XEN

BY
ROGER
PAU MONNÉ

THE XEN HYPERVISOR started 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 allows any user to run any code anywhere. Such platforms price computing resources, and ultimately charge users for resources consumed".

Given this goal, it is clear, Xen was designed for the cloud even before the cloud was created. Today Xen technology powers the biggest enterprise clouds in production, including Amazon EC2, RackSpace, and Verizon Terramark.

Using a hypervisor allows 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), 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 10 years ago, Xen receives broad support from a large community of individual developers and corporate contributors.

The Architecture

Hypervisors can be divided into two categories: Type1—those that run directly on bare metal and are in direct control of the hardware, and Type2—hypervisors that are part of an operating system. Common Type1 hypervisors are VMware ESX/ESXi and Microsoft Hyper-V, while VMware Workstation and VirtualBox are clear examples of Type2 hypervisors.

Xen is a Type1 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 (either HPET or PIT). The rest is taken care of by the control domain (Dom0), a spe-

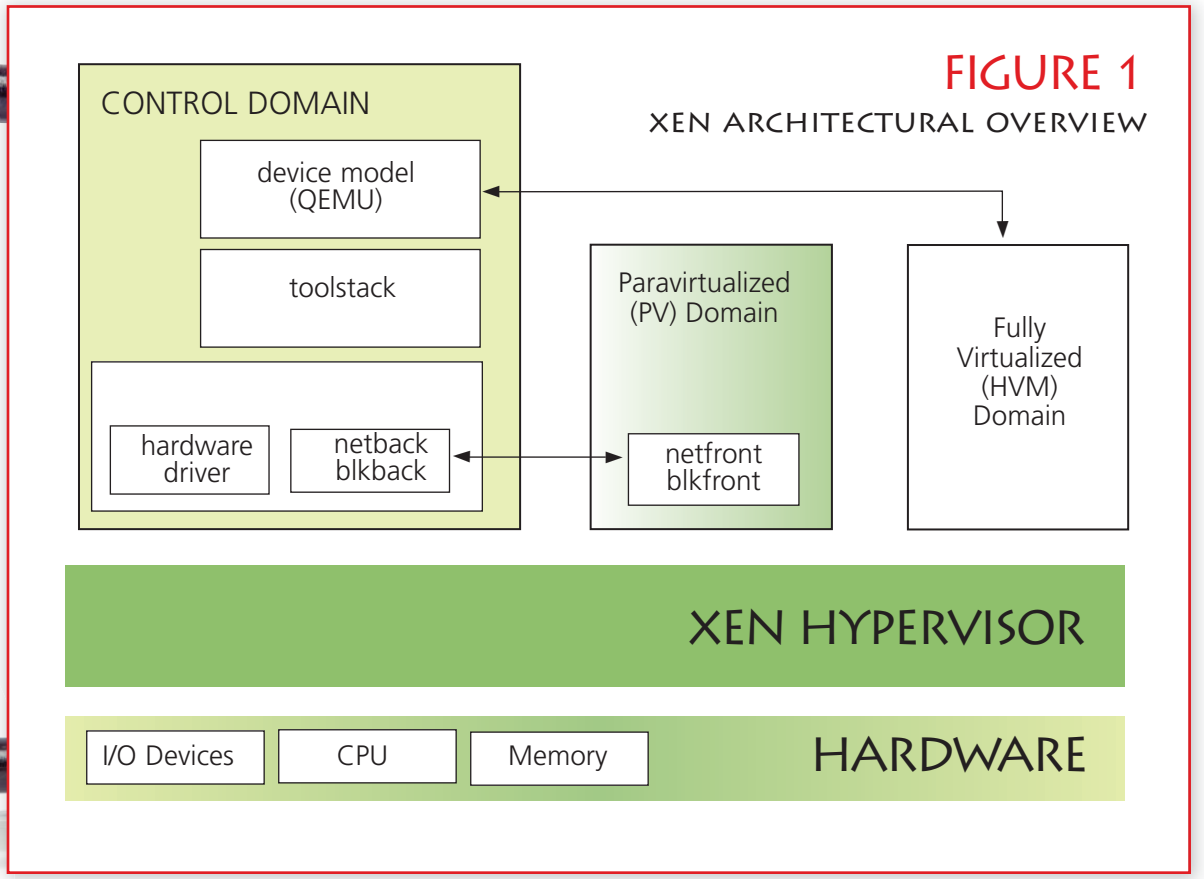

FIGURE 1
XEN ARCHITECTURAL OVERVIEW

cialized 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 (See figure 1).

The Guests

Because Xen was designed back when x86 didn't have the hardware features it has now, it is able to offer several different virtualization modes. In the late 1990s, there were only two options to use virtualization on x86, both with very high overhead—full software emulation or binary translation. To overcome this, Xen took a new approach. We made the guest aware that it was running inside a virtualized environment and provided a whole new interface that removed the extra overhead. This led to what is known today as paravirtualization (PV), and replaced the following interfaces with PV-aware implementations:

- Disk and network
- Interrupts and timers
- Boot process, the guest starts directly in the mode it wishes to run (either 32 or 64 bits)
- Page tables
- Privileged instructions

Some of the interfaces listed above are easy to implement and are not intrusive regarding the guest OS, such as PV disks and nics. On the other hand, some are really intrusive, such as the replacement of the native page table implementation.

A couple of these interfaces are worth explaining in more detail, like the paravirtualization of interrupts. PV guests are not allowed to use native interrupts, so a new technique called event channels was introduced to replace them. Event channels use a single entry point into the guest kernel to inject interrupts, and there's a shared memory region between Xen and the guest to signal which event has fired. All interrupts are routed over this interface when running as a Xen PV guest.

Another important technique used by Xen guests is the hypercall page. This is a memory page in the guest OS that's filled by Xen and contains the hardware-specific hypercall implementations. Hypercalls are much like system calls between user-space and kernel-space in OSes, but in this case the hypercall is between

Figure 2
THE VIRTUALIZATION SPECTRUM

	DISK & NETWORK	INTERRUPTS & TIMERS	EMULATED MOTHERBOARD	PRIVILEGED INSTRUCTIONS & PAGE TABLES
HVM	VS	VS	VS	VH
HVM WITH PV DRIVERS	PV	VS	VS	VH
PVHVM	PV	PV	VS	VH
PVH	PV	PV	PV	VH
PV	PV	PV	PV	PV

POOR PERFORMANCE
ROOM FOR IMPROVEMENT
OPTIMAL PERFORMANCE

VS SOFTWARE VIRTUALIZATION
VH HARDWARE VIRTUALIZATION
PV PARAVIRTUALIZED

the guest kernel and the hypervisor itself. Hypercalls are the only way a guest OS is able to communicate with the Xen hypervisor.

With the introduction of hardware virtualization extensions in x86 in 2005, Xen gained the ability to run unmodified guests in Hardware Virtual Machine (HVM) mode. This was a very important step because it allowed Xen to run guests without any PV-aware interfaces. In order to do this, a device model is needed (which usually runs in Dom0) that emulates the devices provided to the guest. All this emulation is handled by Qemu, which was adapted to work with Xen.

Using device models is expensive for both the Dom0 and the guest. Since each guest needs its own Qemu instance if all of them are running on Dom0, it can easily become a bottleneck in terms of CPU and memory usage. From a guest point of view, it also adds more overhead when compared to using PV interfaces only, because accesses to these emulated devices cause traps into Qemu.

While this separation between PV and HVM guests makes a clear cut, there have been several PV-specific improvements made available to HVM guests to obtain better performance. HVM guests can make use of PV disks and nics to boost IO throughput. When a guest makes use of those interfaces inside an HVM container, it is known as HVM with PV drivers in the Xen argot. But it doesn't stop here, since Xen 4.1, a HVM guest, can also use PV timers and PV IPIs to reduce even more emulation overhead. When a guest runs in this mode, it's known as PVHVM.

In general, HVM guests have better performance, especially regarding page table manipulation operations. The software page table manipulation used in PV guests is one of the main performance problems of pure PV guests. To improve this, a new mode has been recently introduced that allows PV guests to run inside HVM containers. This new mode is called PVH and makes use of the hardware virtualization extensions for the CPU and MMU, while using PV interfaces for the rest. **Figure 2** contains a table that shows the differences between the several guest modes supported by Xen.

New Features in Xen 4.4

Apart from the usual round of bug fixes and across-the-board improvements, the latest release of Xen includes several improvements worth mentioning. On the tools side, the default Xen toolstack (libxl) offers improved libvirt support. This lays the foundation for solid integration into any tools that can use lib-

virt, from GUI VM managers to cloud orchestration layers like CloudStack or OpenStack.

The Xen on ARM port also saw a number of improvements, as now Xen is able to both run on ARM 64-bit hardware and also support ARM 64-bit guests. The ARM ABI (the interface between the guests and the hypervisor) has also been declared stable, which means all changes will be done in a backwards compatible fashion, and ARM guests using the Xen 4.4 interface can rest assured this will continue to work on newer releases. Also this release added support for many new boards including Arndale, Calxeda ECX-2000, Applied Micro X-Gene Storm, TI QMAP5 and Allwinner A20/A30. There's already ongoing work to port FreeBSD as both a guest and a Dom0 for Xen on ARM.

One of the most interesting 4.4 features relevant to FreeBSD is the new virtualization mode called PVH. Xen 4.4 has experimental support for running non-privileged guests in PVH mode, with Dom0 PVH support coming in Xen 4.5. This new virtualization mode is very similar to PVHVM, a mode FreeBSD can run as of FreeBSD 10, but removes the need for any software emulation at all. This means there's no need to run a device model in order to run a PVH guest (something that is needed for PVHVM guests). Since PVH doesn't require a device model to run, it can also be used as Dom0, something that until now was only possible with pure PV guests.

PVH guests make heavy use of the hardware virtualization extensions found in current processors as it runs inside a HVM container and has access to a hardware-virtualized CPU and MMU, which means there are no PV interfaces for page table manipulation or privileged instruction execution. As said before, one of the hardest things about adding PV support into an OS is the fact that the virtual memory subsystem has to be rewritten or completely filled with hooks for Xen, which is both hard to design and hard to maintain. Apart from the simplification of the PV interface, PVH is also much faster regarding page table manipulations, since using the hardware-virtualized MMU is certainly faster than using the PV MMU. Although originally designed with performance in mind, PVH has turned out to be a very important feature for FreeBSD, greatly simplifying the path from PVHVM to being able to run as a Dom0. This means that FreeBSD can bypass all the pain of implementing PV MMU support without any downsides. Performance of PVH is better than the performance of pure PV, and it will also allow FreeBSD to run as Dom0. Due to all these benefits, it is very likely that the embryonic i386 PV port will be removed in favor of PVH.

Xen Support In FreeBSD 10

The 10 release cycle was quite interesting in terms of new Xen features. The FreeBSD/Xen port status in 9 included support for running as a uniprocessor 32-bit PV guest and as a HVM guest with PV drivers for both disk and nics. Most of the work during the last release focused on getting FreeBSD to run as a PVHVM guest and also improving the performance and features of the PV drivers. To run as a PVHVM guest, several under-the-hood improvements were done, and while those are not quite visible from a user point of view, they were crucial to getting us where we are now.

One of the first and most important improvements is the change in how event channel interrupts are injected and handled in FreeBSD. In previous releases, events for all event channels were signaled to the guest via a single, global PCI interrupt. This is simple and works fine if the guest is only using event channels for a few PV disk and nic devices. However it scales poorly since all event channel processing is tied to a single interrupt, running on a single CPU. In order to solve this limitation, support for Xen's newer event delivery scheme was added to FreeBSD. Known as "vector callback," Xen allows the guest to allocate an IDT vector for each event channel. Since Xen can inject an IDT event to a specific CPU, this allows full distribution of interrupt load across all CPUs. It also makes it possible to efficiently implement new, per-cpu, device types that are paravirtualized.

Thanks to the introduction of the vector callback, it is possible to use the PV timer. This timer is implemented as a one-shot-per-cpu event timer that is set using hypercalls and is delivered to the guest using an event channel interrupt. This removes the overhead of using the emulated timers, which comes from the fact that you need to perform reads and writes to the emulated devices registers, which cause VMEXITs into Xen. A VMEXIT is an involuntary trap into the hypervisor that involves a context switch. As with any context switch, the saving and restoring of execution state is costly and should be avoided when possible. VMEXITs occur anytime a guest attempts a privileged

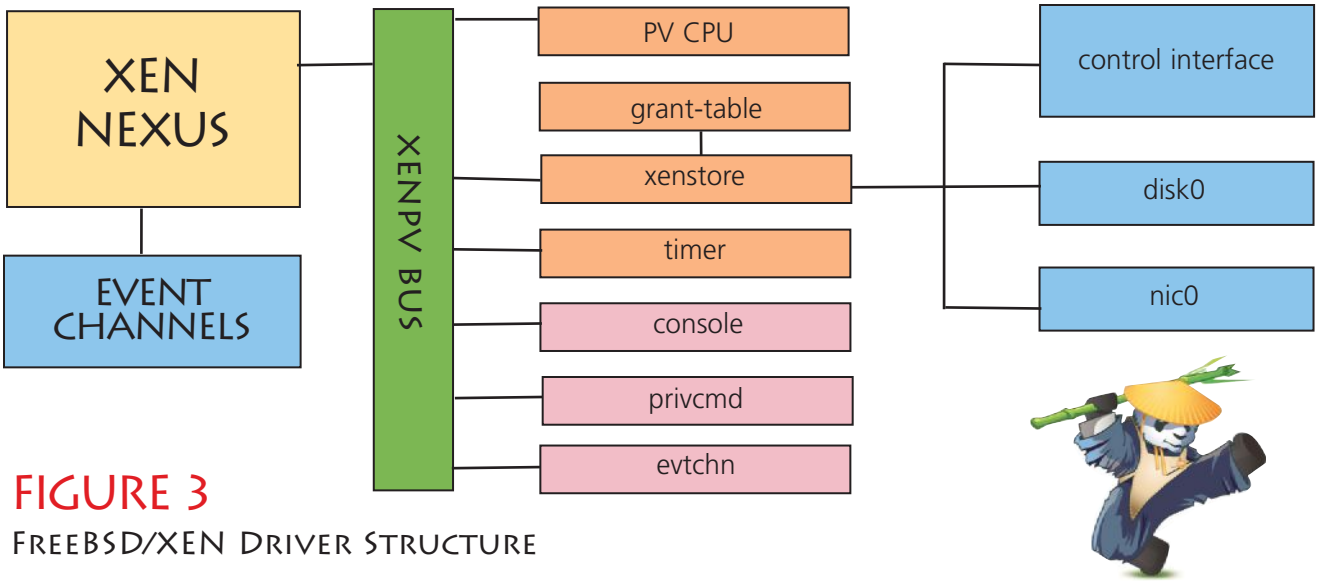

FIGURE 3
FreeBSD/XEN Driver Structure

operation: accessing certain registers, executing certain instructions, or accessing a memory location being managed by the hypervisor, like the memory of devices emulated by Xen.

By using the PV timer only one VMEXIT is taken when issuing the hypercall to set the timer. Xen also provides a shared memory region that contains a lot of time-related information. In addition to setting the PV timer, this information is also used to create a time counter and to provide a clock implementation for FreeBSD. As we can see, we can solve all the OS time-related needs using PV interfaces only.

Another improvement that was made to reduce the emulation overhead is to route inter processor interrupts (IPIs) over event channels. On native x86 hardware, IPIs are delivered using the local APIC, but again when running on a virtualized environment, accesses to the local APIC cause VMEXITs into Xen, which ideally we like to avoid. This is solved by using event channel interrupts to deliver these signals between processors, which greatly reduced the latency of IPIs. Here again we turn multiple VMEXITs into a single hypercall, reducing the emulation overhead.

PVH Support in Head

Since PVH is very similar to PVHVM in terms of the PV interfaces used, getting PVH support into FreeBSD was not a big deal. The main difference between PVHVM and PVH is that PVH uses the PV start sequence. This means that there are no emulated BIOS, and the guest is started by directly jumping into the kernel entry point with some basic page tables already set up by Xen on behalf of the guest. The FreeBSD i386 PV port already had this entry point; however most of the code there was not suitable for PVH, due to it being a fork of the i386 machdep code with the Xen-specific implementation hardcoded in it. We wanted to avoid this as much as possible, since the original idea was to have PVH support in the GENERIC kernel, so that the user would not be required to compile a custom kernel, and could use freebsd-update inside of a PVH guest without problems.

So the first step was to add the PV entry point and some elf notes that are used by Xen in order to know how to load the kernel. This is quite straightforward, and only requires a couple of assembly lines. After that, a specific Xen initialization function is called that sets the page tables as FreeBSD expects to find them (as set by the boot trampoline on native amd64). This function also initializes a couple of Xen-specific global variables with data provided by the hypervisor at boot time.

After that, the native FreeBSD initialization function is called, which has also been slightly tweaked to prevent it from using any emulated devices. Instead of adding a bunch of Xen-specific conditionals in common code, another approach was taken: strategically placed hooks that allow the runtime selection of the proper support code. By default, the hooks point to the native hardware implementation. But if Xen PVH mode is detected, the Xen code is activated. The early initialization hooks provide a dynamic way to change the clock source used during early bootstrap (8259 PIC vs. hypercall) and the method for fetching the hardware memory map (E820 vs. hypercall).

As PVH guests don't have a local APIC, the

way to start the secondary CPUs is also different from native, so another hook had to be added to implement a PVH specific method to start them. On PVH, the way to start secondary CPUs is quite easy (less convoluted than on native I would say), and basically involves setting the initial values of the registers and starting the CPU. This removes the need for any kind of trampoline to set up the page tables, since the CPU is started directly with paging enabled.

Finally all the local APIC-related functions have been converted into hooks, which allows Xen-specific overrides for some of them. Most of them should never be called when running as a PVH domain (because there's no local APIC) so some asserts have been added to make sure no other subsystem tries to use them.

The addition of PVH support also included some major rework of how the Xen code is structured in FreeBSD. Until now, all top-level Xen devices were directly attached to the Nexus. With the introduction of PVH support, the code has been structured in a more hierarchical way by introducing a Xen-specific bus that all top-level Xen devices hang from. **Figure 3** shows how the new xenpv bus is structured, and how descendant devices are organized. Top-level Xen devices include the PV console, the PV timer, and xenstore. Xenstore is a very important component in PVH guests, because all the hardware description comes from it (there are no ACPI tables in PVH). As seen on the diagram, the PV disks and nics hang off xenstore.

Since there is no ACPI on PVH, FreeBSD was falling back to the legacy Nexus implementation. This is not desirable, because it attaches a bunch of buses not present on PVH. So a very simple Xen-specific Nexus is provided in the PVH case that fills this role. Finally, two device drivers were added: the evtchn device, which allows user-space applications to bind event channels, and the privcmd device, present only when running as Dom0, that is used to send management commands from user-space to the hypervisor.

The Future

With PVH support already merged in HEAD, the work has now shifted to PVH Dom0 support. Running as Dom0 is quite different from running as a guest. One of the main differences is that Dom0 has access to the physical hardware, and so it must manage it. But it's not identical to running on bare metal. Two examples of the many differences are physical hardware interrupts delivered via Xen event channels, and the guest's need to set up its physical interrupts using hypercalls.

This might sound very convoluted, but it's actually not that difficult. Xen provides hypercalls that allow Dom0 to set up IO APIC, MSIs and MSI-Xs interrupts in an easy way and without having to deal with the underlying hardware. Again, all of this has been implemented with hooks into the existing code, using the infrastructure provided by newbus, which makes it trivial to replace certain methods with their Xen counterparts. Thanks to this interface, no changes are required to drivers at all.

Regarding the Xen-specific code, some modifications are needed to xenstore in order to run as Dom0. Xenstore is an information storage space shared between domains maintained by the xenstored application. When running as a guest, xenstore contains the hardware description for the domain and is always accessible, but that's not the case for Dom0, since xenstore has not yet been started.

So in Dom0 we have to allow the boot process to skip the xenstore initialization until the daemon has actually been started, and we have to prevent any Xen kernel driver from trying to use xenstore until it is actually running. That's solved by not attaching the xenstore bus until the process has been launched, and thanks to the hierarchical structure of the Xen components in FreeBSD, it's easy to accomplish: just hold off attaching any driver that hangs off xenstore until it is actually initialized.

FreeBSD already has the necessary drivers for providing network and disk services to guests. Thanks to that, there's not much work to do. Some of those backends haven't seen much use, so they will probably need some fine tuning, but that's much less work than actually writing them from scratch.

Dom0 support is still in its early stages, but overall it looks very promising. FreeBSD is heavily focused on performance, especially IO performance, which is exactly what Dom0 needs, since it usually runs a bunch of PV backends to serve requests from guests. Having cutting-edge features like ZFS and Netmap inside the kernel is certainly going to make FreeBSD a very interesting OS within the Xen Project ecosystem. ●

Roger Pau Monné is a Software Engineer at Citrix and a FreeBSD developer. He usually contributes to the Xen Project and the FreeBSD/Xen port, and right now is mainly focused on getting stable PVH support in both projects.