PCI Bus EEH Error Recovery

Linas Vepstas
<linas@austin.ibm.com>
12 January 2005

Overview:

The IBM POWER-based pSeries and iSeries computers include PCI bus controller chips that have extended capabilities for detecting and reporting a large variety of PCI bus error conditions. These features go under the name of "EEH", for "Extended Error Handling". The EEH hardware features allow PCI bus errors to be cleared and a PCI card to be "rebooted", without also having to reboot the operating system.

This is in contrast to traditional PCI error handling, where the PCI chip is wired directly to the CPU, and an error would cause a CPU machine-check/check-stop condition, halting the CPU entirely. Another "traditional" technique is to ignore such errors, which can lead to data corruption, both of user data or of kernel data, hung/unresponsive adapters, or system crashes/lockups. Thus, the idea behind EEH is that the operating system can become more reliable and robust by protecting it from PCI errors, and giving the OS the ability to "reboot"/recover individual PCI devices.

Future systems from other vendors, based on the PCI-E specification, may contain similar features.

Causes of EEH Errors

EEH was originally designed to guard against hardware failure, such as PCI cards dying from heat, humidity, dust, vibration and bad electrical connections. The vast majority of EEH errors seen in "real life" are due to either poorly seated PCI cards, or, unfortunately quite commonly, due to device driver bugs, device firmware bugs, and sometimes PCI card hardware bugs.

The most common software bug, is one that causes the device to attempt to DMA to a location in system memory that has not been reserved for DMA access for that card. This is a powerful feature, as it prevents what; otherwise, would have been silent memory corruption caused by the bad DMA. A number of device driver bugs have been found and fixed in this way over the past few years. Other possible causes of EEH errors include data or address line parity errors (for example, due to poor electrical connectivity due to a poorly seated card), and PCI-X split-completion errors (due to software, device firmware, or device PCI hardware bugs). The vast majority of "true hardware failures" can be cured by physically removing and re-seating the PCI card.

Detection and Recovery

In the following discussion, a generic overview of how to detect and recover from EEH errors will be presented. This is followed by an overview of how the current implementation in the Linux kernel does it. The actual implementation is subject to change, and some of the finer points are still being debated. These may in turn be swayed if or when other architectures implement similar functionality.

When a PCI Host Bridge (PHB, the bus controller connecting the PCI bus to the system CPU electronics complex) detects a PCI error condition, it will "isolate" the affected PCI card. Isolation will block all writes (either to the card from the system, or from the card to the system), and it will cause all reads to return all-ff's (0xff, 0xffff, 0xffffffff for 8/16/32-bit reads). This value was chosen because it is the same value you would get if the device was physically unplugged from the slot. This includes access to PCI memory, I/O space, and PCI config space. Interrupts; however, will continued to be delivered.

Detection and recovery are performed with the aid of ppc64 firmware. The programming interfaces in the Linux kernel into the firmware are referred to as RTAS (Run-Time Abstraction Services). The Linux kernel does not (should not) access the EEH function in the PCI chipsets directly, primarily because there are a number of different chipsets out there, each with different interfaces and quirks. The firmware provides a uniform abstraction layer that will work with all pSeries and iSeries hardware (and be forwards-compatible).

If the OS or device driver suspects that a PCI slot has been EEH-isolated, there is a firmware call it can make to determine if this is the case. If so, then the device driver should put itself into a consistent state (given that it won't be able to complete any pending work) and start recovery of the card. Recovery normally would consist of resetting the PCI device (holding the PCI #RST line high for two seconds), followed by setting up the device config space (the base address registers (BAR's), latency timer, cache line size, interrupt line, and so on). This is followed by a reinitialization of the device driver. In a worst-case scenario, the power to the card can be toggled, at least on hot-plug-capable slots. In principle, layers far above the device driver probably do not need to know that the PCI card has been "rebooted" in this way; ideally, there should be at most a pause in Ethernet/disk/USB I/O while the card is being reset.

If the card cannot be recovered after three or four resets, the kernel/device driver should assume the worst-case scenario, that the card has died completely, and report this error to the sysadmin. In addition, error messages are reported through RTAS and also through syslogd (/var/log/messages) to alert the sysadmin of PCI resets. The correct way to deal with failed adapters is to use the standard PCI hotplug tools to remove and replace the dead card.

Current PPC64 Linux EEH Implementation

At this time, a generic EEH recovery mechanism has been implemented, so that individual device drivers do not need to be modified to support EEH recovery. This generic mechanism piggy-backs on the PCI hotplug infrastructure, and percolates events up through the userspace/udev infrastructure. Following is a detailed description of how this is accomplished.

EEH must be enabled in the PHB's very early during the boot process, and if a PCI slot is hot-plugged. The former is performed by eeh_init() in arch/powerpc/platforms/pseries/eeh.c, and the later by drivers/pci/hotplug/pSeries_pci.c calling in to the eeh.c code. EEH must be enabled before a PCI scan of the device can proceed. Current Power5 hardware will not work unless EEH is enabled; although older Power4 can run with it disabled. Effectively, EEH can no longer be turned off. PCI devices *must* be registered with the EEH code; the EEH code needs to know about the I/O address ranges of the PCI device in order to detect an error. Given an arbitrary address, the routine pci_get_device_by_addr() will find the pci device associated with that address (if any).

The default arch/powerpc/include/asm/io.h macros readb(), inb(), insb(), etc. include a check to see if the i/o read returned all-0xff's. If so, these make a call to eeh_dn_check_failure(), which in turn asks the firmware if the all-ff's value is the sign of a true EEH error. If it is not, processing continues as normal. The grand total number of these false alarms or "false positives" can be seen in /proc/ppc64/eeh (subject to change). Normally, almost all of these occur during boot, when the PCI bus is scanned, where a large number of 0xff reads are part of the bus scan procedure.

If a frozen slot is detected, code in arch/powerpc/platforms/pseries/eeh.c will print a stack trace to syslog (/var/log/messages). This stack trace has proven to be very useful to device-driver authors for finding out at what point the EEH error was detected, as the error itself usually occurs slightly beforehand.

Next, it uses the Linux kernel notifier chain/work queue mechanism to allow any interested parties to find out about the failure. Device drivers, or other parts of the kernel, can use eeh_register_notifier(struct notifier_block *) to find out about EEH events. The event will include a pointer to the pci device, the device node and some state info. Receivers of the event can "do as they wish"; the default handler will be described further in this section.

To assist in the recovery of the device, eeh.c exports the following functions:

rtas_set_slot_reset() -- assert the PCI #RST line for 1/8th of a second
rtas_configure_bridge() -- ask firmware to configure any PCI bridges
   located topologically under the pci slot.
eeh_save_bars() and eeh_restore_bars(): save and restore the PCI

config-space info for a device and any devices under it.

A handler for the EEH notifier_block events is implemented in
drivers/pci/hotplug/pSeries_pci.c, called handle_eeh_events().
It saves the device BAR's and then calls rpaphp_unconfig_pci_adapter().
This last call causes the device driver for the card to be stopped,
which causes uevents to go out to user space. This triggers
user-space scripts that might issue commands such as "ifdown eth0"
for ethernet cards, and so on. This handler then sleeps for 5 seconds,
hoping to give the user-space scripts enough time to complete.
It then resets the PCI card, reconfigures the device BAR's, and
any bridges underneath. It then calls rpaphp_enable_pci_slot(),
which restarts the device driver and triggers more user-space
events (for example, calling "ifup eth0" for ethernet cards).

Device Shutdown and User-Space Events

This section documents what happens when a pci slot is unconfigured,
focusing on how the device driver gets shut down, and on how the
events get delivered to user-space scripts.

Following is an example sequence of events that cause a device driver
close function to be called during the first phase of an EEH reset.
The following sequence is an example of the pcnet32 device driver.

```
    rpa_php_unconfig_pci_adapter (struct slot *)  // in rpaphp_pci.c
    {
      calls
      pci_remove_bus_device (struct pci_dev *) // in /drivers/pci/remove.c
      {
        calls
        pci_destroy_dev (struct pci_dev *)
        {
          calls
          device_unregister (&dev->dev) // in /drivers/base/core.c
          {
            calls
            device_del (struct device *)
            {
              calls
              bus_remove_device() // in /drivers/base/bus.c
              {
                calls
                device_release_driver()
                {
                  calls
                  struct device_driver->remove() which is just
                  pci_device_remove()  // in /drivers/pci/pci_driver.c
                  {
                    calls
                    struct pci_driver->remove() which is just
                    pcnet32_remove_one() // in /drivers/net/pcnet32.c
                    {
                      calls
```

```
                        eeh-pci-error-recovery.txt
                    unregister_netdev() // in /net/core/dev.c
                    {
                      calls
                      dev_close()  // in /net/core/dev.c
                      {
                         calls dev->stop();
                         which is just pcnet32_close() // in pcnet32.c
                         {
                           which does what you wanted
                           to stop the device
                         }
                      }
                    }
                  which
                  frees pcnet32 device driver memory
               }
     }}}}}}
```

in drivers/pci/pci_driver.c,
struct device_driver->remove() is just pci_device_remove()
which calls struct pci_driver->remove() which is pcnet32_remove_one()
which calls unregister_netdev() (in net/core/dev.c)
which calls dev_close() (in net/core/dev.c)
which calls dev->stop() which is pcnet32_close()
which then does the appropriate shutdown.

Following is the analogous stack trace for events sent to user-space
when the pci device is unconfigured.

```
rpa_php_unconfig_pci_adapter() {             // in rpaphp_pci.c
  calls
  pci_remove_bus_device (struct pci_dev *) { // in /drivers/pci/remove.c
    calls
    pci_destroy_dev (struct pci_dev *) {
      calls
      device_unregister (&dev->dev) {        // in /drivers/base/core.c
        calls
        device_del(struct device * dev) {    // in /drivers/base/core.c
          calls
          kobject_del() {                    //in /libs/kobject.c
            calls
            kobject_uevent() {               // in /libs/kobject.c
              calls
              kset_uevent() {                // in /lib/kobject.c
                calls
                kset->uevent_ops->uevent()   // which is really just
                a call to
                dev_uevent() {               // in /drivers/base/core.c
                  calls
                  dev->bus->uevent() which is really just a call to
                  pci_uevent () {            // in drivers/pci/hotplug.c
                    which prints device name, etc....
                 }
               }
```

```
            then kobject_uevent() sends a netlink uevent to userspace
            --> userspace uevent
            (during early boot, nobody listens to netlink events and
            kobject_uevent() executes uevent_helper[], which runs the
            event process /sbin/hotplug)
          }
        }
        kobject_del() then calls sysfs_remove_dir(), which would
        trigger any user-space daemon that was watching /sysfs,
        and notice the delete event.
```

Pro's and Con's of the Current Design

There are several issues with the current EEH software recovery design, which may be addressed in future revisions. But first, note that the big plus of the current design is that no changes need to be made to individual device drivers, so that the current design throws a wide net. The biggest negative of the design is that it potentially disturbs network daemons and file systems that didn't need to be disturbed.

-- A minor complaint is that resetting the network card causes user-space back-to-back ifdown/ifup burps that potentially disturb network daemons, that didn't need to even know that the pci card was being rebooted.

-- A more serious concern is that the same reset, for SCSI devices, causes havoc to mounted file systems. Scripts cannot post-facto unmount a file system without flushing pending buffers, but this is impossible, because I/O has already been stopped. Thus, ideally, the reset should happen at or below the block layer, so that the file systems are not disturbed.

   Reiserfs does not tolerate errors returned from the block device. Ext3fs seems to be tolerant, retrying reads/writes until it does succeed. Both have been only lightly tested in this scenario.

   The SCSI-generic subsystem already has built-in code for performing SCSI device resets, SCSI bus resets, and SCSI host-bus-adapter (HBA) resets. These are cascaded into a chain of attempted resets if a SCSI command fails. These are completely hidden from the block layer. It would be very natural to add an EEH reset into this chain of events.

-- If a SCSI error occurs for the root device, all is lost unless the sysadmin had the foresight to run /bin, /sbin, /etc, /var and so on, out of ramdisk/tmpfs.

Conclusions

There's forward progress ...