Virtualisation and Open Source

What Makes It the Right Match?

Virtualisation continues to be a buzzword. The idea isn't new in itself; IBM mainframes have always had to have a hypervisor. It's in the news now because even simple desktops can now act as virtual machine hosts. A lot of possibilities have opened up as a result. Let's take a brief tour of what virtualisation means, in its classic sense, and look at why open source virtualisation is going to win.

Well, we all know what virtualisation is about—we've read and heard of it over and over again. But let's look at it from the view point of the open source world.

Virtualisation means the simulation of a computer system, in software. The virtualisation software creates an environment for a guest, a complete OS, to execute within this created world. This means the view that should get exported to the guest should be of a complete computer system—with the processor, system peripherals, devices, buses, memory and so on. The virtualisation software can be strict about what view to export to the guest, for example, the processor and processor features, types of devices, buses exported to the guest, etc, or it can be flexible with the user getting a choice to select individual components and parameters.

There are some constraints to creating a virtualised environment or a set of sufficient requirements, as has been noted by Popek and Goldberg in their paper on virtual machine monitors.

- Fidelity: Software running in a virtualised environment should not be able to detect it is running on a virtualised system.
- Containment: Activities within a virtual machine (VM) should be contained within the VM itself without disturbing the host system. A guest should not cause the host or other guests running on the host to malfunction.
- Performance: Performance is crucial to how the user sees the utility of the virtualising environment. In this age of extremely fast and affordable general-purpose computer systems, if it takes a few seconds for some input action to get registered in a guest, no one will be interested in using the virtual machine at all.
- Stability: The virtualisation software itself should be stable enough to handle the guest OS and any quirks it may exhibit.

There are several reasons why one would

want virtualisation. For data centres, it makes sense to run multiple servers (Web, mail, etc,) on a single machine. These servers are mostly under-utilised, so clubbing them on one machine with a VM for each of the existing machines enables fewer machines, less rack space and lower electricity consumption.

For enterprises, serving users' desktops on a VM simplifies management, IT servicing, security considerations and costs, by virtue of the reduced expenditure on desktops.

For developers, testing code written for different architectures or target systems becomes easier, since access to the actual system becomes optional. For example, a new mobile phone platform can be virtualised on a developer machine rather than actually deploying the software on the phone hardware each time, allowing for the software to be developed along with the hardware. The virtualised environment can also be used as validation for the hardware platform itself, before going into production, to avoid the costs that arise later due to changes that might be needed in the hardware.

There are several such examples that can be cited for any kind of application or use-case. It's not impossible to imagine a virtualised system being beneficial anywhere a computer is used.

Now is a good time to get acquainted with some of the terms (the mandatory alphabet soup) that we'll be using throughout the article:

- VM – virtual machine
- VMM – virtual machine monitor
- Guest OS – the OS that is run within a VM
- Host OS – the OS that runs on the physical computer system and hosts guests
- Paravirtualised guest – the guest OS that is modified to have the knowledge of a VMM
- Full virtualisation – the guest OS is run unmodified in this environment
- Hypervisor – an analogous term for a VMM
- Hypercall – infrastructure, via which a paravirtualised guest and the VMM communicate

Types of VMM

There are several virtual machine monitors available. They differ in various aspects like scope, motivation, and method of implementation. A few types of monitor software are:

- ‘Native’ hypervisors: These VMMs have an OS associated with them. A complete software-based implementation will need a scheduler, a memory management subsystem and an IO device model to be exported to the guest OS. Examples are: VMWare ESX server, Xen, KVM, and IBM mainframes. In IBM mainframes, the VMM is an inherent part of the architecture.
- Containers: In this type of virtualisation, the guest OS and the host OS share the same kernel. Different namespaces are allocated for different guests. For example, the process identifiers, file descriptors, etc, are virtualised in the sense that a PID obtained for a process in the guest OS will only be valid within that guest. The guest can have a different userland (for example, a different distribution) from the host. Examples are OpenVZ, FreeVPS and Linux-Vserver.
- Emulation: Each and every instruction in the guest is emulated. It is possible to run code compiled for different architectures on a computer—for example, running ARM code on a PowerPC machine. Other examples are qemu and pearpc. qemu supports multiple CPU types, and it runs ARM code under x86 as well as x86 under x86, whereas pearpc only emulates the PPC platform.

Virtualisation on x86

Virtualising the x86 architecture is difficult to do since the instruction and register sets are not compatible with virtualisation. Not all accesses to privileged instructions or registers raise a trap. So we either have to emulate the guest entirely or patch it at run-time to behave in a particular way. This was true till about four years back, before virtualisation-specific instructions were added to the architecture.

With the two leading x86 processor manufacturers, Intel and AMD, adding virtualisation extensions to their processors, virtualising the x86 platform seamlessly has become easier. The ideas behind their virtualisation extensions are more or less the same, with the implementation, instructions and register sets being slightly different.

The new extensions add a new mode, the 'guest-mode', in addition to the user-mode and kernel-mode that we had (ring -1 in addition to the rings 0-3, with the hypervisor residing in ring -1). The implementations also enable support for hiding the privileged state. Disabling interrupts while in the guest mode will not affect the host-side interrupts in any way.

Open source virtualisation

Now that we've seen what virtualisation is about and what's needed on the software side to present a virtual machine to a guest operating system, let's talk about the strides open source software has been making in this field.

Xen was the first open-source hypervisor to be announced. The Xen project was started when hardware extensions to virtualisation were not yet available, and the developers took the paravirtualisation approach towards virtualising a system. The Xen team created a new hypervisor, taking bits from the Linux kernel, to run modified Linux guests. A privileged Linux guest called the Dom0, has access to the system hardware and arbitrates the access to physical resources by guest operating systems.

The Xen project got wide acceptance and was backed by a large number of companies—developers from IBM, Red Hat, Novell, Intel, AMD, all contributed to the Xen code base. It was even included in enterprise Linux offerings from various distributions as the supported virtualisation technology.

When the hardware manufacturers on x86 started adding virtualisation extensions to the processors, unmodified guests could be made to run on hypervisors. The deficiencies of the x86 instruction set were masked by these advances.

With this advancement, along came a new line of thought: why have a separate hypervisor, when all a hypervisor has to do is schedule guests, manage memory and arbitrate access to hardware?

The Linux kernel has been doing all of this for years. Therefore, the kernel code could easily be leveraged to perform all these tasks. And the addition of code to handle the new CPU instructions and state would make Linux itself function as the hypervisor and host VMs.

The KVM project was started for doing just this, and it was evident by the quick developer acceptance that this really is how virtualisation on Linux was finally going to be acceptable. The KVM project was announced in late 2006 and was accepted the same year in Linus' kernel tree. On the other hand, the Xen Dom0 code has yet to find upstream acceptance. The Xen hypervisor, itself bearing Linux code, will always continue to be a separate project.

The open source advantage

A commonly-cited advantage of open source software is the ‘more eyeballs’ concept. As more people look at the code, bugs become more obvious and get fixed faster, often before the code enters a stable release. This is definitely true. However, there are other advantages when it comes to open source software with large communities, beyond just more eyeballs.

If one follows the LWN.net "Who develops Linux" articles, it's clear that most of the developers are sponsored to work on Linux by companies. It isn't a big surprise to people any more that companies are running businesses and making profits by relying on open source software. Linux already runs on the widest array of platforms—it can run on simple embedded devices and also on big supercomputers, including everything in between. The developers come from not just one part of the world, but from everywhere. The experience, culture and insights they all bring in are invaluable.

Contrast this to a proprietary OS maker. Perhaps all the developers sit in one campus and are probably used to following a particular train of thought. Just one company cannot match the resources that 50 companies (and, of course, the individuals in the community) put together to collaboratively enhance the OS.

Red Hat, IBM, Novell, Intel, AMD, HP, Fujitsu, Oracle, Nokia and Google, all figure on the latest LWN.net compilation for companies that are funding developers to contribute to the Linux kernel. The sheer scale at which the development happens is mind-boggling.

This, however, does not mean that companies can push whatever code they want to into the repositories. Merit wins. There is a peer review of all the patches that flow in. There are people who deeply care about the code that gets accepted. Almost all the patches submitted the first time have to be adjusted after review comments by others. There hardly are patches that go in their unmodified form from the time they were first sent out for review. In many cases, people maintaining subsystems that reject patch submissions could be working for the same company that's promoting the patches. And there's no love lost. Everyone involved understands the prime cause: to create better software. People understand this, and the companies involved understand this too.

Now why does all this matter in the virtualisation perspective? It's simple. The Linux kernel itself is a hypervisor. Any advances in Linux, the operating system, are directly beneficial to Linux, the hypervisor. By using the KVM technology, guests running on top of KVM can enjoy the benefits immediately when patches get accepted to Linux. KVM guests can already enjoy the support of 64 vCPUs (and more!), huge-page backed memory, a wide range of memory over-commit options, NUMA support and so on. And KVM is just four years old. It has taken other projects many more years to reach the state they currently are in, and even then, they do not offer some of the features that KVM offers. It's an interesting exercise for the reader: compare the feature set as announced in releases of virtualisation software one year back to the current set. The number of features and enhancements KVM can provide in one year's time, others would only dream of achieving in five.

This comes as no surprise. There are two basic mindsets at play. First, the UNIX one: 'Do one thing and do it right'. The KVM developers just focused on providing the best support to exploit the hardware support for virtualisation and left the CPU scheduling, memory management, etc, to Linux. KVM also leveraged the QEMU project heavily that provides a device model. The virtual computer that gets exposed to the guest is provided by QEMU, and KVM developers have heavily updated the upstream QEMU code to enable it to support modern devices, KVM-based guests and a lot of optimisation.

The second philosophy is to contribute as much as possible to upstream software, fighting the urge to ship a forked copy of the codebase with some features that would be deemed controversial upstream, or which would take a longer time to gain acceptance. This might result in some features getting delayed as discussions pan out, and developers pitch in with their opinions on how to do things the ‘right’ way. But, in the end, the best technical solution wins and maintaining the solution that's accepted by all is easier in the long run. With most enterprise Linux vendors offering seven-year support guarantees, this becomes a big plus. This is because keeping the private functionality in the stable offering working, while also backporting fixes and optimisations from an upstream codebase that changes more and more each day, would soon become a nightmare for the maintainers of the enterprise software.

Just comparing the two open source virtualisation solutions, Xen and KVM, shows us the stark contrast in these principles and the benefits of collaborative development. END

By: Amit Shah

The author is part of the virtualisation team at Red Hat and is excited to be a part of the technology that’s rediscovering commodity x86 servers.