# Live Migration with Pass-through Device for Linux VM

Edwin Zhai, Gregory D. Cummings, and Yaozu Dong
*Intel Corp.*
{edwin.zhai, gregory.d.cummings, eddie.dong}@intel.com

## Abstract

Open source Linux virtualization, such as Xen and KVM, has made great progress recently, and has been a hot topic in Linux world for years. With virtualization support, the hypervisor de-privileges operating systems as guest operating systems and shares physical resources among guests, such as memory and the network device.

For device virtualization, some mechanisms are introduced for improving performance. Paravirtualized (PV) drivers are implemented to avoid excessive guest and hypervisor switching and thus achieve better performance, for example Xen's split virtual network interface driver (VNIF). Unlike software optimization in PV driver, IOMMU, such as Intel® Virtualization Technology for Directed I/O, AKA VT-d, enables direct passing through of physical devices to guests to take advantage of hardware DMA remapping, thus reducing hypervisor intervention and achieving high bandwidth.

Physically assigned devices impose challenges to live migration, which is one of the most important virtualization features in server consolidation. This paper shows how we solve this issue using virtual hot plug technology, in addition with the Linux bonding driver, and is organized as follows: We start from device virtualization and live migration challenges, followed by the design and implementation of the virtual hotplug based solution. The network connectivity issue is also addressed using the bonding driver for live migration with a direct assigned NIC device. Finally, we present the current status, future work, and other alternative solutions.

## 1 Introduction to Virtualization

Virtualization became a hot topic in Linux world recently, as various open source virtualization solutions based on Linux were released. With virtualization, the hypervisor supports simultaneously running multiple operating systems on one physical machine by presenting a virtual platform to each guest operating system. There are two different approaches a hypervisor can take to present the virtual platform: full virtualization and paravirtualization. With full virtualization, the guest platform presented consists of all existing components, such as a PIIX chipset, an IDE controller/disk, a SCSI controller/disk, and even an old Pentium® II processor, etc. which can be already supported by modern OS without any modification. Paravirtualization presents the guest OS with a synthetic platform, with components that may not have existed in the real world to date, and thus are unable to run a commercial OS directly. Instead, paravirtualization requires modifications to the guest OS or driver source code to match the synthetic platform, which is usually designed to avoid excessive context switches between guest and hypervisor, by using the underlying hypervisor knowledge, and thus achieving better performance.

## 2 Device Virtualization

Most hardware today doesn't support virtualization, so device virtualization could only rely on pure software technology. Software based virtualization shares physical resources between different guests, by intercepting guest access to device resource, for example trapping I/O commands from a native device driver running in the guest and providing emulation, that is an emulated device, or servicing hypercalls from the guest front-end paravirtualized drivers in split device model, i.e. a PV device. Both sharing solutions require hypervisor intervention which cause additional overhead, which limits performance.

To reduce this overhead, a pass-through mechanism is introduced in Xen and KVM (work in progress) to allow assignment of a physical PCI device to a specific guest so that the guest can directly access the physical resource without hypervisor intervention [8]. A pass-through mechanism introduces an additional requirement for the DMA engines. A DMA engine transaction requires the host physical address but a guest can only provide the guest physical address. So a method must be invoked to convert a guest physical address to a host physical address for correctness in a non-identical mapping guest and for secure isolation among guests. Hardware IOMMU technologies, such as Intel® Virtualization Technology for devices, i.e. VT-d [7], are designed to convert guest physical addresses to host physical addresses. They do so by remapping DMA addresses provided by the guest to host physical addresses in hardware via a VT-d table indexed by a device requestor ID, i.e. Bus/Device/Function as defined in the PCI specification. Pass-through devices have close to native throughput while maintaining low CPU usage.

PCI SIG I/O Virtualization based Single Root I/O Virtualization, i.e. SR-IOV, is another emerging hardware virtualization technology which specifies how a single device can be shared between multiple guest via a hardware mechanism. A single SR-IOV device can have multiple virtual functions (VF). Each VF has its own requestor ID and resources which allows the VF to be assigned to a specific guest. The guest can then directly access the physical resource without hypervisor intervention and the VF specific requestor ID allows the hardware IOMMU to convert guest physical addresses to host physical addresses.

Of all the devices that are virtualized, network devices are one of the most critical in data centers. With traditional LAN solutions and storage solutions such as iSCSI and FCoE converging on to the network, network device virtualization is becoming increasingly important. In this paper, we choose network devices as a case study.

## 3 Live Migration

Relocating a virtual machine from one physical host to another with very small down-time of service, such as 100 ms [6], is one major benefit of virtualization. Data centers can use the VM relocation feature, i.e. live migration, to dynamically balance load on different hosting platforms, to achieve better throughput. It can also be used to consolidate services to reduce the number of hosting platforms dynamically to achieve better power savings, or be used to maintain the physical platform after running for a long time because each physical platform has its life cycles, while VMs can run far longer than the life cycle of a physical machine. Live migration, or its similar features, like VM save and VM restore, is achieved by copying VM state from one place to another including memory, virtual devices, and processor states. The virtual platform, where the migrated VM is running, must be the same as the one where it previously ran, and it must provide the capability that all internal states can be saved and restored, which depends on how the devices are virtualized.

The guest memory subsystem, making up the guest platform, is kept identical when the VM relocates, assigning the same amount of memory in the target VM with the same layout. The live migration manager will copy contents from the source VM to the target, using an incremental approach, to reduce the service outage time [5], given that the memory a guest owns may vary from tens of megabytes to tens of gigabytes, and even more in the future, which means a relatively long time to transmit even in a ten gigabit Ethernet environment.

The processor type the guest owns and features the host processor have usually need to be the same across VM migration, but certain exceptions can be taken if all the features the source VM uses exist in the target hosting processor, or if the hypervisor could provide emulation of those features which do not exist on the target side. For example, live migration can request the same CPUID in host side, or just hide the difference in host side by providing the guest a common subset of physical features. MSRs are more complicated, except that the host platform is identical. Fortunately, today's guest platform presented is pretty simple and won't use those model-specific MSRs. The whole CPU context size saved at the final step of live migration is usually in the magnitude of tens of kilobytes, which means just several milliseconds of out of service time.

On the device side, cloning source device instances to the target VM after live migration is much more complicated. If the source VM includes only those software emulated devices or paravirtualized devices, identical platform device could be maintained by generating exactly the same configuration for the target VM startup, and the device state could be easily maintained since the hypervisor knows all of its internal state. Those devices are called migration friendly devices. But for guests who have pass-through devices or SR-IOV Virtual Functions on the source VM side, things are totally different.

### 3.1 Issues of Live Migration with pass-through device

Although guest with pass-through device can achieve almost native performance, maintaining identical platform device after migration may be impossible. The target VM may not have the same hardware. Furthermore, even if the target guest has the identical platform device as the source side, cloning the device instance to target VM is also almost impossible, because some device internal states may not be readable, and some may be still in-flight at migration time, which is unknown to the hypervisor without the device-specific knowledge. Even without those unknown states, knowing how to write those internal states to the relocated VM is another big problem without device-specific knowledge in the hypervisor. Finally, some devices may have unique information that can't be migrated, such as a MAC address. Those devices are migration unfriendly.

To address pass-through device migration, either the hypervisor needs to have the device knowledge to help migration or the guest needs to do those device-specific operations. In this paper, we ask for guest support by proposing a guest hot plug based solution to request cooperation from the guest to unplug all the migration unfriendly devices before relocation happens, so that we can have identical platform devices and identical device states after migration. But hot unplugging an Ethernet card may lead to network service outage, usually in the magnitude of several seconds. The Linux bonding driver, originally developed for aggregating multiple network interfaces, is used here to maintain connectivity.

## 4 Solution

This section describes a simple and generic solution to resolve the issue of live migration with pass-through device. This section also illustrates how to address the following key issues: save/restore device state and keeping network connectivity for NIC device.

### 4.1 Stop Pass-through Device

As described in the previous section, unlike emulated devices, most physical devices can't be paused to save and restore their hardware states, so a consistent device state across live migration is impossible. The only choice is to stop the guest from using physical devices before live migration.

How to do it? One easy way is to let the end user stop everything using a pass-through device including applications, services, and drivers, and then restore them on the target machine after the hypervisor allocates a new device. This method works, but it's not generic, as different Linux distributions have different operations. Moreover, a lot of user intervention is needed inside the Linux guest.

Another generic solution is ACPI [1] S3 (suspend-to-ram), in which the operating system freezes all processes, suspends all I/O devices, then goes into a sleep state with all context lost except system memory. But this is overkill because the whole platform is affected, besides the target device, and service outage time is intolerable. PCI hotplug is perfect in this case, because:

- Unlike ACPI S3, it is a device-level, fine-grained mechanism.
- It's generic, because the 2.6 kernel supports various PCI hotplug mechanisms.
- No huge user intervention, because PCI hotplug can be triggered by hardware.

The solution using PCI hotplug looks like the following:

1. Before live migration, on the source host, the control panel triggers a virtual PCI hot removal event against the pass-through device into the guest.
2. The Linux guest responds to the hot removal event, and stops using the pass-through device after unloading the driver.
3. Without any pass-through device, Linux can be safely live migrated to the target platform.
4. After live migration, on the target host, a virtual PCI hot add event, against a new pass-through device, is triggered.
5. Linux guest loads the proper driver and starts using the new pass-through device. Because the guest reinitializes a new device that has nothing to do with the old one, the limitation described in 3.1 doesn't hold.

### 4.2 Keeping Network Connectivity

The most popular usage model for a pass-through device is assigning a NIC to a VM for high network throughput. Unfortunately, using PCI NIC hotplug within live migration breaks the network connectivity, which leads to an unpleasant user experience. To address this issue, it is desired that the Linux guest can automatically switch to a virtual NIC after hot removal of the physical NIC, and then migrate with the virtual NIC. Thanks to the powerful and versatile Linux network stack, the Ethernet bonding driver [3] already supports this feature.

The Linux bonding driver provides a mechanism for enslaving multiple network interfaces into a single, logical "bonded" interface with the same MAC address. Behavior of the bonded interfaces depends on modes. For instance, the bonding driver has the ability to detect link failure and reroute network traffic around a failed link in a manner transparent to the application, which is active-backup mode. It also has the ability to aggregate network traffic in all working links to achieve higher throughput, which is referred to as trunking [4].

The active-backup mode can be used for an automatic switch. In this mode, only one slave in the bond is active, while another acts as a backup. The backup slave becomes active if, and only if, the active slave fails. Additionally, one slave can be defined as primary that will always be the active while it is available. Only when the primary is off-line will secondary devices be used. This is very useful when bonding pass-through device, as the physical NIC is preferred over other virtual devices, for performance reasons.

It's very simple to enable bonding driver in Linux. The end user just needs to reconfigure the network before using a pass-through device. The whole configuration in the Linux guest is shown in Figure 1, where a new bond is created to aggregate two slaves: the physical NIC as primary, and a virtual NIC as secondary. In normal conditions, the bond would rely on the physical NIC, and take the following actions in response to hotplug events in live migration:

- When hot removal happens, the virtual NIC becomes active and takes over the in/out traffic, without breaking the network inside of the Linux guest.
- With this virtual NIC, the Linux guest is migrated to target machine.
- When hot add is complete on the target machine, the new physical NIC recovers as the active slave with high throughput.

In this process, no user intervention is required to switch because the powerful bonding driver handles everything well.

### 4.3 PCI Hotplug Implementation

PCI hotplug plays an important role in live migration with a pass-through device. It should be implemented in the device model, according to the hardware PCI hotplug spec. Currently, the device model of most popular Linux virtualization solutions such as Xen and KVM, are derived from QEMU. Unfortunately, QEMU did not support virtual PCI hotplug when this solution was developed, so we implemented a virtual PCI hotplug device model from scratch.

#### 4.3.1 Choosing Hotplug Spec

The PCI spec doesn't define a standard hotplug mechanism. Here are the three existing categories of PCI hotplug mechanisms:

- **ACPI Hotplug**: This is a similar mechanism as the ACPI dock hot insert/ejection, where some ACPI control methods work with ACPI GPE to service the hotplug.
- **SHPC [2] (Standard HotPlug Controller)**: It's the spec from PCI-SIG to define a complicated controller to handle the PCI hotplug.
- **Vendor-specific**: There are other vendor-specific standards, such as Compaq and IBM, which have their own hardware on servers for PCI hotplug.

Linux 2.6 supports all of the above hotplug standards, which gives us more choices to select a simple, open, and efficient one. SHPC is a really complicated device, so it's hard to implement. Vendor-specific controllers are not well supported in other OS. ACPI hotplug is best suited to being emulated in the device model, because interface exposed to OSPM is very simple and well defined.

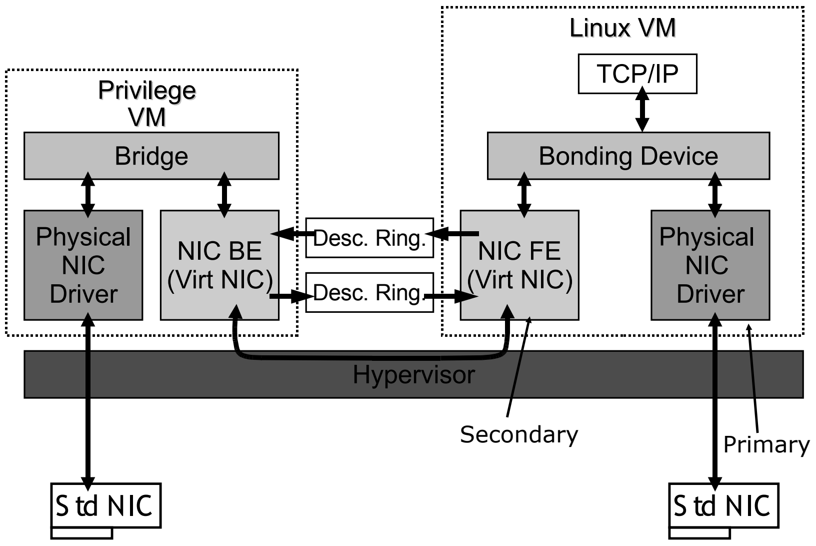

Figure 1: Live Migration with Pass-through Device

#### 4.3.2 Virtual ACPI hotplug

Making an ACPI hotplug controller in device model is something like designing a hardware platform to support ACPI hotplug, but using software emulation. Virtual ACPI hotplug needs several parts in the device model to coordinate in a sequence similar to native. For system event notification, ACPI introduces GPE (General Purpose Event), which is a bitmap, and each bit can be wired to different value-added event hardware depending on design.

The virtual ACPI hotplug sequence is described in Figure 2. When the end user issues the hot removal command for the pass-through device, analogous to pushing the eject button, the hotplug controller updates its status, then asserts the GPE bit and raises a SCI (System Control Interrupt). Upon receiving a SCI, the ACPI driver in the Linux guest clears the GPE bit, queries the hotplug controller about which specific device it needs to eject, and then notifies the Linux guest. In turn, the Linux guest shuts down the device and unloads the driver. At last, the ACPI driver executes the related control method `_EJ0`, to power off the PCI device, and `_STA` to verify the success of the ejection. Hot add is similar to this process, except it doesn't call the `_EJ0`.

In the process shown above, it's obvious that following components are needed:

- **GPE**: A GPE device model, with one bit wired to the hotplug controller, is described in the guest FADT (Fixed ACPI Description Table).
- **PCI Hotplug Controller**: A PCI hotplug controller is needed to respond to the user's hotplug action and maintain the status of the PCI slots. ACPI abstracts a well-defined interface so we can implement internal logic in a simplified style, such as stealing some reserved ioports for register status.
- **Hotplug Control Method**: ACPI control methods for hotplug, such as `_EJ0` and `_STA`, should be added in the guest ACPI table. These methods interact with the hotplug controller for device ejection and status check.

## 5 Status and Future Work

Right now, hotplug with a pass-through device works well on Xen. With this and the bonding driver, Linux guests can successfully do live migration. Besides live migration, pass-through device hotplug has other useful usage models, such as dynamically switching physical devices between different VMs.

There is some work and investigation that needs to be done in future:

- **High-level Management Tools**: Currently, hotplug of a pass-through device is separated from generic live migration logic for a clean design, so

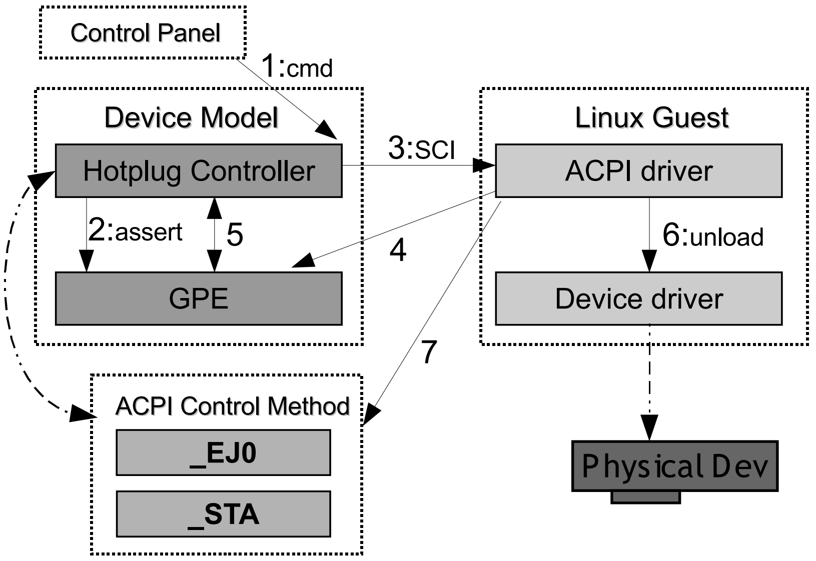

Figure 2: ACPI Hotplug Sequence

the end user is required to issue hotplug commands manually before and after live migration. In the future, these actions should be pushed into high level management tools, such as a friendly GUI or scripts, in order to function without user intervention.

- **Virtual S3**: The Linux bonding driver works perfectly for a NIC, but bonding other directly assigned devices, such as graphics cards, is not as useful. Since Linux has good support for ACPI S3, we can try virtual S3 to suspend all devices before live migration and wakeup them after that. Some drawbacks of virtual S3 need more consideration:

  – All other devices, besides pass-through devices, go into this loop too, which takes more time than virtual hotplug.

  – With S3, the OS is in sleep state, so a long down time of the running service is unavoidable.

  – S3 has the assumption that the OS would wake up on the same platform, so the same type of pass-through devices must exist in the target machine.

  – S3 support in the guest may not be complete and robust.

  Although virtual S3 for pass-through device live migration has its own limitation, it is still useful in some environments where virtual hotplug doesn't work, for instance, hot removal of pass-through display cards which are likely to cause a guest crash.

- **Other Guest**: Linux supports ACPI hotplug and has a powerful bonding driver, but other guest OS may not be lucky enough to have such a framework. We are in the process of extending support to other guests.

## 6 Conclusion

VM direct access of physical device achieves close to native performance, but breaks VM live migration. Our virtual ACPI hotplug device model allows VM to hot remove the pass-through device before relocation and hot add another one after relocation, thus making pass-through devices coexist with VM relocation. By integrating the Linux bonding driver into the relocation process, we enable continuous network connectivity for directly assigned NIC devices, which is the most popular pass-through device usage model.

## References

[1] "Advanced Configuration & Power Specification," Revision 3.0b, 2006, Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba. `http://www.acpi.info`

[2] "PCI Standard Hot-Plug Controller and Subsystem Specification," Revision 1.0, June, 2001, `http://www.pcisig.info`

[3] "Linux Ethernet Bonding Driver," T. Davis, W. Tarreau, C. Gavrilov, C.N. Tindel *Linux Howto Documentation, April, 2006.*

[4] "High Available Networking," M. John, *Linux Journal, January, 2006.*

[5] "Live Migration of Virtual Machines," C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfiled, In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[6] "Xen 3.0 and the Art of Virtualization," I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, In *Proceedings of the Linux Symposium* (OLS), Ottawa, Ontario, Canada, 2005.

[7] "Intel Virtualization Technology for Directed I/O Architecture Specification," 2006, `ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`

[8] "Utilizing IOMMUs for Virtualization in Linux and Xen," M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L.V. Doorn, A. Mallick, and J. Nakamima, In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, (OLS), 2006.

# Proceedings of the Linux Symposium

Volume Two

July 23rd–26th, 2008
Ottawa, Ontario
Canada