

Contents of This White Paper

Virtual Desktop Value Proposition	2
Virtual Machine Overview	3
Virtual Desktop Components	4
Virtual Machine Monitor.....	4
Host OS.....	5
Guest OS.....	6
Guest Applications.....	6
Guest Data	6
Guest Preferences/Settings.....	6
VMM Program	7
Overview of Virtual Machine	
Deployments	7
Hypervisors	7
Types of Hypervisors.....	7
Hardware Virtualization Support in	
Modern CPUs.....	10
Paravirtualizers	12
Guest OS Modification.....	12
Application Containers	14
Emulators	16
Host Hardware Emulation.....	17
Workspace Virtualization.....	17
Summary	19

Introduction to Virtual Desktop Architectures

Enterprises have embraced various forms of virtualization during the past few decades.

Server virtualization, made popular in the early 2000s through efficient hypervisor technology, made it possible for enterprises to do more with less – increasing datacenter capacity by allocating server workloads on otherwise unused or underused servers. This directly translated into reduced operating costs due to lower total server count, lower power consumption, and lower heat dissipation in the data center.

Storage virtualization, another widely used and accepted technology, allows system administrators to pool storage in low-cost arrays and distribute it to servers as required. This flexible pooling provides a “no waste” operational configuration. By eliminating wasted/unused storage, system administrators can make more efficient use of the storage budget allocated to them.

Desktop virtualization is the current wave of virtualization technology. Desktop virtualization aims to separate the workspace (an end user’s applications, data, network traffic and settings) from the architectural layers below it. Desktop virtualization, while similar in concept to existing virtualization solutions, has additional functional requirements that must be met for the deployment to be considered successful.

In this paper, we will describe several desktop virtualization architectures, and the success criteria required for each.

Virtual Desktop Value Proposition

Virtualized desktops can provide many tangible benefits to an enterprise, the most important being grouped into several high-level categories which include:

- **Enabling crossover of professional and personal devices** – Increasing numbers of enterprise users work on PCs in their professional and personal life. The line between work computer and home computer is becoming blurred, with personal apps and data finding their way onto enterprise PCs and vice versa. This creates an insecure and unmanageable paradigm which can easily be addressed by virtual desktops that separate personal material from corporate. Some enterprises have taken this scenario to the extreme with “employee-owned PCs,” whereby the user works on their home machine with a corporate environment provided via a virtual desktop. This paradigm reduces IT help desk costs by shifting the burden of PC support to the employee and the vendor of the PC.
- **Fostering mobility** – By decoupling the workspace and associated applications, data and resources from the hardware, virtual desktops provide an enterprise with a true “work anywhere, anytime” solution for employees, increasing productivity. A mobile virtual desktop solution can allow employees to use their enterprise desktop anywhere there is a PC.
- **Lowering TCO** – By separating the desktop from the lower layers, administrative costs are reduced, while functionality is increased. Applications deployed at a workspace level, without regard for the underlying host operating system, reducing application provisioning and compatibility issues.
- **Enabling Disaster Recovery** – A robust disaster recovery solution can be architected using virtual desktops. Best used in conjunction with mobile virtual desktops, an employee’s entire workspace can be encapsulated on a portable storage device, laptop or stored on the network for use off-site or off-campus if the need arises due to unforeseen outages in business continuity.
- **Providing security** – A key value proposition of any virtual desktop solution, security is directly related to a vital technology feature of virtualization – isolation. Isolation as a technical underpinning of virtualized desktops will be detailed in a subsequent section.

Virtual Machine Overview

Virtual Machine (VM) technology, the foundation for all virtual desktop solutions, allows a computer to simultaneously support and execute two or more computing environments, in which “environment” includes operating systems plus user applications and data. A VM is a virtual computing system that borrows computing resources (CPU, disk, memory, etc) from its *host computer*, co-existing alongside the host computer as a *guest computer*. Sharing computer resources can provide benefits to the end user, including:

- **Isolation** – The disparate environments operate without regard for the others. While some resources may be shared (for example, the keyboard, mouse, and screen), others are protected (for example, data files). In a VM environment, host resources need to be protected from concurrent access and control by both the host and the guest. In traditional VM systems, this isolation and control is provided by a special software program called a virtual machine monitor (VMM). The VMM may act as a peer of the host computing environment or may act autonomously from the host and guest computers.
- **Resource Balancing** – Each operating environment functions autonomously and therefore can have its resource consumption monitored and restricted, if desired.
- **Transportability/Mobility** – Certain VM configurations are considered mobile; that is, the entire environment can be moved from one host computer to a different one. Today’s increasingly mobile world can benefit from such virtual machines, or VMs that can be transported from one host to another easily. Mobility is crucial when using VM technology to craft a virtual desktop solution. Although software products exist today that provide basic VM capabilities to current PCs, these products provide limited or no support for mobile VMs, due to their large software footprint. This is a key deficiency in these solutions.

There are many ways to implement a VMM and its supporting software. In this paper we will illustrate a variety of different VMM technologies, including:

- **Hypervisors** – VMMs that intercept and trap low-level CPU instructions, emulating any instructions that would violate isolation or cause system instability. Hypervisors can provide a complete virtualized desktop, with varying degrees of overhead / performance loss.
- **Paravirtualizers** – VMMs that intercept and trap low-level CPU instructions, failing any instructions that would violate isolation or cause system instability. Paravirtualizers can require that the guest desktop operating system be modified to avoid these privileged instructions. Paravirtualized systems can provide a complete virtualized desktop, with varying degrees of overhead / performance loss.

- **Workspace Virtualization Engines (WVE)** – VMMs that intercept and trap low-level OS API calls, emulating or redirecting those that would violate isolation or cause system instability. Some WVEs can provide the capability to virtualize privileged code modules and full operating system subsystems at a kernel level enabling the workspace to join an enterprise domain, have an isolated network stack and support applications such as endpoint security, databases, and PC management software that require drivers and security services. WVEs can provide a complete virtualized desktop, with little or no performance loss.
- **Application Containers** – Systems that intercept and trap the highest level OS API calls, emulating those that would violate isolation or cause system instability. Application containers cannot typically provide a complete virtualized desktop.
- **Emulators** – Systems that emulate entire hardware systems, including CPUs, I/O devices, etc. Emulators can provide a complete virtualized desktop, but at a tremendous performance loss.

Virtual Desktop Components

Virtual Machine Monitor

One of the difficulties of illustrating a virtual machine is that there is no single architecture that shows how a “typical” VM works. Generally speaking, though, most VM architectures use some arrangement of the following components:

- A host computer hardware platform (CPU, memory, disk, etc)
- A host computer operating system
- A guest computer operating system
- Host computer data and applications
- Guest computer data and applications
- Guest preferences and settings (personalization)
- A VMM program

Regardless of what type is in use, the function of the VMM program is to isolate host resources from each guest in the virtual machine environment. The VMM program intercepts requests for resources from each guest computer and forwards these requests to the host computer for processing. In this way, each guest computer sees a complete set of resources available to it. Although the guest believes it has full control over these resources, the elements that it actually controls are virtual, in the sense that they do not correspond directly to physical or logical host resources. Using this approach, a number of guest computers can be virtualized using a single set of resources (those available on the host computer), combined with a suitable VMM program.

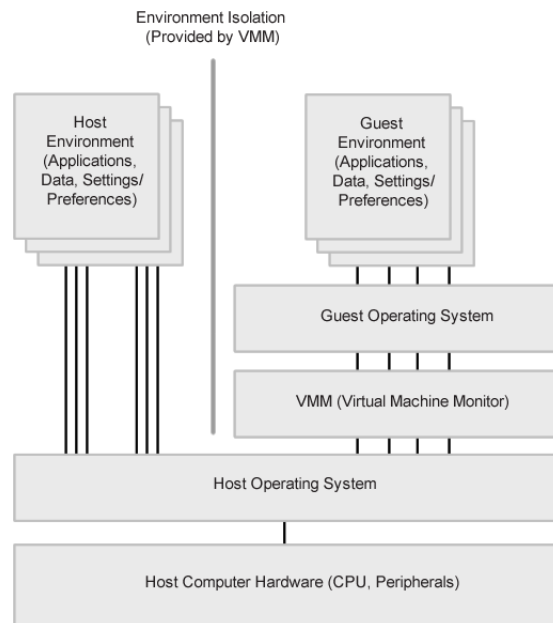


Figure 1 - Typical Virtual Machine Architecture

Figure 1 illustrates the most typical way a VM-based system is organized.

Host OS

All reasonable virtual desktop solutions assume the existence of a host operating system. Although there are a handful of solutions that require no host OS, referred to as “bare metal” VMM solutions, it is unrealistic to expect these to be installed on every single host computer that an enterprise may use. This is especially true when considering a VMM solution for work-at-home employees or “day-extender” employees who may be working on unmanaged host PCs.

Most modern operating systems (32 or 64 bit, protected mode) serve as suitable host operating systems.

Guest OS

The guest operating system (or guest view of the OS, in the case of some solutions), provides the underlying operating system that the virtual desktop will use to execute its applications. In hypervisor-based, paravirtualized, and emulator configurations, the guest operating system may be a distinctly different one than the host operating system. In workspace virtualization, the operating system (described shortly), the choice can be made to either share the host operating system (and thus present the host's OS to the virtual desktop environment) or to present an alternate guest operating system. In application containers, the guest view of the OS is actually a view into the host operating system, and therefore reflects the same things.

Guest Applications

A virtual desktop environment will most likely contain one or more virtualized applications, which are executed using the guest operating system, described above. In the ideal virtual desktop environment, the applications installed are isolated from applications installed in the host computer. In some virtual desktop environments, applications can also be shared or launched from the host computer. This configuration, possible when using workspace virtualization or application containers, allows for greater flexibility, but carries with it a lower level of isolation. Nevertheless, this configuration may be desirable to some enterprises. Sharing or launching applications from the host computer is not easily accomplished using hypervisor-based, paravirtualized, or emulated virtual desktops.

Guest Data

A user's data is almost certainly part of their virtual desktop, and thus needs to be isolated and managed just like any other aspect of the system.

Guest Preferences/Settings

Not unlike data, preferences and settings are part of a desktop, and thus part of a virtual desktop as well. Certain preferences and settings are personal – these might include desktop background wallpaper selections, display font sizes, system sounds, plug-ins and the like. Other preferences and settings should be managed by a system administrator – these settings typically comprise system-level settings such as password strength requirements and group policy enforcement (for Windows desktops). It is imperative for any enterprise virtual desktop solution to be able to handle management and isolation of such settings properly.

VMM Program

A Virtual Machine Monitor provides the core virtualization engine responsible for executing the virtual desktop environment.

Overview of Virtual Machine Deployments

As discussed earlier, there are many ways to construct a VM environment. Each approach has strengths and weaknesses. In this section, we will examine several popular VM technologies and illustrate the architecture of each.

Regardless of which configuration is used, the chief role of a VMM program is to isolate host computer resources from each guest, and provide each guest computer with the perception that it has a full set of hardware resources available to it. These resources may be traditionally found in hardware (such as CPUs, memory, disks, printers, input devices, CRTs, etc), or may be seen in software (such as shared library files, configuration files, shared data, etc). It is the job of the VMM to ensure that each participant sees what it perceives to be a private set of these resources, and that changes to them do not “leak” through to another participant's view. There are several approaches that can be used to solve this problem. These approaches differ at the level at which they virtualize.

Hypervisors

In the architectural stack, hypervisor VMMs sit as close to the real hardware as possible, managing the hardware resources of the host computer on behalf of the guest operating systems as well as the desktops that run on these guest OSes. Hypervisors will intercept hardware access requests from guest operating systems and redirect these requests either to the real hardware, if permitted by the hypervisor's configuration policy or rules, or to emulated hardware. Emulated hardware is often required in hypervisor scenarios, as not all host-resident hardware is natively shareable (for example, disk controllers, video controllers, etc).

Types of Hypervisors

Hypervisors have evolved over time to assume an increasing role in host hardware/resource management. In legacy configurations, the hypervisor used parts of a host operating system for resource management. This eliminated the need for the hypervisor vendor to implement complex hardware resource management software, which is typically better left to operating system vendors. This configuration necessitates

the presence of a host operating system. In this configuration, the hypervisor comprises a loadable program that uses the native hardware abstraction layer (HAL) of a host operating system.

Other more recent hypervisor-based solutions do not require a host operating system. These solutions are known as “bare metal” hypervisors, meaning that the hypervisor VMM is responsible for assuming all resource management normally assumed by the host operating system. This approach is usually employed in server virtualization environments, where the need for a host operating system, with its associated overhead and management concerns, is unwarranted. Bare metal hypervisors typically have a limited set of validated/approved hardware configurations, since driver software supplied by hardware vendors may not exist.¹ Implementing a bare metal hypervisor may be made easier by using an off-the-shelf operating system,² removing everything except the resource management code and driver support, and then adding in the VMM. This yields a very thin VMM but somewhat blurs the line between a bare metal and non-bare metal configuration, since technically there actually *is* a host operating system albeit a very minimal one

Traps & Instruction Emulation

The interception technology employed by modern x86 hypervisors is based on *trapping* invalid or privileged instructions. Such instructions are defined as those that, if left to execute as-is, could compromise the integrity of the host, VMM, or other guest environments, and *emulating* or *redirecting* these instructions so that they do not cause any side-effects.

Most privileged instructions are automatically trapped by the CPU, and thus it is a trivial task for a hypervisor VMM to register itself to be the *trap handler* – the entity responsible for handling these privileged instructions. In this situation, when the CPU detects a privileged instruction, such as a guest operating system attempting to control an I/O device, the following steps will occur:

1. The CPU traps the privileged instruction
2. The guest operating system is temporarily suspended

¹ Hardware support for a truly generic bare-metal hypervisor system would entail a similar hardware device driver development effort as supporting a commodity OS.

² Linux is a popular choice

3. The privileged request is sent to the hypervisor for handling
4. The hypervisor satisfies the request – by one of several possible means:
 - a. Emulating the instruction in a fashion that does not generate side-effects
 - b. Redirecting the I/O (in this example) to a virtual (emulated) hardware device
 - c. Permitting the original request (for situations where the I/O device in this example may be dedicated to that guest operating system exclusively, since in such a configuration, there is no chance for corruption or side-effects)
5. The hypervisor resumes the guest operating system at the point where the interruption occurred
6. The guest operating system continues execution as if nothing special occurred

Unfortunately, the x86 architecture was not initially designed with virtualization in mind. The entire sequence of steps described above hinges on step 1, “the CPU traps the privileged instruction” actually occurring. There are instructions that *are* privileged, but *do not* trap, and thus will not go through the sequence of steps described above. If these instructions are not detected and intercepted properly, it is possible for a guest operating system to execute privileged code that could conceivably compromise the entire system.

It is in these scenarios that the hypervisor VMM needs to work much harder. At a high level, a hypervisor needs to “scan ahead” in the instruction stream of an executing guest operating system in order to anticipate non-trappable instructions that will be executed in the near future. When these instructions are detected, they are replaced in-memory with instructions that will cause a trap or a call to the VMM. This aggressive and repeated scan-and-patch process imparts a noticeable performance overhead to this type of VMM. Prescanning and patching of pages in memory causes other headaches for hypervisor VMM implementations:

- **Duality of views of a memory page** – If the VMM alters an instruction from a non-trappable privileged instruction to something else (that causes a trap, for instance), actual memory in the guest operating system is being altered. This memory is typically code pages corresponding to or mapped to some executable file. If the guest operating system performs a self-integrity check by checksumming its memory pages, the VMM needs to somehow revert the view of the memory page back to what it was before patching, or else the guest OS may incorrectly assume that it had been tampered with. Thus, the VMM needs to understand and handle the difference between executing a memory page and reading a memory page.

- **When to scan and patch?** – In modern commodity operating systems, memory pages may be stored in a variety of locations. They may be physically mapped to a real memory address, they may be paged out to a swap file, or backed/mapped to a file in nonvolatile storage. The VMM may choose any of these locations to patch, but which one is correct? Many modern hypervisors perform the scan and patch operation at fault-in time, which is the time when a page is mapped to physical memory. Page faults occur *very* frequently - possibly thousands of times per second. Increasing the time for processing a page fault (by scanning and patching, for example), has a direct impact on overall system performance.
- **Self-modifying code** – In this scenario, a page of memory is scanned and no non-trappable privileged instructions are detected. However, code in the memory page is written in such a way that it modifies itself and places a non-trappable privileged instruction directly in the code path. Although this is admittedly poor coding practice, a robust hypervisor VMM needs to handle this possibility as well.³

Hardware Virtualization Support in Modern CPUs

To avoid the issues surrounding the non-trappable privileged instructions, CPU manufacturers have added virtualization capabilities to modern CPUs. These CPUs still need to behave exactly like their older counterparts, to ensure backward-compatibility with existing applications, so the behavior of the non-trappable privileged instructions is identical on these newer units. However, a new set of instructions, which were not previously available, provides for a special virtual machine mode of operation. This mode provides extra levels of I/O and memory protection, and allows the special instructions to be detected and handled externally by the CPU in conjunction with a suitably written hypervisor VMM.

The first revision of this instruction set provided basic support for virtualizing the non-trappable privileged instructions, and the supporting instructions required to interface the hardware support to VMMs written to make use of it.⁴

³ One way to implement this detection feature would be to mark all code pages read only, resulting in a trap when the modification is performed. The VMM, registered as the trap handler, could then detect the context and either write the desired instruction(s) itself if said write did not comprise a non-trappable privileged instruction, or instead write a different instruction that would trap, if said write comprised a non-trappable privileged instruction. There are probably more efficient ways to accomplish this, but the example illustrates the coding contortions that a hypervisor VMM needs to be able to handle.

⁴ First revisions of hardware support for virtualization in CPUs did not actually reveal a significant performance improvement over existing hypervisor VMMs, primarily due to the fact that by that time existing hypervisors had become quite fast in trapping/emulating the non-trappable privileged instructions.

CPUs with first generation hardware virtualization support are not capable of nesting VMMs in a way that allows the nested VMM to also take advantage of the hardware virtualization capabilities. This means that when using CPUs with first generation hardware virtualization, only the “outermost” VMM can benefit from the hardware acceleration. While this may not initially seem to be a problem, some modes of operation are not possible without hardware virtualization support, meaning those modes will not be available in a nested-VMM configuration.⁵

Furthermore, some computer manufacturers intentionally disable hardware virtualization support in their PCs, even if the hardware is capable of supporting it.⁶

The second revision of this hardware support, present in only the newest CPUs at the time of this writing, *does* allow virtualization of the hardware virtualization instructions, allowing for some degree of nested VMM capabilities.

Limitations and Challenges

- **Performance** – There will always be some level of performance overhead associated with hypervisor VMMs. As CPUs evolve to include more hardware support for virtualization, the areas of performance impact will shift from the VMM to the emulated hardware instead, which can still result in a non-trivial impact.
- **VM inside another VM** – As mentioned before, unless one uses extremely new hardware it is not possible to run a hardware-accelerated VM inside another hardware-accelerated VM.
- **No hardware assist** – Especially important in a virtual desktop solution, one must consider the millions of PCs in the field that could be used as host PCs that do *not* have even the first generation of virtualization support.⁷

⁵ For example, executing 64 bit guest environments on a 32 bit host environment on certain Intel® CPUs.

⁶ Most PC manufacturers that disable virtualization support have not publicly explained the reasons for this decision, but frequently, the choice to disable hardware virtualization is made for the “low end” PC models and enabled for “high end” PC models, possibly in an attempt to differentiate otherwise similar hardware configurations.

⁷ Even newly-shipping CPUs may not include hardware virtualization support for cost reasons. It is less expensive (and a feature differentiator for the CPU manufacturer) to omit or disable this support in low-cost CPUs. This is also a significant issue for laptop users as CPU manufacturers may omit hardware virtualization support in the CPU in order to save energy and thermal output.

Paravirtualizers

Another popular virtualization technique is paravirtualization. Paravirtualization is illustrated in Figure 2 and consists of a special VMM program called a hypervisor that manages a specially-constructed host computer and guest computers. The special construction that is typically required consists of modifications to the operating system core components.

These modifications may be trivial or non-trivial, depending on the operating systems and computer architectures involved. However, some modifications will need to be made, and this fact prevents the use of common off-the-shelf operating systems in paravirtualized environments. It should be noted that a “legacy” paravirtualizer such as described will run unmodified on CPUs that do not include support for hardware virtualization, typically at higher performance levels than hypervisor configurations running on the same hardware. When using hardware-accelerated virtualization, enterprises should judge carefully which approach yields the optimal performance.

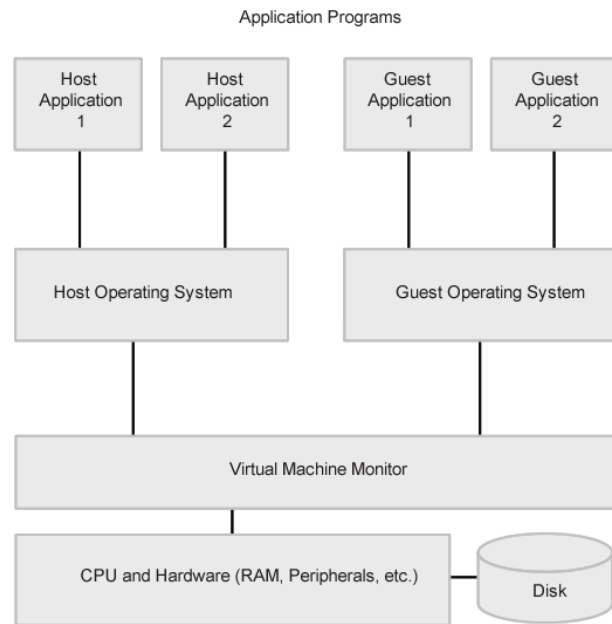


Figure 2 – Paravirtualized VM Architecture

Guest OS Modification

Hypercalls

In order to avoid the interception problem that face hypervisors, paravirtualized guest operating systems need to be modified to remove all privileged instructions, whether trappable or not. Privileged instructions are replaced with a *hypercall*, which is a call into the paravirtualized VMM. The VMM then handles the request on behalf of the guest operating system and returns.

The areas in the guest operating system that need to be modified comprise areas that contain privileged instructions. These are exclusively kernel modifications – user applications will not require such modifications. Areas in a typical kernel that require modification include:

- **Memory Management** – Since the guest operating system no longer controls physical memory exclusively, all memory management code must be replaced by calls into the VMM to perform memory management on behalf of the guest OS.
- **Hardware Interface** – Similarly, hardware is not controlled directly by the guest Hardware Abstraction Layer (HAL) and must be redirected to the VMM for processing.
- **Scheduling** – CPUs are virtual in paravirtualized systems, meaning that they are simulated or emulated by the VMM. Scheduling code, usually driven by a hardware timer interrupt directed to the operating system, is contained in the VM. On receipt of a timer interrupt, the VMM dispatches a virtual interrupt to the guest operating system, which then uses its own scheduling code internally to schedule processes.

Complexity of Guest OS Modifications

The effort required to modify a guest OS to reflect the changes required above depends on the type of guest operating system, and the amount of privileged code. There may be few modifications required or significant change may be needed. Further, for a given guest operating system, the amount of modifications required in the above areas may not be equal for a given operating system (e.g. a given OS may require more modification in its memory management subsystem and comparatively few in its HAL subsystem).

Most popular open-source operating systems have already been modified to work with popular paravirtualized VMMs. Support for non-open source operating systems is not as extensive.

Limitations

It should be obvious that the need to change the guest operating system is a major limitation of this approach if the guest operating system chosen is not easily modifiable.⁸ Generally speaking, most popular paravirtualizers support a mode of operation that mimics a traditional hypervisor; this mode of operation may require hardware-assistance, as discussed in the hypervisor section of this paper. In this mode, there is no need to modify the guest operating system.

⁸ This could be the case if technical limitations or vendor intransigence preclude it.

Application Containers

Application containers take a different approach to solving the problem of desktop virtualization. Rather than virtualizing an entire desktop, they instead virtualize individual applications. This approach provides a few benefits over paravirtualizers and hypervisors:

- **No need to have a separate guest operating system** – This reduces footprint of the virtual desktop and can also help to reduce software licensing costs.
- **Higher levels of performance** – Since there is no translation of privileged code, the performance concerns of hypervisors are largely unwarranted when using an application container.
- **Lower privilege levels required on host operating system** – Since the only thing being virtualized are user applications, there is no privileged code is being executed. This approach can therefore run with lowered privileges, if configured properly.

Architecture of a Typical Application Container

Not unlike other technologies described in this paper, application container VMs perform *interception* and *redirection*. As shown in Figure 3, application container VMs intercept at the highest level of all the technologies, typically entirely in user space.

In an application container, calls to the underlying OS are intercepted at the subsystem level and redirected to the application container vendor's proprietary reimplementation of the subsystem, contained in the application container VM. The VM then executes the request after translating it in a container-specific way. This translation typically involves rewriting the request so as to not cause interference with host-resident applications or applications running in other containers.

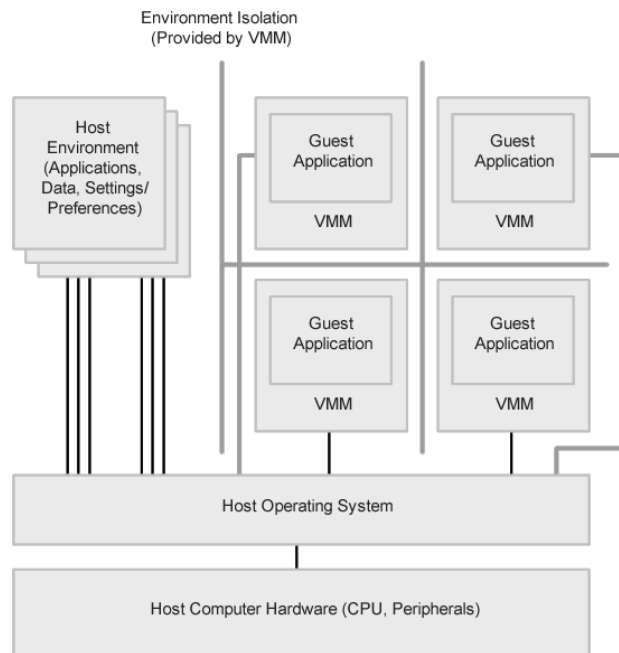


Figure 3 – Application Container

In many popular application container implementations, applications are pre-packaged, typically through a tool or program that examines their behavior. On discovery of potentially troublesome API calls, the application is rewritten automatically to transfer control to the VM instead of the host operating system.

Application containers can instantiate their VMs on a per-container basis, or on a per-system basis. In the former approach, each container receives its own copy of the VM code. In the latter approach, the VM is instantiated as a process shared between all containers. The latter approach provides an architecture that is more conducive to inter-container communication, which is described shortly.

Isolation Concerns

Since an ideal application container runs in unprivileged space on a potentially untrusted host operating system, there is no capability for contained applications to protect themselves from observation/modification. This means that application containers typically have the lowest level of native host isolation, or isolation of the virtual desktop from potential threats in the host computer, of all the approaches described in this paper.

Inter-container Communication

As shown above, applications running in containers are by definition isolated from applications running in other containers. However, applications that are part of *application suites* frequently need to communicate between themselves. This leads to a packaging or management concern for enterprises that are deploying application containers that need to use application suites. One approach would be to package each application in the suite in separate containers, foregoing communication between these applications and thus reducing the applications' capability for users. Another approach would be to sacrifice isolation and package the entire suite as a single container. This leads to a larger container with less isolation between applications, but more capability for users.

These isolation issues have recently led application container vendors to add support for dynamic container isolation – allowing administrators to define, at runtime, which containers are isolated and which are not.

Limitations

Application containers have their own set of limitations:

- **Applications that install privileged components** – Since an ideal application container does not require privilege, it is difficult if not impossible for such a configuration to support the installation of applications that require privileged components.⁹
- **Preparation of the container itself** – The need to prepare or package the application entails an additional installation step not required by any of the other desktop virtualization techniques
- **3rd Party implementation of core system functions** – Most application containers need to reimplement lower-level operating system functions. If not implemented exactly to specification, including all bugs and behaviors present in the native functionality, applications inside the container may fail or operate improperly.

Emulators

Consider the VM shown in Figure 4. This VM provides a completely virtualized set of hardware resources, emulating CPUs and other hardware resources as needed.

Emulation as a virtualization technology has some strengths and drawbacks:

- Emulation is a completely portable solution across architectures – Since the virtual host hardware is completely isolated, the emulation software can be reimplemented on any host.
- It is possible to emulate hardware resources not present on a particular host – Since the virtual host hardware is completely emulated, a software solution can be designed to provide hardware features lacking on a given host. This is especially important when considering challenges posed by mobile VMs.

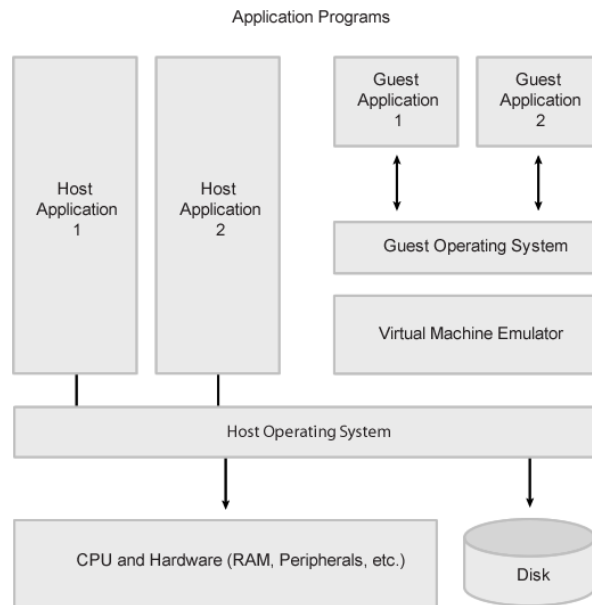


Figure 4 – Emulator Architecture

⁹ Such applications include antivirus/security products, VPN clients, and multimedia software. On Windows® systems, this class of applications would also include any software that installed a service managed by Windows service control manager.

- Performance is generally magnitudes slower than traditional VMs or paravirtualized VMs - Since hardware is completely emulated in software, large amounts of host CPU processing power are needed to provide a modest amount of guest processing power.
- Host hardware emulation (or acceleration) is typically required for this type of VM.

Host Hardware Emulation

Host hardware emulation is the process of simulating a set of host hardware for each guest machine. Emulation in this fashion yields the ultimate separation and isolation of host and guest, since nothing is shared between them whatsoever. Referring again to Figure 4, the VMM program is responsible for not only isolating hardware resources, but actually simulating a “fake” set of hardware resources that the guest(s) will use during execution.

Workspace Virtualization

Sitting in the architectural middle-ground between application virtualization and hypervisor-based virtualization (Virtual Machines) is workspace virtualization. Workspace virtualization is a virtualization approach that encapsulates and isolates an entire computing workspace. At a minimum, the

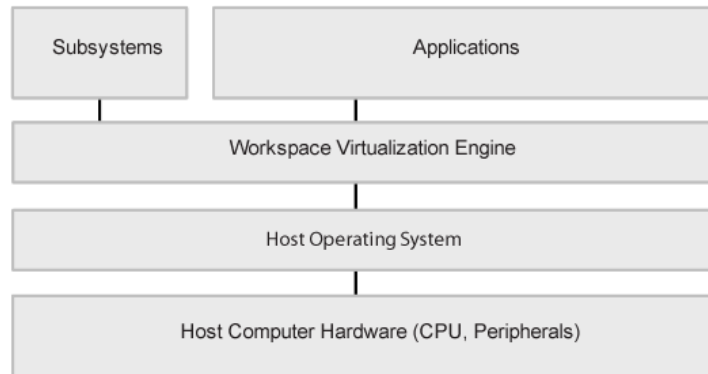


Figure 5 - Minimalist Workspace Virtualization Architecture

workspace would comprise everything above the operating system kernel – applications, data, settings, and any non-privileged operating system subsystems required to provide a functional Windows desktop computing environment (see Figure 5).

Such an ideal scenario is almost never achieved, due to architectural concerns imparted by modern operating systems and application development practices:

- Applications frequently contain privileged code (drivers or services), which interacts with the operating system at a privileged level. These privileged code modules need to be properly isolated and virtualized, in privileged/kernel mode.

- Operating system subsystems frequently are separated into privileged and non-privileged modules¹⁰. Both modules need to be virtualized in order to properly support applications using the subsystem.

Due to these concerns, a proper WVE needs to provide the capability to virtualize privileged code. A full WVE that virtualizes privileged code is shown

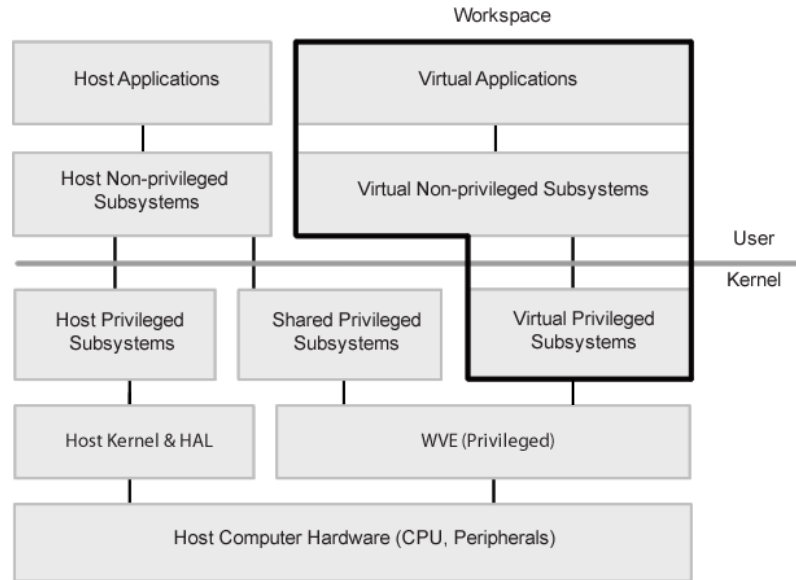


Figure 6 - Preferred Workspace Virtualization Architecture

in Figure 6, can provide the capability to virtualize privileged code modules and full operating system subsystems through a kernel-mode WVE. Full WVEs provide a foundation for executing a full workspace that can join an enterprise domain, has an isolated network stack and supports applications such as endpoint security, databases, strong authentication and PC management software that require drivers and security services.

An ideal WVE enables:

- A duplicate set of privileged virtual subsystems (I/O, Networking, Security, etc)
- A duplicate set of non-privileged virtual subsystems (Software installation, remote management, etc)
- Support for isolated workspace application (end-user application software)
- Sharing of redundant subsystems with the host PC where architecturally relevant (for high performance I/O to shared hardware devices, for example)

By enabling the above capabilities, the WVE-based workspace virtualization solution can offer a high level of application compatibility while simultaneously providing a high level of performance to virtual applications by allowing some selected subsystems to be shared with the host PC.¹¹

¹⁰ This can be due to architectural requirements of the operating system, or by a desire by the operating system vendor to embrace the software design best practice of “privilege separation” – splitting a subsystem into multiple parts in order to achieve a higher level of security.

¹¹ For example, graphics and disk I/O can be passed directly to the host OS subsystem for these hardware resources, instead of depending on hardware emulation as hypervisor-based solutions require.

Summary

When considering a virtual desktop deployment, a variety of architectural approaches may be chosen. Each approach has its strengths and drawbacks, and enterprises are cautioned to carefully research each one before finalizing their decision. Where performance is of paramount concern, workspace virtualization and application containers offer the greatest benefit with varying degrees of application interoperability. Where cross-platform support is a must-have, hypervisor or paravirtualized systems provide this capability, with their associated performance and footprint overhead.

A blended approach using several of the implementations described in this document may prove to be the best overall solution – in this scenario, the cross platform support and high levels of isolation provided by a hypervisor is combined with a lightweight, high performance workspace virtualization engine, yielding a solution that can be tightly managed and isolated, but still provide the flexibility required for mobility and offline use.

About RingCube

RingCube is the leading provider of workspace virtualization. The company's innovative virtualization solution, vDesk, enables users to securely access their complete desktop computing experience from any Windows PC anywhere in the world. With vDesk, organizations can increase user productivity, lower desktop management and support costs, and eliminate the performance and resource overhead commonly found with legacy virtualization approaches.



RingCube Technologies, Inc.
100 W. Evelyn Ave., Suite 210
Mountain View, CA 94041
United States
Main: 1-866-323-4278
International: 650-605-6900
Fax: 408-605-6901