Xen and FreeBSD

BY ROGER PAU MONNÉ

The Xen Hypervisor began at the University of Cambridge Computer Laboratory in the late 1990s under the project name Xenoservers. At that time, Xenoservers aimed to provide "a new distributed computing paradigm, termed 'global public computing,' which would allow any user to run any code anywhere. Such platforms price computing resources, and ultimately charge users for resources consumed".

Using a hypervisor allows for sharing the hardware resources of a physical machine among several OSes in a secure way. The hypervisor is the piece of software that manages all those OSes (usually called guests or virtual machines) and provides separation and isolation between them. First released in 2003 as an open-source hypervisor under the GPLv2, Xen's design is OS agnostic, which makes it easy to add Xen support into new OSes. Since its first release more than 20 years ago, Xen has received broad support from a large community of individual developers and corporate contributors.

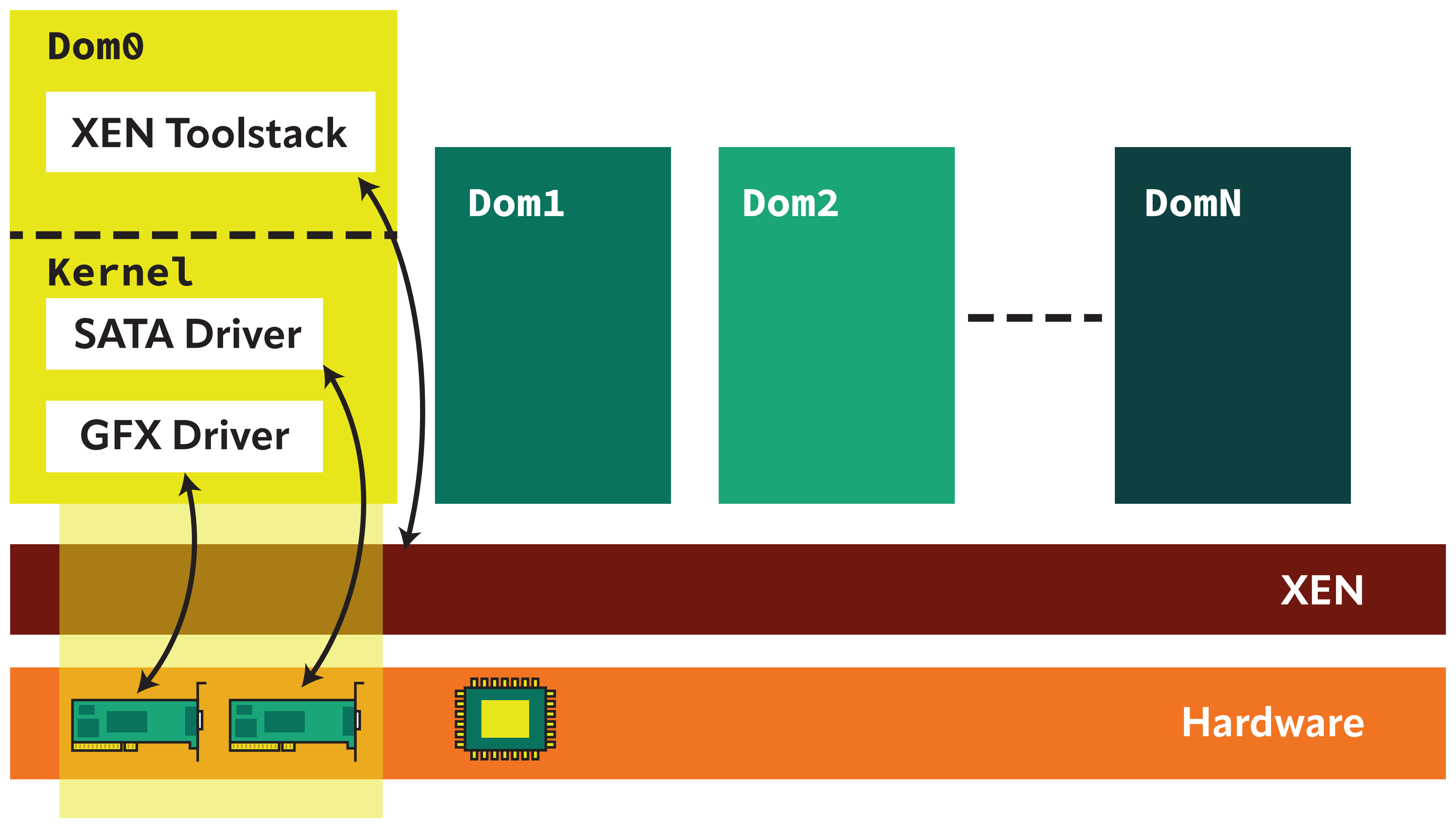
The Architecture

Hypervisors can be divided into two categories:

- **Type 1:** those that run directly on bare metal and are in direct control of the hardware.
- **Type 2:** hypervisors that are part of an operating system.

Common Type 1 hypervisors are VMware ESX/ESXi and Microsoft Hyper-V, while VMware Workstation and VirtualBox are clear examples of Type 2 hypervisors. Xen is a Type 1 hypervisor with a twist — its design resembles a microkernel in many ways. Xen itself only takes control of the CPUs, the local and IO APICs, the MMU, the IOMMU, and a timer. The rest is taken care of by the control domain (Dom0), a specialized guest granted elevated privileges by the hypervisor. This allows Dom0 to manage all other hardware in the system, as well as all other guests running on the hypervisor. It is also important to realize that Xen contains almost no hardware drivers, preventing code duplication with the drivers already present in OSes.

Architecture



When Xen was initially designed there were no hardware virtualization extensions on x86; options for virtualization either involved full software emulation or binary translation. Both options are very expensive in terms of performance, so Xen took a different approach. Instead of intending to emulate the current x86 interfaces, a new interface was provided to guests. The purpose of such a new interface was to avoid the overhead of having to deal with the emulation of hardware interfaces in the hypervisor and, instead, use a new interface between the guest and Xen that's more natural to implement for both. This virtualization approach is also known as Ring Deprivileging.

However, this requires the guest to be aware it's running under Xen, and to use a different set of interfaces compared to running natively. That set of interfaces was designated as ParaVirtualized, and hence the guests that used those interfaces were usually referred to as PV guests. The following interfaces are replaced with PV equivalents on PV guests:

- Disk and network.
- Interrupts and timers.
- Kernel entry point.
- Page tables.
- Privileged instructions.

The main limitation with such an approach is that it requires extensive changes to core parts of the guests kernel OSes. Currently, the only OSes that still have x86 Xen PV support are Linux and NetBSD. There was an initial port of Windows to run as a PV guest that was never published, plus Solaris also had PV support.

With the addition of hardware virtualization extensions to x86 CPUs, Xen also gained support to run unmodified (non-PV) guests. Such guests rely on the usage of hardware virtualization plus emulation of hardware devices. On a Xen system, such emulation is either done by the hypervisor itself (for performance critical devices) or offloaded to an external emulator running in user-space, by default QEMU. This hardware virtualized guests that emulates a full PC-compliant environment is called HVM in Xen terminology.

Instead of intending to emulate the current x86 interfaces, a new interface was provided to guests.

So now we have gone over two very different types of guests, on one side we have PV guests that use PV interfaces to avoid emulation, and on the other side, we have HVM guests that rely on hardware support and software emulation in order to run unmodified guests.

Emulated IO devices used by HVM guests, such as disks or network cards, don't perform very well due to the amount of logic required to handle data transfers and the overhead of emulating legacy interfaces. To avoid this penalty, Xen HVM guests also get the option to use PV interfaces for IO. Some other PV interfaces are available to HVM guests (like a one-shot PV timer) to reduce the possible overhead of using emulated devices.

While HVM allows every possible unmodified x86 guest to run, it also has a wide attack surface due to emulating all devices required for a PC compatible environment. To reduce the amount of interfaces (and thus the surface of attack) exposed to guests, a slightly modified version of HVM guests was created, named PVH. This is a slimmed down version of HVM, where a lot of emulated devices that would be present on HVM guests are not available. For example, a PVH guests only gets an emulated local APIC and maybe an emulated IO APIC, but there's no emulated HPET, PIT or legacy PIC (8259). PVH mode, however, might require modifications in the guest OS kernel so it's aware it's running under Xen and some devices are not available. PVH mode also uses a specific kernel entry point that allows direct booting into the guest kernel, without relying on an emulated firmware (SeaBIOS or OVMF), thus greatly speeding up the boot process. Note, however, OVMF can also be run in PVH mode to chain load OS-specific bootloaders when startup speed is not of great concern and ease of use is preferred. See the table below for a brief comparison of the different guest modes on x86.

	PV	PVH	HVM
I/O devices	PV (xenbus)	PV (xenbus)	emulated + PV
Legacy devices	NO	NO	YES
Privileged instructions	PV	hardware virtualized	hardware virtualized
System configuration	PV (xenbus)	ACPI + PV (xenbus)	ACPI + PV (xenbus)
Kernel entry point	PV	PV + native*	native

* It's possible for PVH guests to re-use the native entry point when booted with firmware, but that requires adding logic to the native entry point to detect when booting in a PVH environment. Not all OSes support this.

The PVH approach has also been adopted by other virtualization technologies like Firecracker from AWS. While Firecracker is based on KVM, it re-uses the Xen PVH entry point and applies the same attack surface reduction by not exposing (and thus emulating) legacy x86 devices.

Speaking about ARM architecture, the fact that the Xen port was developed once ARM already had support for hardware virtualization extensions led to a different approach when compared to x86. ARM has only one guest type, and it would be the equivalent of PVH on x86. The focus is also to attempt to not expose an excess of emulated devices to reduce the complexity and the attack surface.

It's quite likely that the upcoming RISC-V and PowerPC ports will take the same approach of supporting only one guest type, more akin to HVM or PVH on x86. Those platforms also have hardware virtualization extensions that forego the need for something like classic PV support.

Usage and Unique Features

The first commercial uses of Xen were strictly focused on server virtualization, either on premise usage of Xen-based products or through cloud offerings. However, due to its versatility, Xen has now also extended into the client and embedded space. Xen's small footprint and security focus makes it suitable for a wide range of environments.

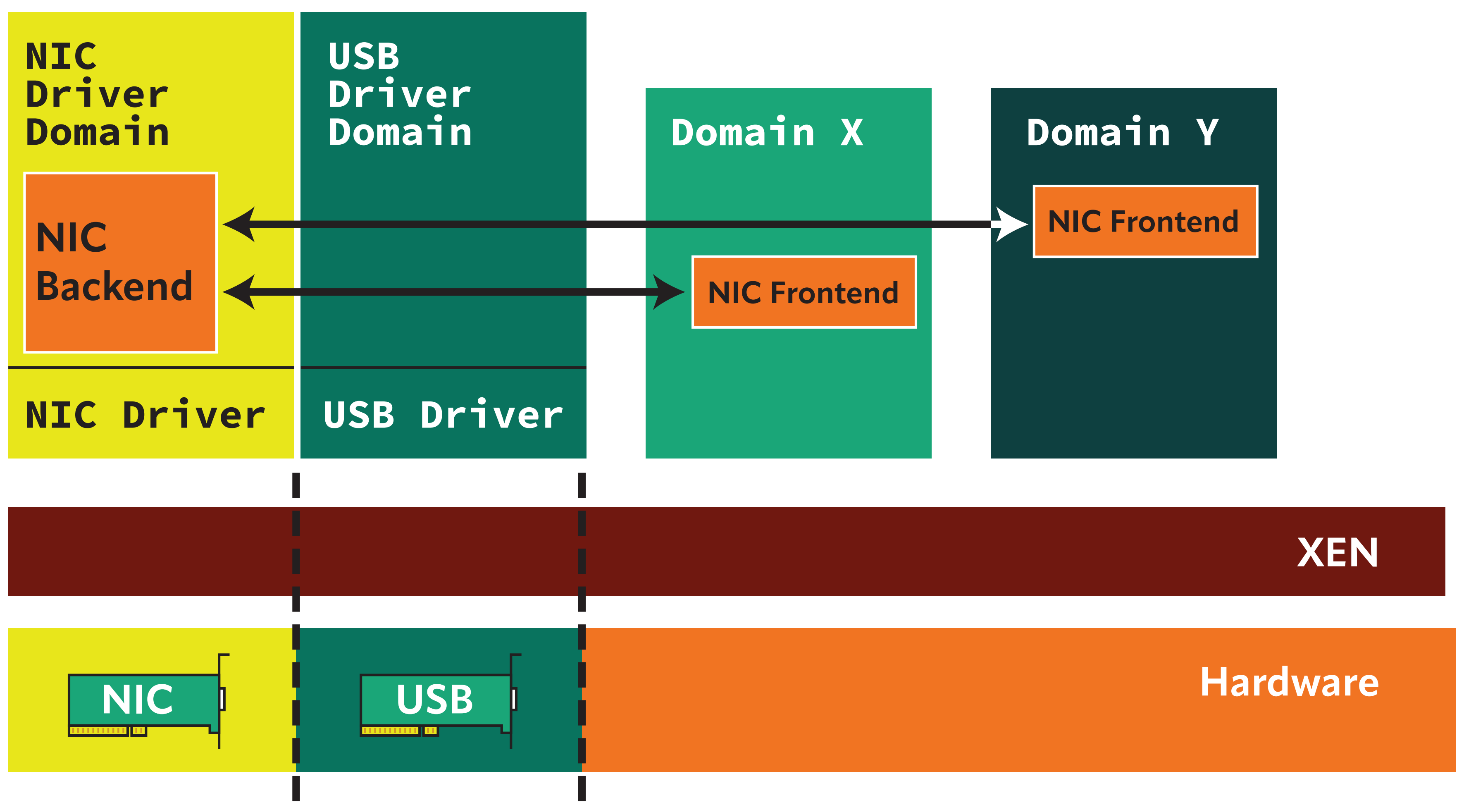
A great example of a client (desktop) usage of Xen is QubesOS, a Linux-based OS that's focused on security through isolation of different processes in virtual machines, all running on top of the Xen hypervisor and even supporting the usage of Windows applications. QubesOS relies heavily in some key Xen-specific features:

- Driver domains: network cards and USB drivers are run in separate VMs, so that security issues from the usage of those devices cannot compromise the entire system. See the diagram about driver domains.
- Stub domains: the QEMU instance that handles the emulation for each HVM guests is not run in dom0, but rather in a separate PV or PVH domain. This isolation prevents security issues in QEMU from compromising the entire system.

- Limited memory sharing: by using the grant sharing interfaces, a domain can decide what pages of memory are shared to which domains, thus preventing other domains (even semi-privileged ones) from being able to access all guest memory.

Similarly to QubesOS there's also OpenXT: a Xen-based Linux distribution focused on client security used by governments.

Driver Domains



A couple more of unique Xen x86 features that are used by diverse products:

- Introspection: allows external monitors (usually running in user-space on a different VM) to request notifications for actions performed by a guest. Such monitoring includes, for example, access to a certain register, MSR, or changes in execution privilege level. A practical application of this technology is DRAKVUF, a malware analysis tool that doesn't require any monitor to be installed in the guest OS.
- VM-fork: much like process forking, Xen allows forking of running VMs. Such a feature still doesn't create a fully functional fork, but it's enough to be used for kernel fuzzing. The KF/x fuzzing project puts the kernel into a very specific state, and then starts fuzzing by creating forks of the guest. All forks start execution at the same instruction, but with different inputs. Being able to fork a VM in a very specific state extremely fast and in parallel is key to achieving a very high rate of iterations per minute.

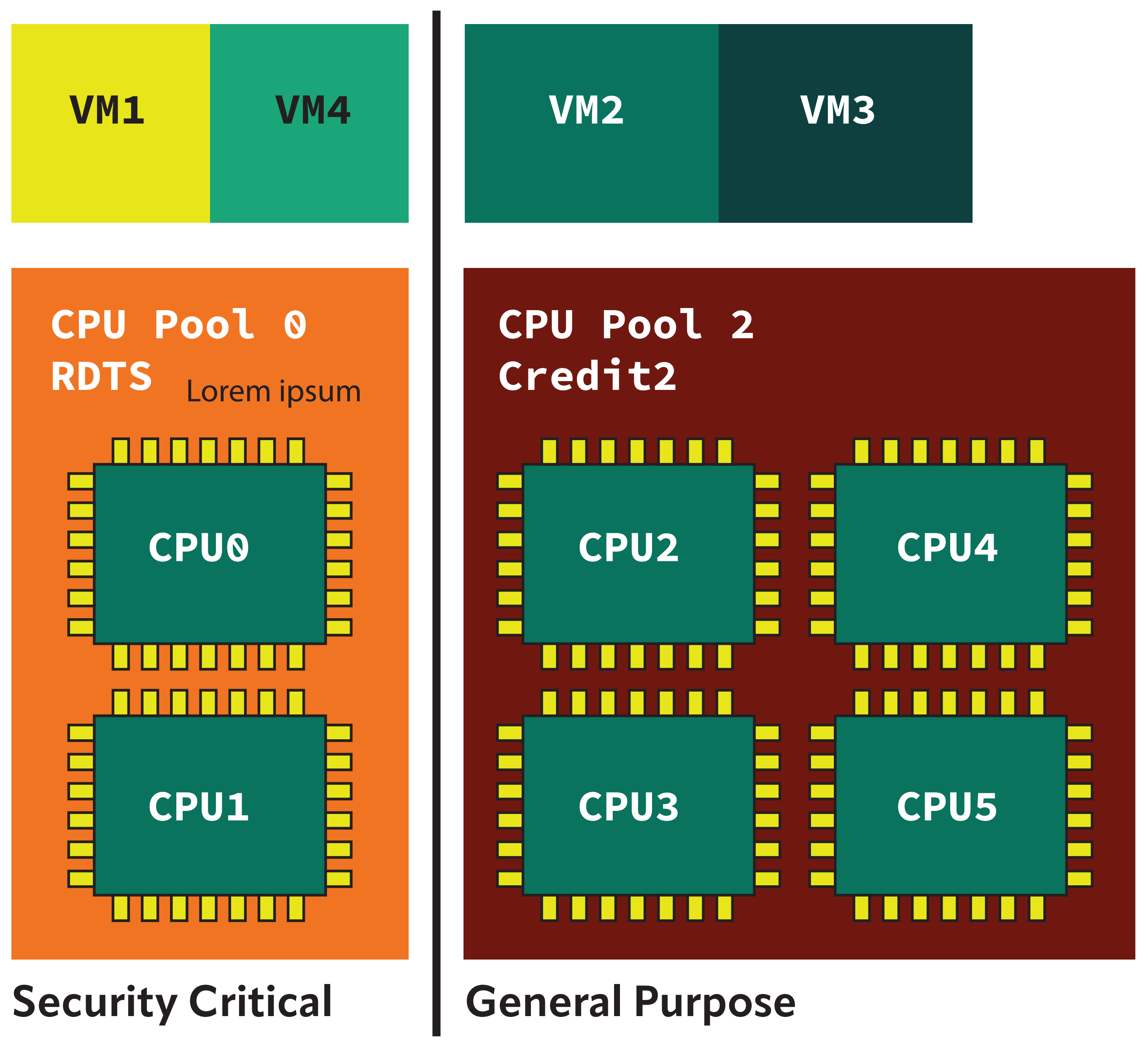
Since the addition of the ARM port, there's been a wide interest in using Xen on embedded deployments, from industrial to automotive. Apart from the small footprint and security focus, there are some key features of Xen that make it appealing for such usages. First, the amount of code in Xen is quite limited when compared to Type-2 hypervisors, so it's conceivable to attempt to safety-certify it. There's currently an effort upstream to attempt to comply with the applicable parts of the MISRA C standard so Xen can be safety certified.

Some unique features that make it very appealing to embedded uses include:

- Small code base: makes it possible to audit and safety certify, also the code base is being adapted to comply with the MISRA C standard.
- cpupools: Xen has the ability to partition the CPUs into different groups and assign a different scheduler to each group. Guests can then be assigned to those groups, allowing for a set of guests that run using a real-time scheduler, like RTDS or ARINC653, while a different set of guests can run using a general-purpose scheduler like credit2. See CPU Pools diagram.
- CPU pinning: it's also possible to apply restrictions on which host CPUs get to schedule which guest CPUs, so, for example, a guest CPU can be exclusively given a host CPU when running latency sensitive workloads.

- Deterministic interrupt latency: significant efforts have been put into Xen to ensure interrupt latency remains both low and deterministic, even in the presence of cache pressure caused by noisy neighbors. There's a patch series currently in review that adds cache coloring support to Xen. Additionally, Xen is being ported to run on Arm-v8R MPU (memory protection unit) based systems. This is a quite significant change in Xen's architecture, as it has always been supported on Memory Management Unit (MMU) based systems. With MPU, there is flat mapping between VA and PA and thus one can achieve real-time effect since there is no translation involved. There are a limited number of memory protection regions that can be created by Xen to enforce memory type and access restrictions on different memory ranges.
- dom0less/hyperlaunch: a feature that originated in ARM and is currently also being implemented for x86 allows multiple guests to be created statically at boot time. This is very useful for static partitioned systems, where the number of guests is fixed and known ahead of time. In such a setup the presence of an initial (privileged) domain is optional, as some setups don't require further operations against the initially created guests.

CPU Pools



FreeBSD Xen Support

FreeBSD Xen support was added quite late compared to other OSes. For instance, NetBSD was the first OS to formally commit Xen PV support because Linux patches for full PV support didn't get merged until Linux 3.0 (around 2012).

FreeBSD had some initial support for PV, but that port was 32bit only and not fully functional. Development on it stopped, and it ended up being deleted from the tree once PVH support was implemented. In early 2010, FreeBSD saw the addition of PV optimizations when running as an HVM guest, which allowed FreeBSD to make use of PV devices for I/O together with the usage of some additional PV interfaces for speedups like the PV timer.

In early 2014, FreeBSD gained support to run as a PVHv1 guest, and shortly after, as a PVHv1 initial domain. Sadly, the first implementation of PVH (also known as PVHv1) was wrongly designed, and had backed in too many PV-related limitations. PVHv1 was designed as an attempt to move a classic PV guest to run inside of an Intel VMX container. This was fairly limiting, as the guest still had a bunch of restrictions inherited from classic PV, and it was also limited to Intel hardware only.

After finding out about those design limitations, work started on moving to a different implementation of PVH. The new approach started with an HVM guest and stripped as much emulation as possible, including all emulation done by QEMU. Most of this work was, in fact, developed with FreeBSD, as that's my main development platform, and I did extensive work in order to implement what was later called PVHv2 and is now plain PVH.

FreeBSD x86 runs as both an HVM and PVH guest and supports running as a PVH dom0 (initial domain). In fact, x86 PVH support was merged earlier in FreeBSD than Linux. Running in PVH mode, however, still has some missing features compared to a classic PV dom0. The biggest one is the lack of PCI passthrough support, which, however, requires changes in both FreeBSD and Xen to be implemented. There's an ongoing effort in Xen upstream to add PCI passthrough support for PVH dom0, however, that's still being worked on, and when finished, will require changes to FreeBSD for the feature to be usable.

On the ARM side, work is underway to get FreeBSD to run as an Aarch64 Xen guest. That required splitting the Xen code in FreeBSD to separate the architecture specific bits from the generic ones. Further work is being done to integrate Xen interrupt multiplexing with the native interrupt handling done in ARM.

Recent Developments in the Xen Community

Apart from the ongoing effort mentioned before that attempts to bring feature parity between a PV and PVH dom0 on x86, there's a lot more going on in upstream Xen. Since the last Xen release (4.19), PVH dom0 has been a supported mode of operation, albeit with caveats due to some key features still missing.

The RISC-V and PowerPC ports are making progress to reach a functional state, hopefully in a couple of releases we might have them reach a state where the initial domain can be booted and guests can be created.

There's a lot more going on in upstream Xen.

At least on x86, a lot of time in recent years has been spent on mitigating the flurry of hardware security vulnerabilities. Since the original Meltdown and Spectre attacks released in early 2018, the amount of hardware vulnerabilities has been increasing steadily. This requires a lot of work and attention on the Xen side. The hypervisor itself needs to be fixed so as not to be vulnerable, but it's also quite likely some new controls need exposure to the guests so they can protect themselves. To mitigate the impact that future hardware vulnerabilities have on Xen, we are working on a new feature called Address Space Isolation (which has also been known as Secret Free Xen), that aims to remove the direct map plus all sensitive mappings from being permanently mapped in the hypervisor address space. This would make Xen not vulnerable to speculative execution attacks, thus allowing the removal of a lot of the mitigations applied on entry points into the hypervisor, and possibly the need to apply more mitigations for any future speculative issues.

Since the beginning of 2021, all Xen commits have been tested for builds on FreeBSD using the Cirrus CI testing system. This has been a massive help to keep Xen building on FreeBSD, as the usage of Clang plus the LLVM toolchain sometimes created or displayed issues that wouldn't manifest when using the GNU toolchain. We currently test that Xen builds on all the supported FreeBSD stable branches, plus the HEAD development branch. Xen recently retired its custom testing system called osstest, and now solely relies on Gitlab CI, Cirrus CI and Github actions to perform testing. This allows for a more open and well documented testing infrastructure, where it's easier for newcomers to contribute and add tests. Future work in that area should include runtime testing on FreeBSD, even if initially using QEMU instead of a real hardware platform.

Recent releases also added toolstack support for exposing VirtIO devices to Xen guests. Both Linux and QEMU currently support using VirtIO devices with grants instead of guest memory addresses as the basis for memory sharing between the VirtIO frontends and backends. This addition hasn't required a VirtIO protocol change, since it's, instead, implemented as a new transport layer. There are also efforts to introduce a transport layer not based on memory sharing, as this is a requirement for some security environments. Going forward, this would allow Xen to use VirtIO devices while keeping the security and isolation that's guaranteed when using the native Xen PV IO devices. The overall goal is to be able to reuse the VirtIO drivers as first-class interfaces on Xen deployments.

Safety certification and the adoption of MISRA C rules has also been one of the main tasks for the past releases. The last Xen release (4.19) has been extended to support 7 directives and 113 rules of a total of 18 directives and 182 rules that conform to the MISRA C specification. Adoption is being done progressively, so that each rule or directive can be debated and agreed upon before being adopted. Given that the Xen code base wasn't designed with MISRA compliance in mind, some of the rules will require either global or local per-instance deviations. Also, as part of the Safety Certification initiative work, it has started adding safety requirements and assumptions of use. Safety requirements provide a detailed description of all the expected behaviors of the software (Xen), enabling independent testing and validation of these behaviors.

The Future of Xen

Looking back at when x86 PVH support was first added on FreeBSD, it's been a long and not always easy road. FreeBSD was an early adopter of PVH for dom0 mode, and a lot of Xen development has been done while using a FreeBSD PVH dom0. It's also notable how FreeBSD has become a first-class Xen citizen in the recent years, as now there is build testing of Xen on FreeBSD for each commit that goes into the Xen repository.

The port of FreeBSD to run as a Xen Aarch64 guest has also gained some traction recently and is certainly a feature to look forward to given the increasing presence of ARM based platforms both on the server, the client, and the embedded environments.

It's good to see Xen being used in so many different use-cases, and so different from its inception design purpose of being focused on server side (cloud) virtualization. I can only hope to see which new deployments and use-cases of Xen will be used in the future.

How to Reach

The Xen community does all the code review on the xen-devel mailing list. For more informal communications and discussions we also run a couple of Matrix rooms free for everyone to access. For FreeBSD/Xen specific questions there's also the freebsd-xen mailing list, and of course the FreeBSD Bugzilla can be used to report any bugs against FreeBSD/Xen.

ROGER PAU MONNÉ is a Software Engineer at Cloud Software Group and a FreeBSD committer. His roles in the Xen community include being a x86 maintainer, part of the Xen Security Team and also a Xen committer. He has done extensive work on the x86 PVH implementation in both Xen and FreeBSD, and now spends most of his time working on security-related features or chasing down bugs.