Tachyxx Xen Hypervisor

Internals and CPU support

Igor Lesik

Version v0.1, 2021.05.07

Table of Contents

Copyright

Introduction

Who should read this book

The primary audience of this book is a) system software engineers who are interested in Xen internals or intend to port Xen to a new CPU architecture and b) hardware CPU designers who want to understand the system requirements for HW virtualization support.

1. Zen of Xen

1.1. Power of Name

If you are a fan of fantasy books, you know that a name has power. Let us find out what name Xen means. Xen was originally developed as virtual machine monitor for XenoServers project. Word *xeno* (/'zeno) originates from Greek word xenos, meaning "alien, stranger, foreigner". Therefore the name Xen is hinting at advanced alien technology.

Word Xen is often combined with word "hypervisor", because Xen virtual machine monitor is a hypervisor. Prefix *hyper* here means "above super". OS kernel runs at elevated privilege level compare to applications and is called *supervisor* because it a) manages the applications and b) has more priviledges than application code. When OS kernel runs under control of Xen virtual machine monitor, the kernal is supervised by the Xen, and Xen has higher privilege level. The fact that Xen runs at higher privilege level than "supervisor mode" OS code and supervises it is the origin of the term "hypervisor".

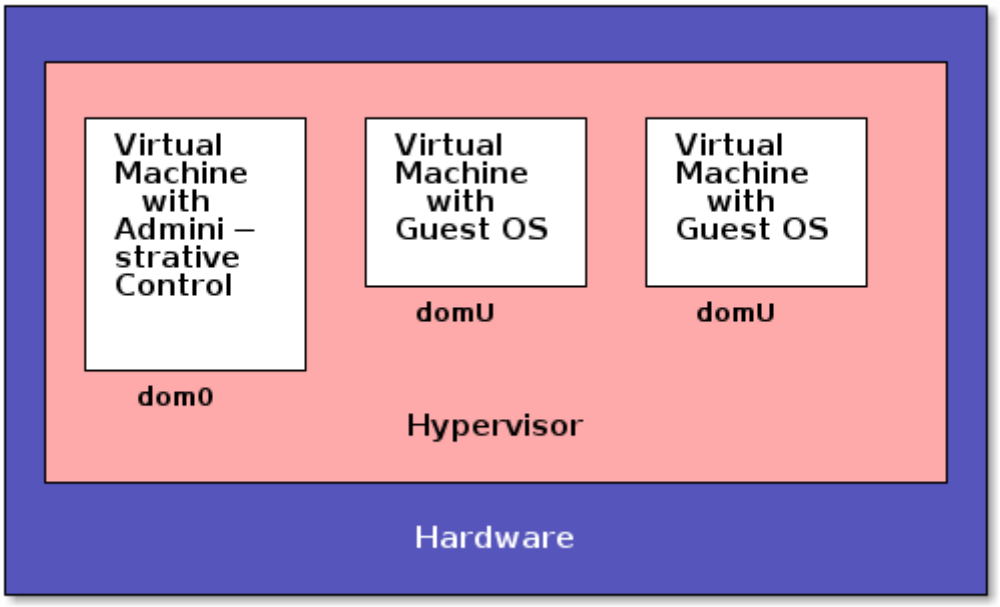
Sometimes word Xen is used just for the hypervisor, other times it is used for all the parts involved in to the virtualization solution: hypervisor, OS kernel virtualization modifications, drivers for paravirtualization, tools and etc.

1.2. What Is Virtualization?

Xen virtualizes the computing system as whole, including CPU, memory, storage devices, network resources, I/O devices and etc. The virtualization in this case is the act of creating a virtual (rather than actual) version of a computer hardware platform. Normally, in order to get most benifits from the virtualization, more than one virtual hardware platform is created. The case when a single virtual platform is created can be considered an emulation (and it is also has its usages).

1.3. Hypervisor-based approach to virtualization

There are many different approaches to virtualization, we are not going to discuss and compare them here; if you are interested in this topic, there are many available books and Internet resources that explain different virtualization techniques. Xen is using hypervisor-based approach, where a hypervisor is a low-level virtual machine monitor that runs directly on the physical hardware and loads first during the machine boot process. The hypervisor runs virtual machines (VM). Virtual machine runs operating system (which runs applications). Normally, an operating system is compiled for the same instruction set as the physical machine on which the virtual systems are running.



Xen terminology defines a concept of "Xen domain", which is a specific instance of a Xen virtual machine. Xen supports two basic types of domains:

- Xen architecture has one domain with a specially privileged Xen-modified kernel that is used to manage, monitor, and administer all other Xen virtual machines. This specially priviliged domain and kernel is known as *domain0* or *dom0*. This kernel communicates with the hypervisor.
- Other domains are known as guest domains, unprivileged domains, *domainU* or *domU*. It is *dom0* that starts any *domU*.

1.4. CPU Virtualization

For a CPU to be completely virtualized, core running in one domain must not affect code running in another domains. Privileged instructions have direct access to CPU resources and can access memory by physical address. These instructions present the biggest problem for virtualization, as they can change CPU state and affect all code running on that CPU. Popek and Goldberg [popek] in their 1974 paper "Formal Requirements for Virtualizable Third Generation Archotectures." devide critical for virtualization instructions into three categories:

- Privileged instructions that execute in a privilege mode. They have to trap if executed outside of their privilege mode. In a virtualized system, the trap handler makes sure that the effect of the instruction is contained inside the domain.
- Control sensitive instructions that change global CPU state by communicating with devices, changing global configuration registers, updating virtual to physical memory mappings.
- Behavior sensitive instructions that behave differently depending on the configuration.

In order for an architecture to be virtualizable, all control sensitive instructions must be trappable privileged instructions.

1.5. Memory Virtualization

Memory virtualization is relatively straightforward. Each physical memory page is assigned to one domain, this way physical memory is spit between domains; the difficulty is due to the fact that OS kernel virtual-to-physical translation operates with interemidiate "psedo" physical address that needs to be translated into "real" physical address, called machine address. Every priviliged instruction that accesses physical memory or changes virtual to physical mapping must be trapped.

Memory virtualization performance critical HW support

- HW support for 2 levels page-tables walk: virtual-to-physical and physical-to-machine.
- Tagged TLB that allows flushing per domain, without flushing all TLB entries.

1.6. I/O Virtualization

With devices, other than CPU and memory, there are two main reasons why virtualization is difficult:

- Most devices are not designed with virtualization in mind.
- For some devices it is not obvious what virtualization means for them or how these devices can support virtualization.

For example, most of the time it is easier to assign keyboard, mouse and display to one domain, instead of sharing it. On other hand, storage devices and network devices most of the time are shared.

Storage devices are usually virtualized similar to memory by partitioning the physical space between domains.

One well known problem is when a device uses DMA without support from *Input/Output Memory Management Unit* (IOMMU). When unmodified OS is unaware it is running not on actual hardware platform but on virtual platform, DMA will use hypervisor-provided physical address instead of machine address.

Here is a list of suggestions for choosing existing devices for your system or when new device is to be designed:

- Make sure the device has *Input/Output Memory Management Unit* (IOMMU).
- Make sure the whole device state can be saved and later restored.
- For security critical devices, make sure there is a way to protect information in case multiple domains share this device.

1.7. Paravirtualization

The paravirtualization approach assumes that a guest system *knows* that it runs on top of a hypervisor. Because of that knoweledge it becomes possible:

1. to compile OS kernel without using problematic instructions that do not trap into the

hypervisor, intead use some direct methods to communuicate with the hypervisor;

2. run inside less priveledged level and do not use any priveledged instructions, instead delegate all work to the hypevisor by means of *hypercalls*;
3. simplify a view at the system, do not implement device drivers for any real hardware, instead delegate administrative domain dom0 to talk to the real hardware and use simple drivers to communicate with dom0 abstract devices.

Note that in the past when x86 did not have enough support for the full hardware-assisted virtualization, the paravirtualization was the only sensible solution providing reasonable performance. However, it does not mean that the paravirtualization is not used now days; due to its simplicity compare to HW/SW solutions and the fact that in many cases paravirtualizartion still provides better performance, the paravirtualization is widely used when customizing OS is possible.

1.8. Hardware-Assisted Virtualization

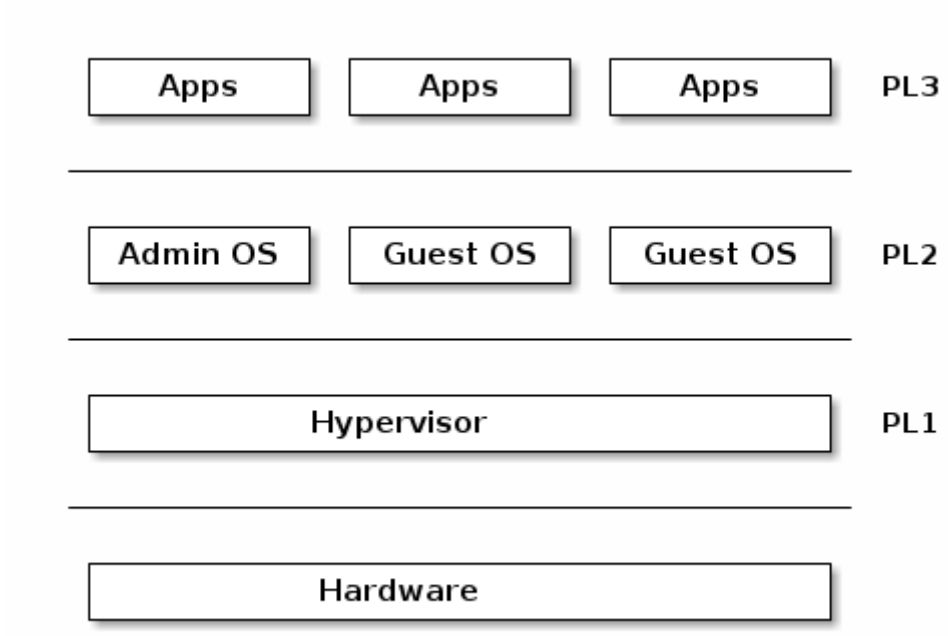
The hardware assisted virtualization, often referred to as HVM (Hardware Virtual Machine), allows running of unmodified operating systems and relies on HW support of virtualization.

1.9. Types of virtualization

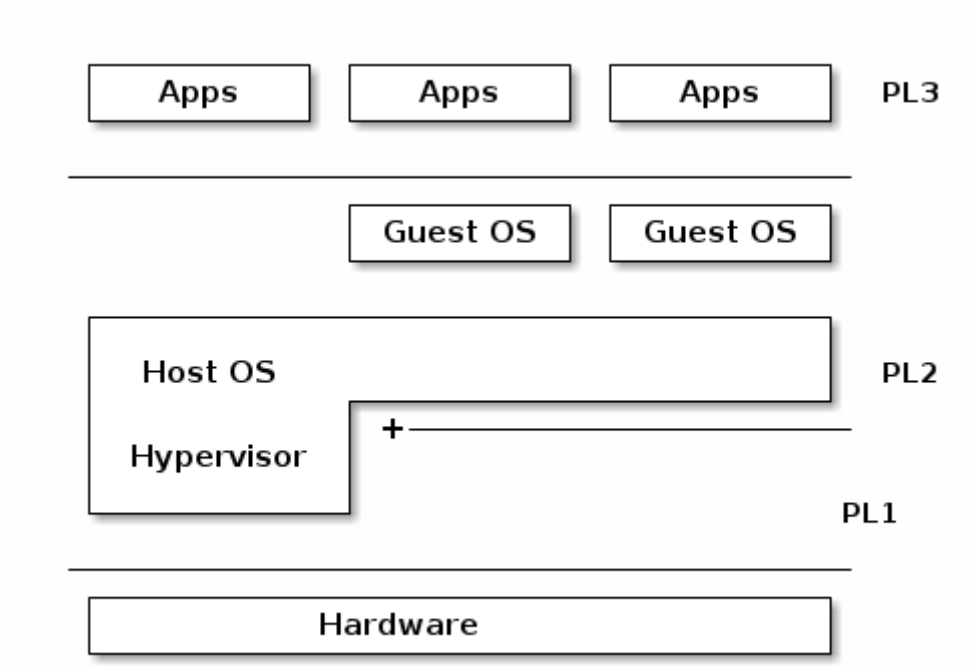
The are many definitions and terminology involved when we look at different types of virtualization, so many that it is confusing sometimes. First of all, there are hypervisor Type-1 and Type-2.

From here on we will use Tachyum notation PLn of the privilege levels, where n=0 is most priveleged level, n=1 is hypervisor level, n=2 is kernel level and n=3 is user level.

Hypervisor Type-1 is native or bare-metal hypervisor. These hypervisors run directly on the host's hardware to control the hardware and to manage guest operating systems.



Hypervisor Type-2 is hosted hypervisor. These hypervisors run on a conventional operating system just as other computer programs do. A guest operating system runs as a process on the host. Type-2 hypervisors abstract guest operating systems from the host operating system.



The line between Type-1 and Type-2 sometime is not clear. For instance, Linux's Kernel-based Virtual Machine (KVM) is a) kernel module, b) uses *lowvisor* that can run at PL1, c) the whole host OS can run in PL1 on ARM systems with Virtualization Host Extension support.

Xen is hypervisor Type-1.

Next distinction is by degree of HW support used by virtualization and how much is paravirtualized. Remember we talked about pure paravirtualization (PV) approach and hardware assisted HVM approach. In Xen there are many hybrid solutions in between these two.

Looking at the history of Xen development can shed some light on the origin of different hybrid solutions.

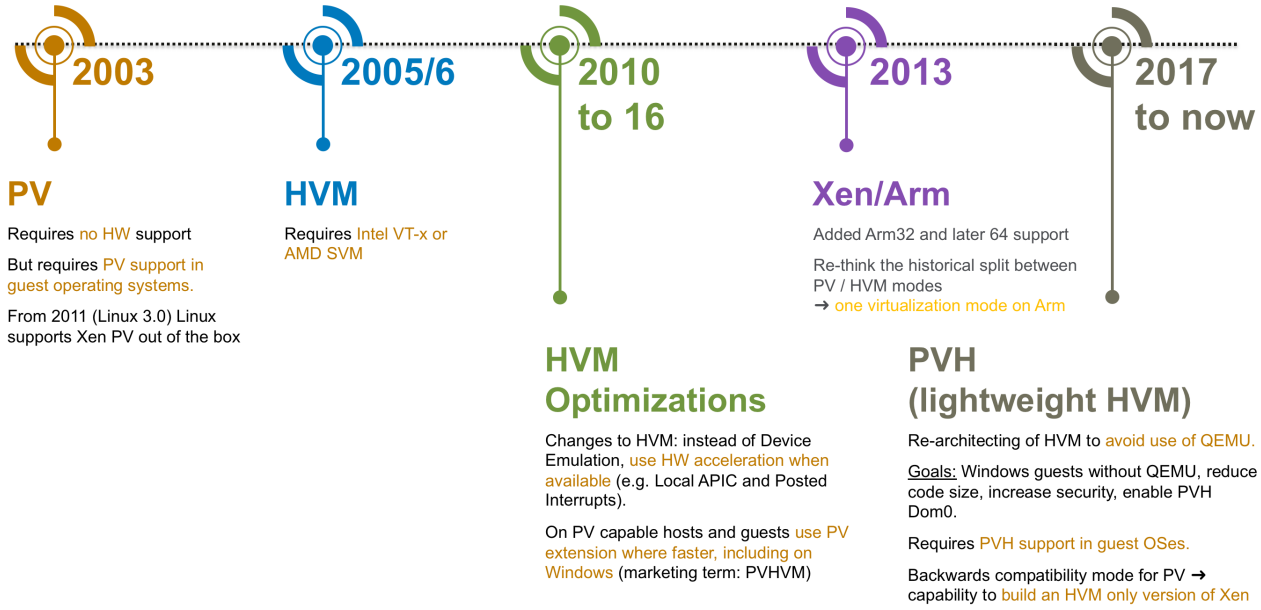


Figure 1. Guest types evolutuion for Xen, wiki.xenproject.org.

Poor Performance
Scope for Improvement
Optimal Performance

PV = Paravirtualized
VS = Software Virtualized (QEMU)
VH = Hardware Virtualized
HA = Hardware Accelerated

x86 Shortcut	Mode	With	Disk and Network	Interrupts & Timers	Boot Path	Privileged Instructions, Page Tables	QEMU Used
HVM / Fully Virtualized	HVM		VS	VS[1]	VS	VH	Yes
HVM + PV drivers	HVM	PV Drivers Installed	PV	VS[1]	VS	VH	Yes
PVHVM	HVM	PVHVM Capable Guest	PV	PV[2]	VS	VH	Yes
PVH	PVH	PVH Capable Guest	PV	HA[3]	PV[4]	VH	No
PV	PV		PV	PV	PV[5]	PV	No
ARM							
N/A	N/A		PV	VH	PV[6]	VH	No

Figure 2. Overview of the various virtualization modes implemented in Xen, wiki.xenproject.org.

1.10. The role of dom0

TODO

1.11. Unprivileged domains

TODO

https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview, I/O Virtualization in Xen. No HVM

without QEMU???

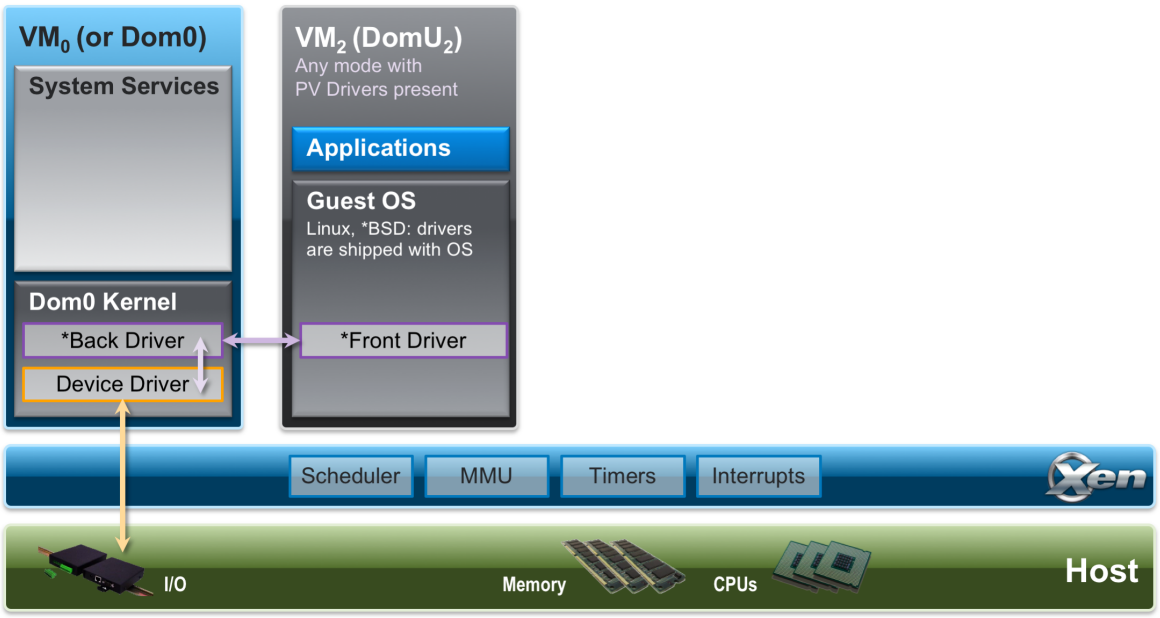


Figure 3. I/O Virtualization using the split driver model .

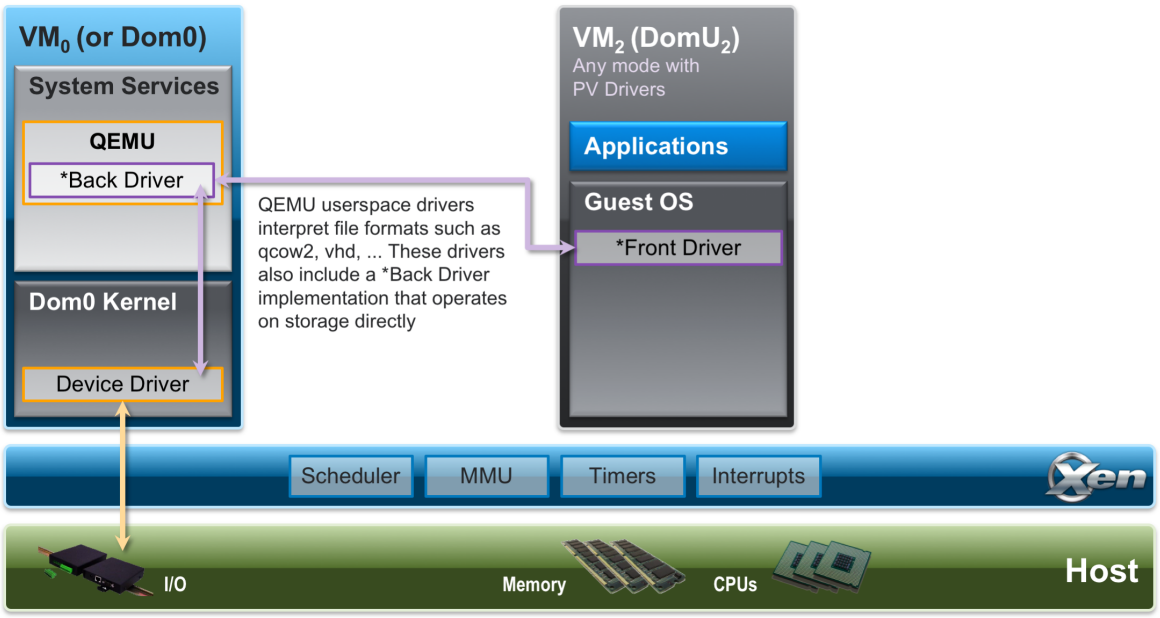


Figure 4. I/O Virtualization using QEMU user space back-end drivers.

1.12. History Of Problems with Virtualization

While designing a new CPU the architects should be aware of the problems other CPU architectures had or still have to support virtualization by Xen. The understading of past and existing problems is crutial in designing a CPU that can support virtualization efficiently, when overhead of running the hypervisor is practically unnoticeable.

- One well known problem with x86 was that some privileged instructions did not trap when they were executed with unsufficient privileges failing silently. Some virtualizers monitored instruction stream and patched those misbehaving instructions, practically performing binary translation, which caused significant degradation of performance.
- In past many architechtures did not have a protection level designed specifically for a hypervisor even when they have several protection levels.
- In case of paravirtualization, an absense of a special instructions in the ISA to be used for fast hypecall to the hypervisor is very critical for the performance.
- It used to be on x86 that booting 32-bit domain0 dictated all other domainU kernels to be 32-bit, similar for 64-bit domain0. New CPU architecture should allow different meaningful combinations of kernels, including bitness and endianess.
- TODO DMA, absence of IOMMU
- TODO Need to use QEMU for full system virtualization FVM

1.13. CPU And System Support For Virtualization

Required functinality

- Ability to bind a virtual machine to a specific CPU on the host system. It helps to solve performance problems of a virtual machine under heavy load.
- No limitation for the size of virtual address space available for OS kernels running inside virtual machine.
- Each guest OS may create high network traffic, multiple guests can easily overload the capabilities of a single network interface. The system must support multiple network interfaces with high bandwidth.
- The system must support high IO traffic as guests may run storage intensive applications, such as database applications.
- The system must satisfy all requirements of Trusted Computing for the virtualized environment when multiple operating systems are simultaneously running on a single platform (with a single TPM device). Including the secure migration of the TPM state from one physical system to another when domainU guests are migrated from one system to another.
- Security concerns due to potential information leaks when instructions executed speculatively.
- Ability to save virtual machine state and migrate it easily to another machine.
- Ability to debug code in any privilige mode (protection ring).
- [dall] Ability to run Host OS at hypervisor privilege level without paying high price for the swithing levels, levels should have its own copy of all special registers to avoid saving and restoring state. OS running at PL1 should be able to communicate with PL3 user space code; requires handling exceptions from PL3 directly to PL1.
- 2 stage memory translation
- [dall] Hypervisor software running in PL1(HYP) should be able to completely disable the stage 2 translations when running the hypervisor OS kernel (KVM) or dom0(Xen), giving it full access to all physical memory on the system, and conversely enable stage 2 translations when running

VM kernels to limit VMs to manage memory allocated to them.

1.14. TODO STUDY

Rapid Virtualization Indexing (RVI)

Helps accelerate the performance of many virtualized applications by enabling hardware-based VM memory management

AMD-V Extended Migration

Helps virtualization software with live migrations of VMs between all available AMD Opteron processor generations

check references: https://en.wikipedia.org/wiki/X86_virtualization#AMD_virtualization_(AMD-V)

https://lwn.net/Articles/182080/ says: There is also a cache called an IOTLB which improves performance. In the AMD IOMMU there is optional support for IOTLBs.

https://www.starlab.io/blog/how-the-xen-hypervisor-supports-cpu-virtualization-on-arm

[dall] What if we run dom0 at PL1?

arm64: Virtualization Host Extension support, https://lwn.net/Articles/674533/

https://www.embedded.com/understanding-virtualization-facilities-in-the-armv8-processor-architecture/, the second translation table for the EL2 level, TTBR1_EL2, was added as a part of VM host extensions so that the hypervisors of Type 2 would have its own translation

check sMMUv3.1, ARM System Memory Management Unit Architecture Specification SMMU architecture version 3.0 and version 3.1

interesting usage of time diagrams: https://genode.org/documentation/articles/arm_virtualization

http://www.cs.columbia.edu/~cdall/pubs/isca2016-dall.pdf

ARM provides interrupt virtualization through a set of virtualization extensions to the ARM Generic Interrupt Controller (GIC) architecture, which allows a hypervisor to program the GIC to inject virtual interrupts to VMs, which VMs can acknowledge and complete without trapping to the hypervisor.

http://www.linux-kvm.org/images/7/79/03x09-Aspen-Andre_Przywara-ARM_Interrupt_Virtualization.pdf

2. Build Xen hypervisor

2.1. Cloning source

The first step of adding a new architecture to Xen is to clone Xen project and setup proper version control to be able to merge quickly evolving main branch into your development branch.

Following examples are obtained with Gitlab, however there is nothing Gitlab specific in the used commands and they should work on any Git server.

First step is to clone Xen sources from the official repository and to make Git origin point to the new Gitlab repository instead of the origin of the main Xen repository.

```
mkdir xen
cd xen/
git clone https://github.com/xen-project/xen.git
cd xen
git remote rename origin old-origin  ①
git remote add origin git@gitlab.tachyum.com:sw-dev/hypervisor/xen-project/xen.git
git push -u origin --all
git push -u origin --tags
```

① Save and preserve Github origin.

Next step is to create a development branch based on some stable snapshot.

```
$ git status
On branch master
$ git checkout RELEASE-4.14.1
Note: switching to 'RELEASE-4.14.1'.
HEAD is now at ad844aa352 update Xen version to 4.14.1
$ git switch -c tachyum-develop
Switched to a new branch 'tachyum-develop'
$ git push --set-upstream origin tachyum-develop
```

2.2. Update to newer Xen version

The way Xen repositories are managed, tags like RELEASE-4.14.1 are not on the master branch, instead they are on a release branch. It complicates merging of our changes with updates of another release.

```
git checkout master
git fetch old-origin    # fetch latest code from github
merge old-origin/master # merge changes from old-origin (github) to origin (Gitlab)
git push origin         # push local changes upstream to origin
git push --tags
git checkout RELEASE-4.15.0
git checkout -b tachyum-develop-new
# cherry pick
git checkout tachyum-develop
git difftool -d RELEASE-4.14.1
git checkout tachyum-develop-new
git difftool -d RELEASE-4.15.0
git checkout tachyum-develop
```

2.3. Building Xen

Xen project has many pieces: the hypervisor, dom0 Linux OS, tools and so on. Right now we are going to focus on building the hypervisor only.

The first phase of building is configuring of the build. We are going to configure the build to use Tachyum toolchain for the compilation.

```
$ ./configure --prefix=`pwd`/../build --exec_prefix=`pwd`/../tbuild --target=tachy
--host=tachy --disable-tools --disable-docs --disable-stubdom
```

```
checking build system type... x86_64-pc-linux-gnu
checking host system type... Invalid configuration `tachy': machine `tachy-unknown'
not recognized
configure: error: /bin/bash ./config.sub tachy failed
```

```
$ git diff config.sub
diff --git a/config.sub b/config.sub
index f53af5a2da..1e728546cf 100644
--- a/config.sub
+++ b/config.sub
@@ -1241,6 +1241,7 @@ case $cpu-$vendor in
                      | sparcv8 | sparcv9 | sparcv9b | sparcv9v | sv1 | sx* \
                      | spu \
                      | tahoe \
+                     | tachy \
                      | tic30 | tic4x | tic54x | tic55x | tic6x | tic80 \
                      | tron \
                      | ubicom32 \
```

make build-xen XEN_TARGET_ARCH=tachy CROSS_COMPILE=tachy-linux-gnu-

```
$ export PATH=$PATH:/project/prodigy-sw/ilesik/install/set0/bin
$ make build-xen XEN_TARGET_ARCH=tachy CROSS_COMPILE=tachy-linux-gnu-
Config.mk:69: /project/prodigy-sw/ilesik/xen/xen/config/tachy.mk: No such file or
directory
make: *** No rule to make target '/project/prodigy-sw/ilesik/xen/xen/config/tachy.mk'.
Stop.
```

Changes needed for `make build-xen` *to finish without an error.*

```
$ git status
On branch tachyum-develop
Your branch is up to date with 'origin/tachyum-develop'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   config/tachy.mk
    new file:   xen/arch/tachy/Kconfig
    new file:   xen/arch/tachy/Kconfig.debug
    new file:   xen/arch/tachy/Makefile
    new file:   xen/arch/tachy/Rules.mk
    new file:   xen/arch/tachy/arch.mk
    new file:   xen/arch/tachy/configs/tachy_defconfig
    new file:   xen/include/asm-tachy/byteorder.h

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   config.sub
```

xen/arch/tachy - head.S, Makefife, arch.mk xen/include/asm-tachy

xen/common/ C files

```
ls ./xen/include/asm-tachy/
altp2m.h       device.h         irq.h         setup.h
asm-offsets.h  div64.h          mem_access.h  smp.h
asm_defns.h    domain.h         mm.h          softirq.h
atomic.h       event.h          monitor.h     spinlock.h
bitops.h       flushtlb.h       nospec.h      string.h
bug.h          grant_table.h    numa.h        system.h
byteorder.h    guest_access.h   p2m.h         time.h
cache.h        guest_atomics.h  page.h        trace.h
config.h       hardirq.h        paging.h      types.h
cpufeature.h   hypercall.h      pci.h         vm_event.h
current.h      init.h           percpu.h      xenoprof.h
debugger.h     io.h             processor.h
delay.h        iocap.h          random.h
desc.h         iommu.h          regs.h
```

3. Virtual CPU

3.1. Definition Of Virtual CPU

An VM's OS naturally expects HW to have CPU(s). We do not want to (generally) allocate one physical CPU (pCPU) to VM because we may have less CPUs than there are VMs and because such allocation does not allow for flexible allocation of HW resources. For that there is such as abstraction as Virtual CPU (vCPU); we can create as many vCPUs as we wish (considering practical limitations). Each vCPU is seen as a single physical CPU core by the VM's operating system. One or more vCPUs are assigned to every Virtual Machine (VM).

To create an abstraction of vCPU all virtualization solutions use the idea of time multiplexing when each vCPU gets its own time slot on some pCPU. Similar to OS processes that can run sequentially on a single physical CPU to create an illusion of simultaniously running tasks, vCPUs sequentially run on a physical CPU to create an illusion of existence of many CPUs. An OS running inside virtualization domain believes that it runs on designated only to it CPU(s). In addition to saving architectural state (registers and etc.) on the switch between vCPUs, hypervisor and virtualizing HW should properly queue and route virtual interrupts to vCPUs.

3.2. vCPU Pinning for guests

You can dedicate a physical CPU to a particular virtual CPU or a set of virtual cpus.

3.3. NUMA

https://wiki.xenproject.org/wiki/Tuning_Xen_for_Performance

https://wiki.xenproject.org/wiki/Xen_on_NUMA_Machines

A NUMA machine is typically a multi-sockets machine built in such a way that processors have their own local memory. A group of processors connected to the same memory controller is usually called a node. Accessing memory from remote nodes is always possible, but it is usually very slow. Since VMs are usually small (both in number of vcpus and amount of memory) it should be possible to avoid remote memory access altogether. Both XenD and xl (starting from Xen 4.2) try to automatically make that happen by default. This means they will allocate the vcpus and memory of your VMs trying to take the NUMA topology of the underlying host into account, if no vcpu pinning or cpupools are specified.

4. Memory

Xen hypervisor is responsible for managing the allocation of physical memory to domains, and for ensuring safe use of the paging hardware.

4.1. Memory allocation

Xen hypervisor resides within a small fixed portion of physical memory; it also reserves the top 64MB of every virtual address space. (TODO check if it is still true; and explain purpose) The remaining physical memory is available for allocation to domains at a page granularity. Xen tracks the ownership and use of each page, which allows it to enforce secure partitioning between domains.

Each domain has a maximum and current physical memory allocation. A guest OS may run a "balloon driver" to dynamically adjust its current memory allocation up to its limit.

4.2. Pseudo-physical and machine memory

Machine memory refers to the entire amount of memory installed in the machine, including that reserved by Xen, in use by various domains, or currently unallocated. We consider machine memory to comprise a set of 4K *machine page frames* numbered consecutively starting from 0. Machine frame numbers (MFN) mean the same within Xen or any domain.

Pseudo-physical memory, on the other hand, is a per-domain abstraction. It allows a guest operating system to consider its memory allocation to consist of a contiguous range of physical page frames starting at physical frame 0, despite the fact that the underlying machine page frames may be sparsely allocated and in any order.

To achieve this, Xen maintains a globally readable *machine-to-physical* table which records the mapping from machine page frames to pseudo-physical ones. In addition, each domain is supplied with a physical-to-machine table which performs the inverse mapping. Clearly the machine-to-physical table has size proportional to the amount of RAM installed in the machine, while each physical-to-machine table has size proportional to the memory allocation of the given domain.

5. Interrupts

http://www-archive.xenproject.org/files/xen_interface.pdf says: A virtual IDT is provided - a domain can submit a table of trap handlers to Xen via the `set_trap_table()` hypercall.

5.1. PV Interrupts and events

The hypervisor handles the interrupts, turns them into *events* that get delivered asynchronously to the target domain using a callback supplied via the `set_callbacks()` hypercall. A guest OS can map these events onto its standard interrupt dispatch mechanisms.

6. Time

Guest operating systems need to be aware of the passage of both real (or wallclock) time and their own "virtual time", which is the time for which they have been executing. Furthermore, Xen has a notion of time which is used for scheduling.

6.1. Notions of time

The following notions of time are provided:

Cycle counter time

This provides a fine-grained time reference. The cycle counter time is used to accurately extrapolate the other time references. On SMP machines it is currently assumed that the cycle counter time is synchronized between CPUs.

System time

This is a 64-bit counter which holds the number of nanoseconds that have elapsed since system boot.

Wall clock time

This is the time of day in a Unix-style `struct timeval`.

Domain virtual time

This progresses at the same pace as system time, but only while a domain is executing, it stops while a domain is de-scheduled.

6.2. PV Timestamps

Xen exports timestamps for system time and wall-clock time to guest operating systems through a shared page of memory. Xen also provides the cycle counter time at the instant the timestamps were calculated, and the CPU frequency in Hertz. This allows the guest to extrapolate system and wall-clock times accurately based on the current cycle counter time.

Since all timestamps need to be updated and read *atomically* two version numbers are also stored in the shared info page. The first is incremented prior to an update, while the second is only incremented afterwards. Thus a guest can be sure that it read a consistent state by checking the two version numbers are equal.

6.3. PV periodic ticker

Xen includes a periodic ticker which sends a timer event to the currently executing domain every 10ms. The Xen scheduler also sends a timer event whenever a domain is scheduled; this allows the guest OS to adjust for the time that has passed while it has been inactive. In addition, Xen allows each domain to request that they receive a timer event sent at a specified system time by using the `set_timer_op()` hypercall. Guest OSes may use this timer to implement timeout values when they block.

7. SMP support

how domains are assigned to CPUs?????

8. Devices

8.1. PV devices

Devices such as network and disk are exported to guests using a splitd evice driver. The device driver domain (usually dom0), which accesses the physical device directly also runs a backend driver, serving requests to that device from guests. Each guest will use a simple frontend driver, to access the backend. Communication between these domains is composed of two parts: First, data is placed on to a shared memory page between the domains. Second, an event channel between the two domains is used to pass notification that data is outstanding. This separation of notification from data transfer allows message batching, and results in very efficient device access.

Event channels are used extensively in device virtualization; each domain has a number of end-points or *ports* each of which may be bound to one of the following *event sources*:

- a physical interrrupt from a real device,
- a virtual interrupts (callback) from Xen hypervisor, or
- a signal from another domain.

Events are lightweight and do not carry much information beyond the source of the notification. Hence when performing bulk data transfer, events are typically used as synchronization primitives over a shared memory transport. Event channels are managed via the `event_channel_op()` hypercall.

8.2. PV Network I/O

8.3. PV Block I/O

8.4. PV PCI

Guest operating systems should not attempt to determine the PCI configuration directly by accessing the PCI BIOS. Instead, they should use Xen hypercall `physdev_op(void *physdev_op)` to perform PCI configuration; it can be PCI config read, write and so on.

9. IOMMU - I/O Memory Management Unit

There are cases when we want to give VM guest pass-through access to a device. First case is when we either can't effectively share the device (display, for example) or we simply want just one guest to have access to it. Second case is when we have Self Virtualized I/O device that can present itself to the system as many devices (for example, SR-IOV NIC Virtual Function).

The pass-through access to a device means that only one VM has access to it, other VMs do not have access to this device and therefore can't use it. There are two major concerns related to pass-through access that have to be handled properly:

- Safe DMA.
- Correct interrupt delivery.

The problem with DMA is due to the fact that VM guest does not know actual machine addresses, it operates with intermediate physical addresses. Therefore it can't properly set up DMA transaction without an additional help from SW driver or HW. To solve this problem CPU designers use I/O Memory Management Unit (IOMMU) that eliminates the need for virtualization-aware drivers for pass-through devices with DMA.

9.1. Operation of IOMMU

An IOMMU is similar to the processor's MMU, but it works from the device's perspective. VM guest communicates with IOMMU capable device using a virtual address, therefore the device initiates a DMA operation also using a virtual address, rather then a physical address. IOMMU translates the address into a machine address. To perform the translation IOMMU uses the same page tables as CPU's MMU.

Tachyxx IOMMU also includes a mechanism for interrupt remapping. A device's interrupts are assigned to a Virtual CPU (vCPU), rather than physical CPU (pCPU). Each interrupt is uniquely identified not only by its interrupt number, but also by the originator ID (derived from the PCI device ID). This pair {IRQ, devID} is then used to generate a mapping to a vCPU number. These virtual interrupts are automatically queued by the hardware, and only delivered when the target vCPU is scheduled (without any involvement from a hypervisor).

9.2. Tachyxx IOMMU implementation

Tachyuxx systems usually have several IOMMUs connected to different PCIe channels. Each IOMMU has I/O MMU and IOTLB. The IOTLB is a four-level lookup to convert virtual to physical addresses. The IOTLB does a lookup and returns a physical address to be used in DMA operation. On IOTLB miss (when IOTLB does not have required VA to PA translation) the IOMMU does the lookup by way of doing HW page walking; the new translation then gets inserted into the IOTLB.

Tachyxx CPU connects to all external devices with PCIe channels. PCIe channels are connected to several PCIe SubSystem (PCIeSS) modules. IOMMU (with IOTLB and I/O MMU) is included into PCIeSS. The other side of PCIeSS module is interfacing memory Mesh; IOMMU is translating an address coming from Mesh. DMA engines are also part of PCIeSS module.

When PCIe Controller inside PCIeSS generates a virtual address, it gets translated to physical by IOTLB (IOMMU address translation happens on the TLB miss).

Tachyxx CPU VA/PA/MA address size:

- The Virtual Address (VA) size is 64 bits.
- The Physical Address (PA) size is 53 bits.
- The Machine Address (MA) size is 53 bits.

9.3. Interrupt Remapping for Virtualization

The I/O device can send Message Signaled Interrupt (MSI-X) by sending write packet to predefined memory location. Without interrupt remapping Hypervisor will have to be involved in creating I/O and in handling interrupts and forwarding them to guest operating system. The IOMMU provides functionality to translate MSI-X memory location and therefore virtualize interrupts.

10. Self-Virtualized I/O and SR-IOV

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.92.9539&rep=rep1&type=pdf

https://www.usenix.org/legacy/event/wiov08/tech/full_papers/dong/dong.pdf

https://www.usenix.org/legacy/event/wiov08/tech/full_papers/levasseur/levasseur_html/#:~:text=The%20Alternative%20Routing%2DID%20Interpretation,is%20assumed%20to%20be%200).

Glossary

mud

wet, cold dirt

rain

water falling from the sky

References

- [popek] GJ Popek. Formal requirements for virtualizable third generation architectures. 1974
- [jeongseob] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. http://calab.kaist.ac.kr/~jhuh/papers/ahn_isca2012.pdf
- [dall] Christoffer Dall, Shih-Wei, LiJason Nieh. Optimizing the Design and Implementation of the Linux ARM Hypervisor.
- [dong] Yaozu Dong, Zhao Yu and Greg Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. 2008 https://www.usenix.org/legacy/event/wiov08/tech/full_papers/dong/dong.pdf

Index