisor runs a virtual switch to able to network the VM's** and **CPU is doing the work for moving the packets.**

CPU does the work!



Packet processing on CPUs

Flexible slow, CPU-expensive: Now packet processing on CPUs can be quite flexible because it can have general purpose forwarding logic. You can have packet filters on arbitrary fields run multiple packet filters if necessary, etc. But if done naively, this can also be **very CPU-expensive and slow.**

Packet forwarding: The packet forwarding entails: At 10Gbps line rates with the smallest packets, that's 84 Bytes. We only have an interval of 67ns before the next packet comes in on which we need to make a forwarding decision. Note that ethernet frames are 64 bytes, but together with the preamble which the tells the receiver that a packet is coming and the necessary gap between packets. The envelope becomes 84 bytes. For context a CPU to memory access takes tens of nanoseconds. So, 67 nanoseconds is really quite small.

Packet processing on CPUs

Need time for Packet I/O: Moving packets from the NIC buffers to the OS buffers, which requires CPU interrupts. Until recently a single X86 core couldn't even saturate a ten gigabits per second link. And this is without any switching required. This was just moving packets from the NIC to the OS. After significant engineering effort, packet I/O is now doable at those line rates. However for a software switch we need more.

Userspace overheads: If any of the switching logic is in userspace, one incurs the overhead for switching between userspace and kernel space.

Packet classification: Further, for switching we need to match rules for forwarding packets to a forwarding table. All of this takes CPU time. Also keep in mind that forwarding packets is not the main goal for the CPU. The CPU is there to be doing useful computation.

Approaches for Networking of VMs

There are two different approaches to address the problem of networking VMs:

(i) One, using specialized hardware:

- SR-IOV, single-root I/O virtualization

(ii) Other using an all software approach:

- Open vSwitch

(i) Hardware based approach

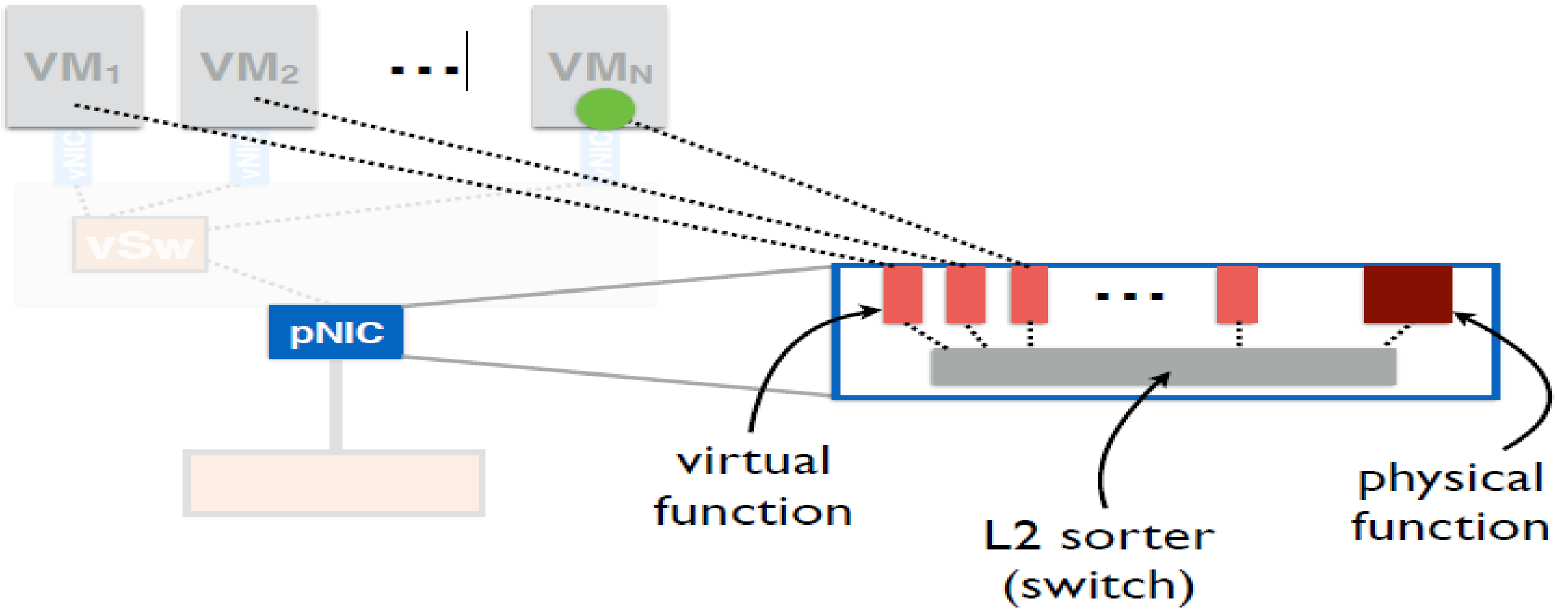
- **The main idea behind the hardware approach is that CPUs are not designed to forward packets, but the NIC is.**

- The naive solution would be to just give access to the VMs, to the NIC. But then problems arise. **How do you share the NIC's resources? How do you isolate various virtual machines?**

- **SR-IOV, single-root I/O virtualization**, based NIC provides one solution to this problem.

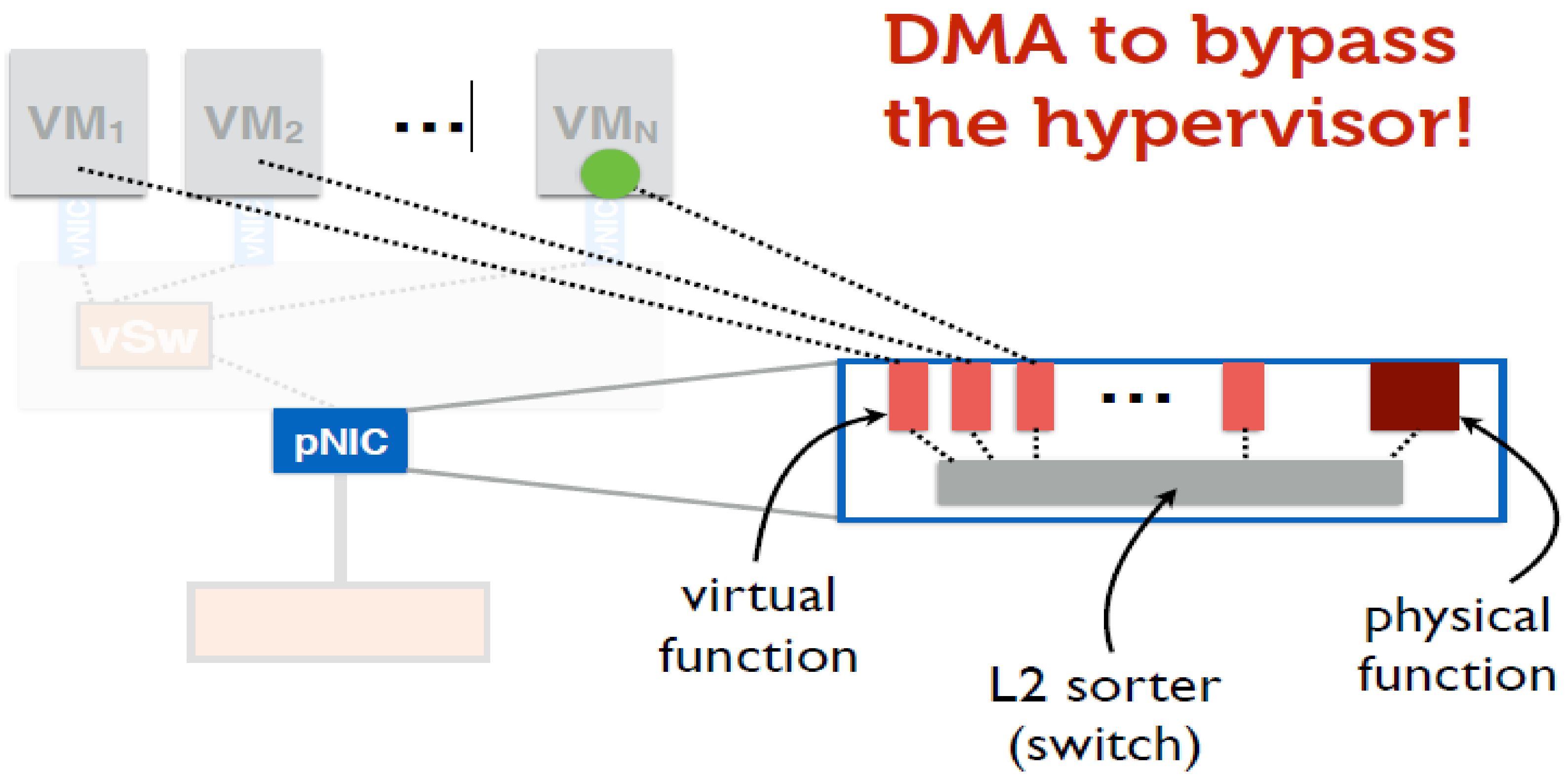
SR-IOV: Single-root I/O Virtualization

- **The SR-IOV, the physical link itself, supports virtualization in hardware.** So let's see inside this SR-IOV enabled network interface card.
- The NIC provides a physical function, which is just a standard ethernet port. In addition, it also provides several virtual functions which are simple queues that transmit and receive functionality.
- **Each VM is mapped to one of these virtual functions. So the VMs themselves get NIC hardware resources.**



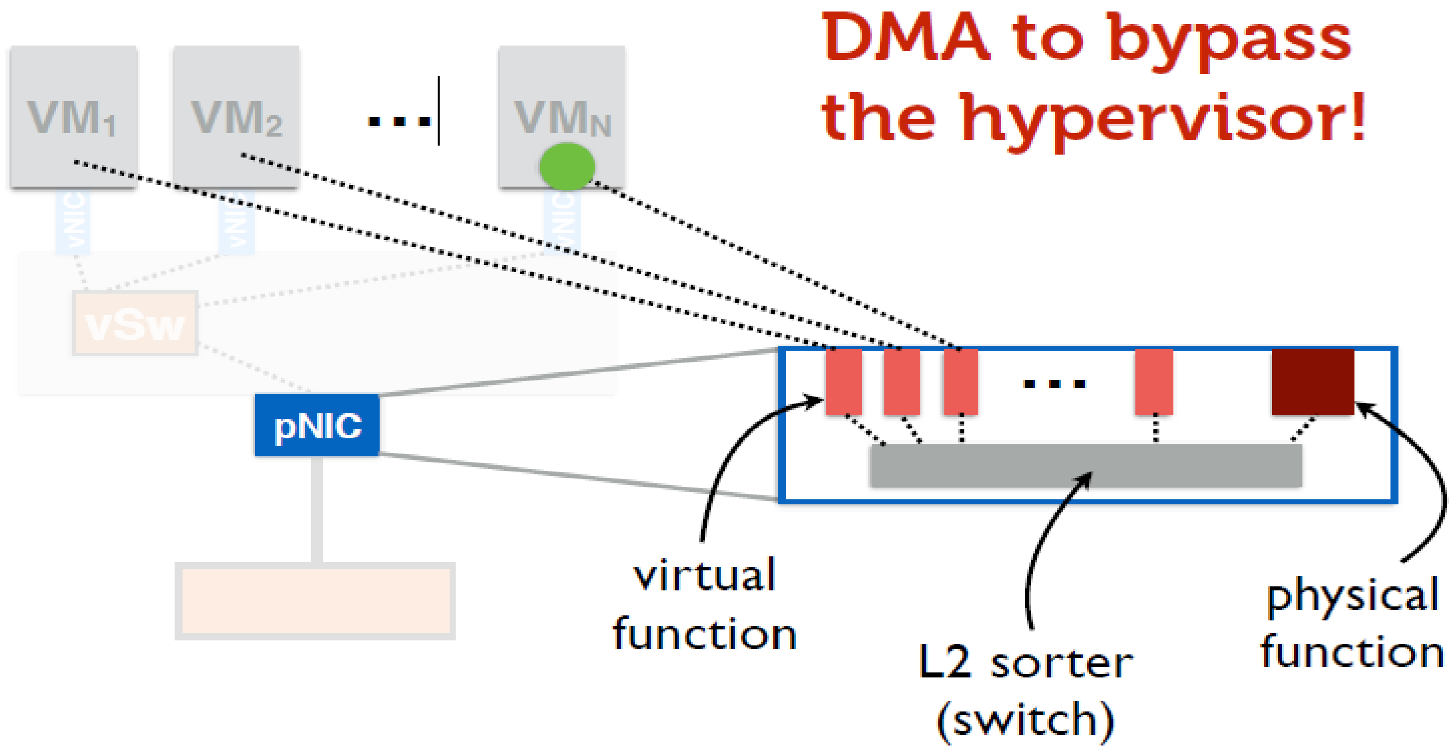
SR-IOV: Single-root I/O Virtualization

- **On the NIC there also resides a simple layer two, which classifies traffic into queues responding to these virtual functions**. Further, packets are moved directly from the net virtual function to the responding VM memory using **DMA (direct memory access).** This allows us to bypass the hypervisor entirely.



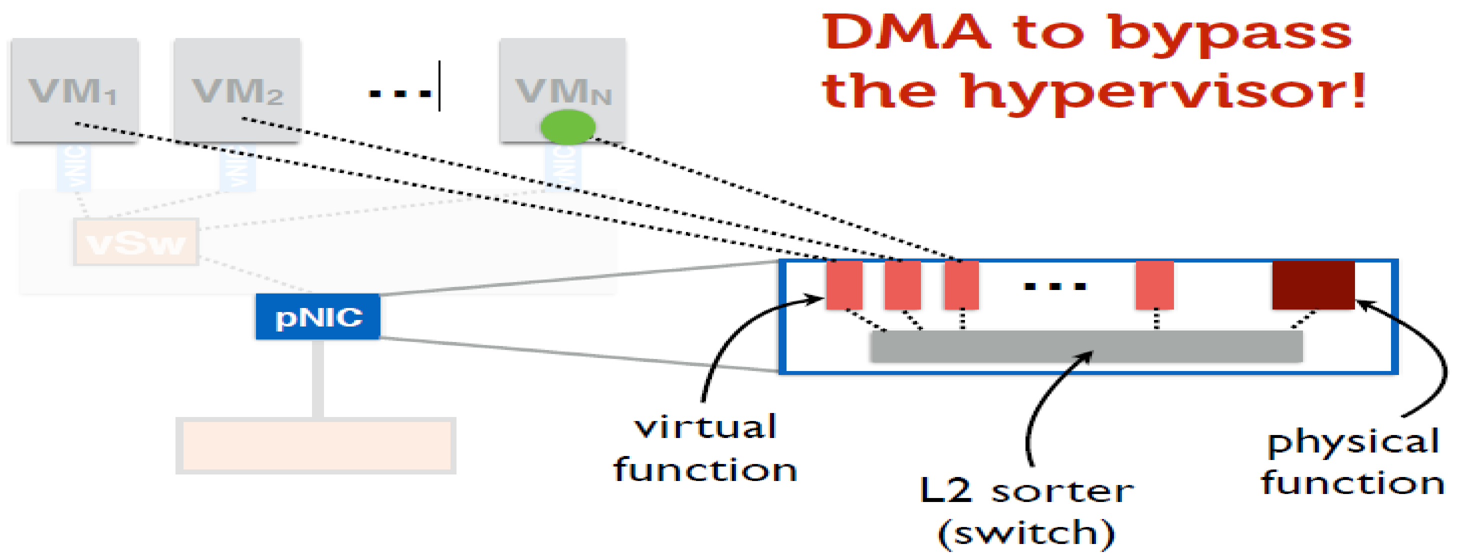
SR-IOV: Single-root I/O Virtualization

- **The hypervisor is only involved in the assignment of virtual functions to virtual machines.** And the management of the physical function, but not the data part for packets. The upshot to this is **higher through-put, lower latency and lower CPU utilization** and this give us close to native performance.



SR-IOV: Single-root I/O Virtualization

- There are downsides to this approach though. For one, **live VM migration becomes trickier**, because now you've tied the virtual machine to physical resources on that machine. The forwarding state for that virtual machine now resides in the layer two switch inside the NIC.
- **Second, forwarding is no longer as flexible.** We're relying on a layer two switch that is built into the hardware of the NIC. So we cannot have general purpose rules and we cannot be changing this logic very often. It's built into the hardware. **In contrast, software-defined networking (SDN) allows a much more flexible forwarding approach.**



(ii) Software based approach

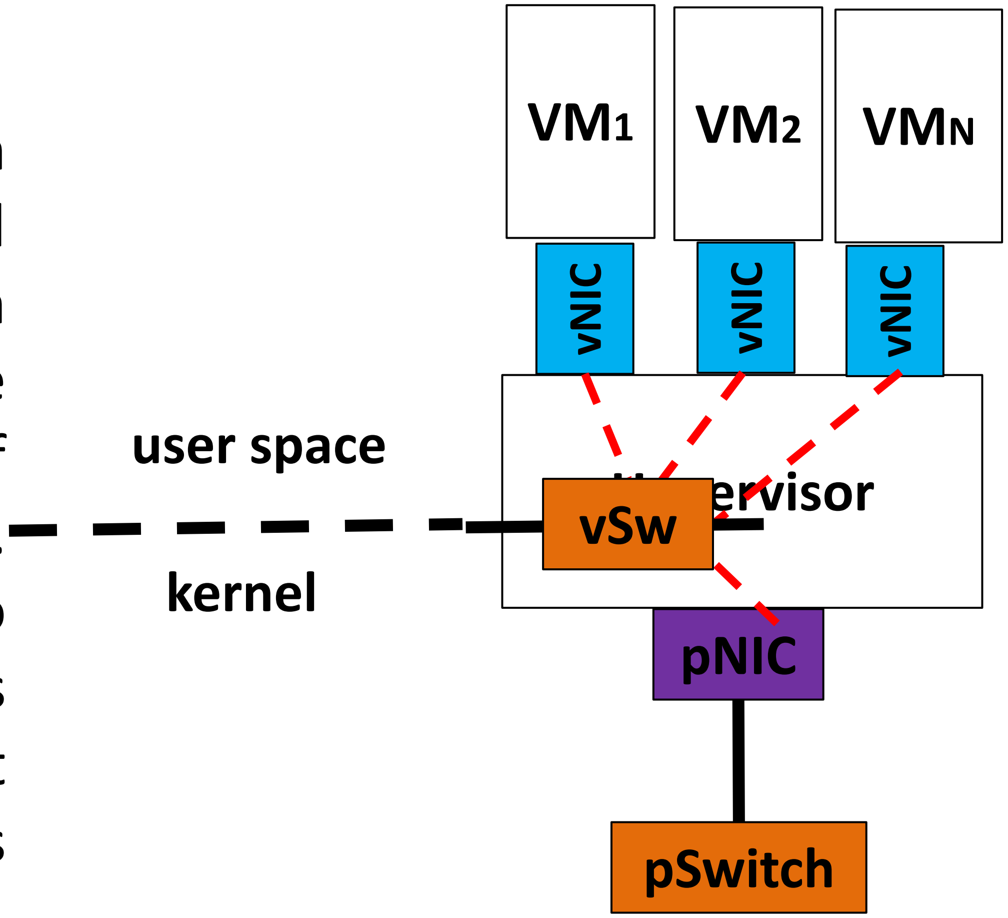
The software based approach that addresses:

(i) **Much more flexible forwarding approach**

(ii) **Live VM migration**

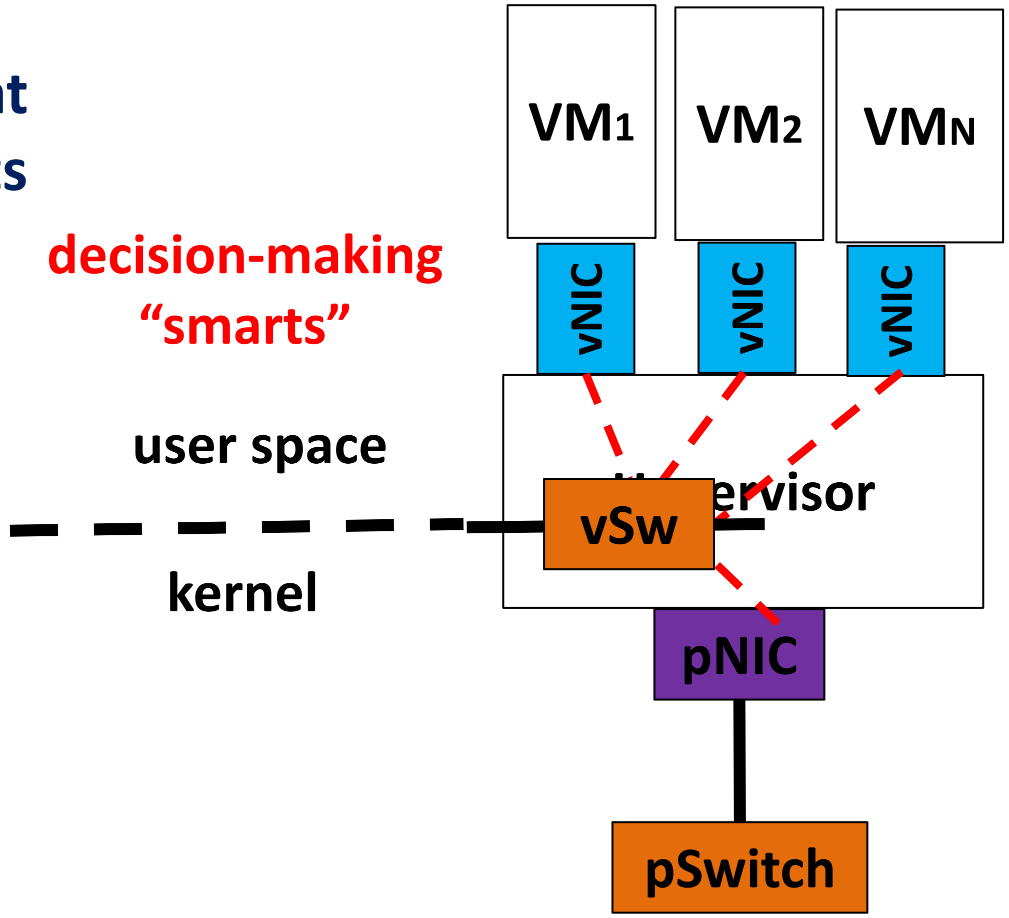
Open vSwitch

- Open vSwitch design goals are **flexible and fast-forwarding.**
- This necessitates a division between **user space** and **kernel space** task. One can not work entirely in the kernel, because of development difficulties. It's hard to push changes to kernel level code, and it's desirable to keep logic that resides in the kernel as simple as possible.



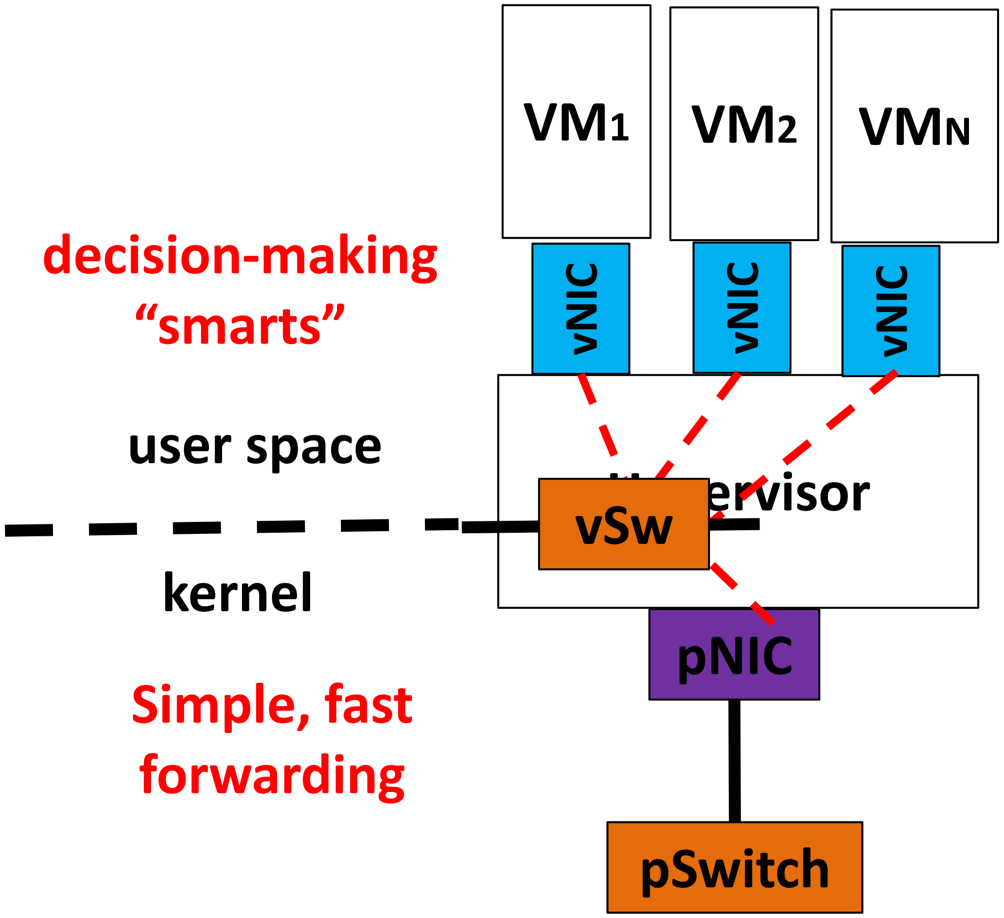
Open vSwitch

- So, the smarts of this approach, is the **switch routing decisions** lie in user space.
- This is where one **decides what rules or filters apply to packets of a certain type.** Perhaps based on network updates from other, possibly virtual, such as in the network. This behavior can also be programmed using **open flow.** **So, this part is optimized for processing network updates, and not necessarily for wire speed packet forwarding.**



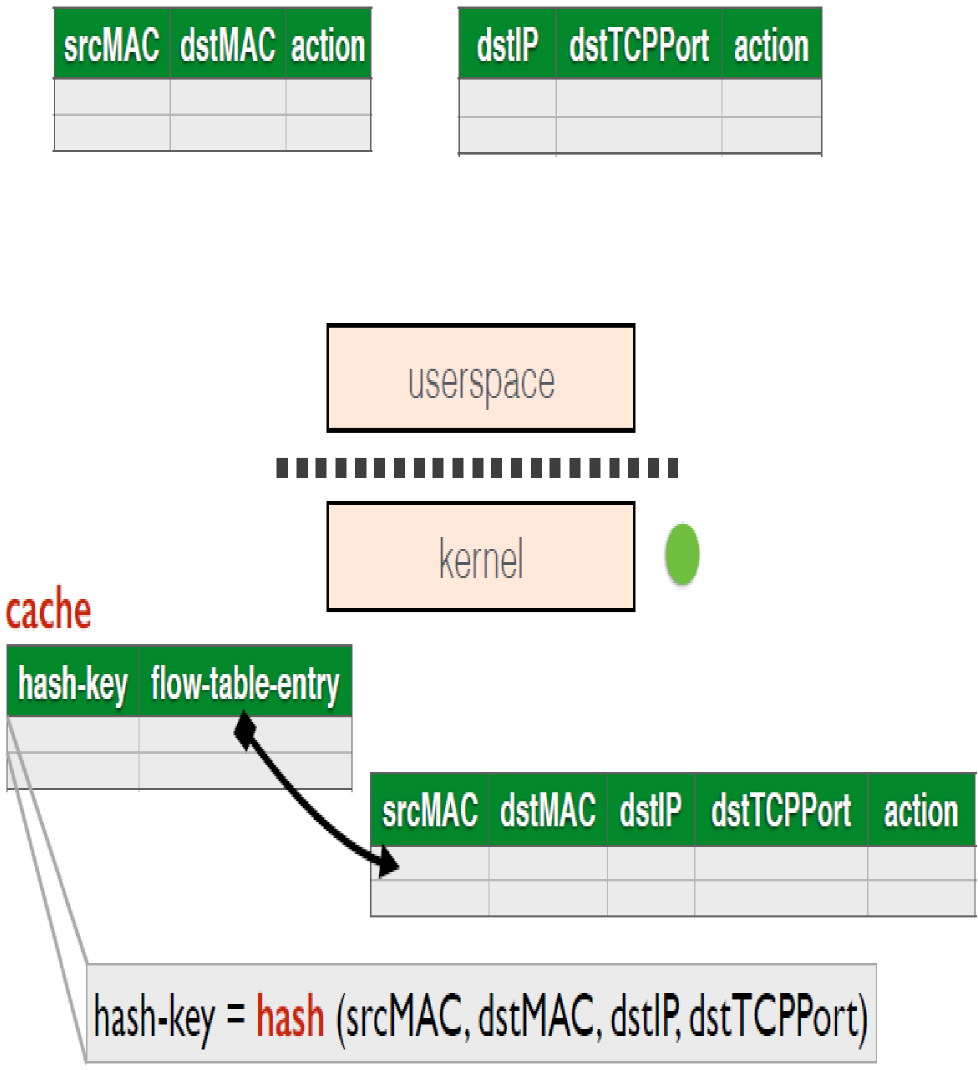
Open vSwitch

- **Packet forwarding**, on the other hand, is handled largely in the kernel, broadly.
- **Open vSwitch approach is to optimize the common case, as opposed to the worst case line rate requirements and caching will be the answer to that need.**



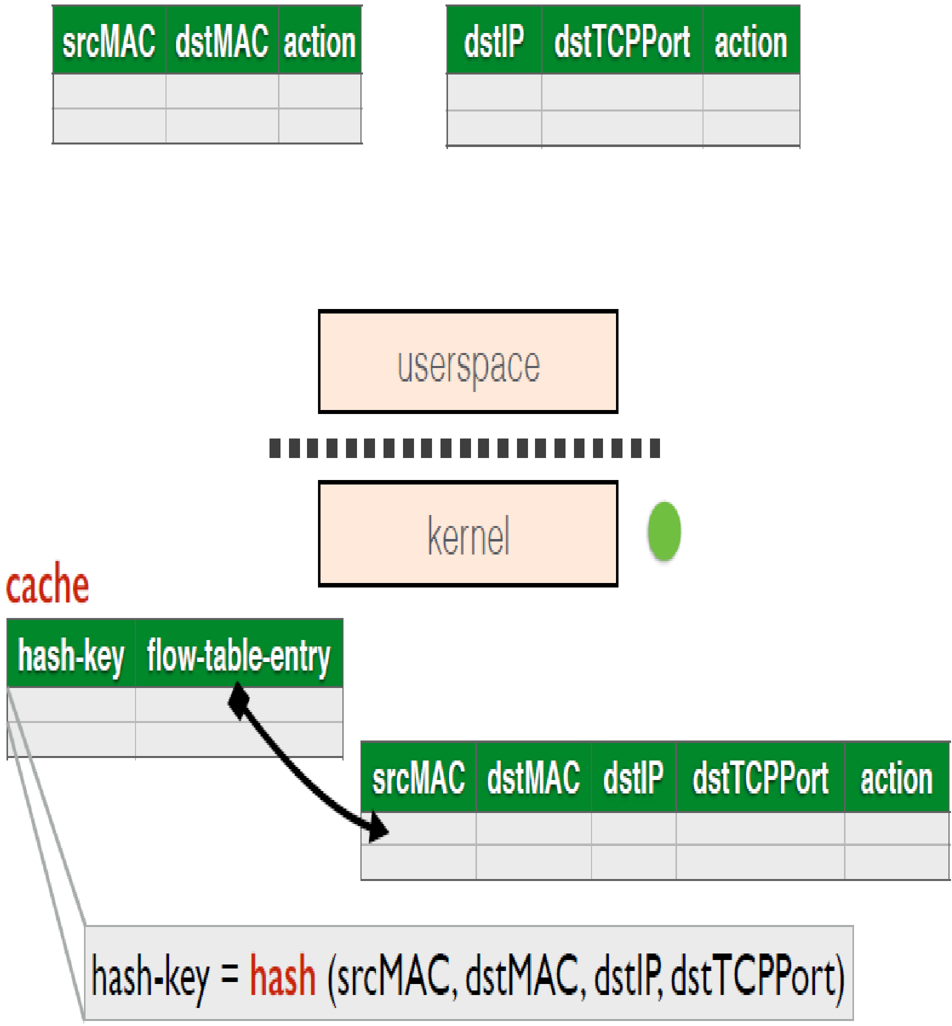
Inside Open vSwitch

- **The first packet of a flow goes to userspace here several different packet classifiers may be consulted.** Some actions may be based on MAC addresses and some others might depend on TCP PORTs, etc.
- The **highest priority matching action** across these different classifiers will be used to forward the packet.
- **Once a packet is forwarded, a collapsed rule used to forward that packet is installed in the kernel. This is a simple classifier with no priorities.** The following packets of this flow will never enter user space, seeing only the kernel level classifier.



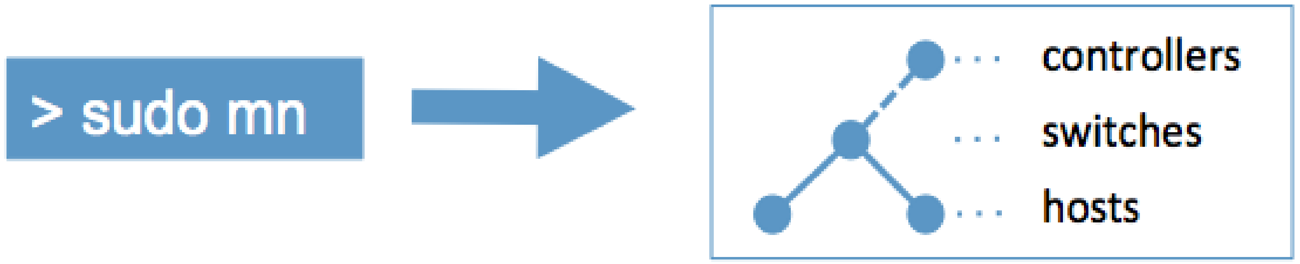
Inside Open vSwitch

- The problem though is when it's still running a packet classifier in the kernel in software. What this means is for **every packet that comes in, you are searching in this table for the right entry that matters and using that entry for forward the packet. This can be quite slow.**
- **Open vSwitch solves this problem is to create a simple hash table based cache into the classifier.** So instead of looking at this entire table and finding the right rule. You'll hash what fields are used to match packet, and the hash key is now your pointer to the action that needs to be taken. And, these hash keys and their actions can be cache.



Introduction to Mininet

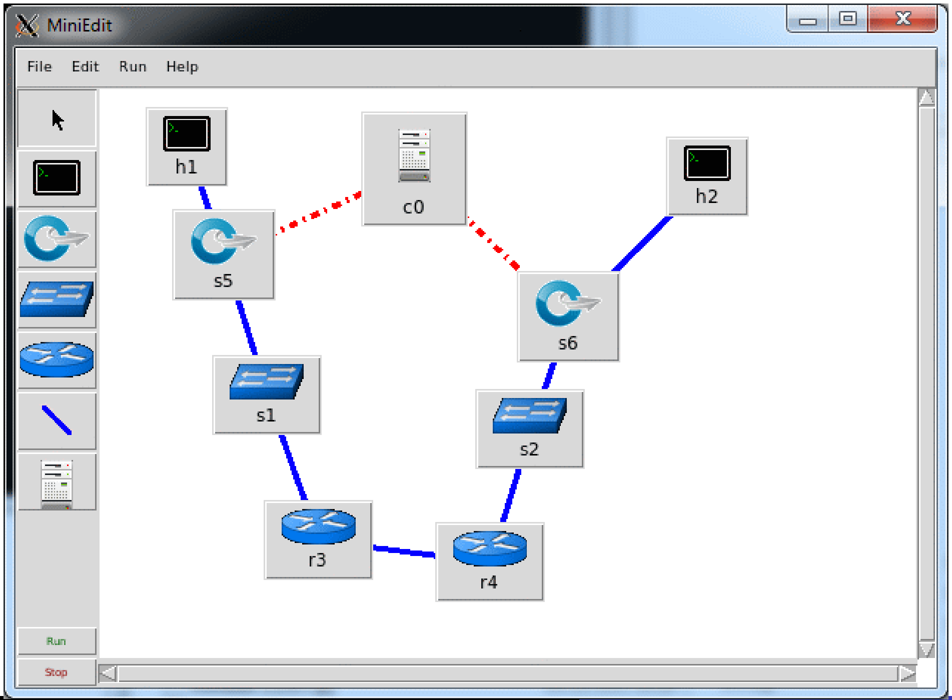
- Mininet creates a **realistic virtual network, running real kernel, switch and application code,** on a single machine (VM, cloud or native), in seconds, with a single command:



- It is a **network emulator** which creates realistic virtual network
  - Runs real kernel, switch and application code on a single machine
  - Provides both Command Line Interface (CLI) and Application Programming Interface (API)
    - **CLI: interactive commanding**
    - **API: automation**
  - **Abstraction**
    - Host: emulated as an OS level process
    - Switch: emulated by using software-based switch
      - **E.g., Open vSwitch, SoftSwitch**

Mininet Applications

- **MiniEdit (Mininet GUI)**
  - A GUI application which eases the Mininet topology generation
  - Either save the topology or export as a Mininet python script
- **Visual Network Description (VND)**
  - A GUI tool which allows automate creation of Mininet and OpenFlow controller scripts



Important Links for Mininet

- mininet.org
- github.com/mininet
- github.com/mininet/mininet/wiki/Documentation
- reproducingnetworkresearch.wordpress.com

Comparison

- The **hardware based approach** the **SR-IOV** takes **sacrifices flexibility and forwarding logic** for line rate performance in all scenarios. Virtually hitting native performance.

- While the **software based approach** **Open vSwitch**, the compromise made is to avoid targeting worst case performance, and **focusing on forwarding flexibility**.

References

- W. Felter, A. Ferreira, R. Rajamony and J. Rubio, "**An updated performance comparison of virtual machines and Linux containers**," *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Philadelphia, PA, 2015, pp. 171-172.

- Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado, "**The design and implementation of open vSwitch**" In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15). USENIX Association, Berkeley, CA, USA, 2015, pp. 117-130.

Conclusion

- In this lecture, we have discussed **server virtualization and also discuss the need of routing and switching for physical and virtual machines.**

- We have discussed two methods of virtualization:
  (i) Docker based and (ii) Linux container based.

- To address the problem of networking VMs, two approaches are discussed: **(i) One, using specialized hardware: SR-IOV, single-root I/O virtualization and (ii) Other using an all software approach: Open vSwitch**