PCI Error Recovery

Authors: Linas Vepstas <linasvepstas@gmail.com>
Richard Lary <rlary@us.ibm.com>
Mike Mason <mmlnx@us.ibm.com>

Many PCI bus controllers are able to detect a variety of hardware PCI errors on the bus, such as parity errors on the data and address buses, as well as SERR and PERR errors. Some of the more advanced chipsets are able to deal with these errors; these include PCI-E chipsets, and the PCI-host bridges found on IBM Power4, Power5 and Power6-based pSeries boxes. A typical action taken is to disconnect the affected device, halting all I/O to it. The goal of a disconnection is to avoid system corruption; for example, to halt system memory corruption due to DMA's to "wild" addresses. Typically, a reconnection mechanism is also offered, so that the affected PCI device(s) are reset and put back into working condition. The reset phase requires coordination between the affected device drivers and the PCI controller chip. This document describes a generic API for notifying device drivers of a bus disconnection, and then performing error recovery. This API is currently implemented in the 2.6.16 and later kernels.

Reporting and recovery is performed in several steps. First, when a PCI hardware error has resulted in a bus disconnect, that event is reported as soon as possible to all affected device drivers, including multiple instances of a device driver on multi-function cards. This allows device drivers to avoid deadlocking in spinloops, waiting for some i/o-space register to change, when it never will. It also gives the drivers a chance to defer incoming I/O as needed.

Next, recovery is performed in several stages. Most of the complexity is forced by the need to handle multi-function devices, that is, devices that have multiple device drivers associated with them. In the first stage, each driver is allowed to indicate what type of reset it desires, the choices being a simple re-enabling of I/O or requesting a slot reset.

If any driver requests a slot reset, that is what will be done.

After a reset and/or a re-enabling of I/O, all drivers are again notified, so that they may then perform any device setup/config that may be required. After these have all completed, a final "resume normal operations" event is sent out.

The biggest reason for choosing a kernel-based implementation rather than a user-space implementation was the need to deal with bus disconnects of PCI devices attached to storage media, and, in particular, disconnects from devices holding the root file system. If the root file system is disconnected, a user-space mechanism would have to go through a large number of contortions to complete recovery. Almost all of the current Linux file systems are not tolerant of disconnection from/reconnection to their underlying block device. By contrast, bus errors are easy to manage in the device driver. Indeed, most device drivers already handle very similar recovery procedures; for example, the SCSI-generic layer already provides significant mechanisms for dealing with SCSI bus errors and SCSI bus resets.

Detailed Design

Design and implementation details below, based on a chain of public email discussions with Ben Herrenschmidt, circa 5 April 2005.

The error recovery API support is exposed to the driver in the form of a structure of function pointers pointed to by a new field in struct pci_driver. A driver that fails to provide the structure is "non-aware", and the actual recovery steps taken are platform dependent. The arch/powerpc implementation will simulate a PCI hotplug remove/add.

This structure has the form:

```
struct pci_error_handlers
{
        int (*error_detected)(struct pci_dev *dev, pci_channel_state_t);
        int (*mmio_enabled)(struct pci_dev *dev);
        int (*slot_reset)(struct pci_dev *dev);
        void (*resume)(struct pci_dev *dev);
};
```

The possible channel states are:

```
typedef enum {
        pci_channel_io_normal,  /* I/O channel is in normal state */
        pci_channel_io_frozen,  /* I/O to channel is blocked */
        pci_channel_io_perm_failure, /* PCI card is dead */
} pci_channel_state_t;
```

Possible return values are:

```
enum pci_ers_result {
        PCI_ERS_RESULT_NONE,        /* no result/none/not supported in device driver */
        PCI_ERS_RESULT_CAN_RECOVER, /* Device driver can recover without slot reset */
        PCI_ERS_RESULT_NEED_RESET,  /* Device driver wants slot to be reset. */
        PCI_ERS_RESULT_DISCONNECT,  /* Device has completely failed, is unrecoverable */
        PCI_ERS_RESULT_RECOVERED,   /* Device driver is fully recovered and operational */
};
```

A driver does not have to implement all of these callbacks; however, if it implements any, it must implement error_detected(). If a callback is not implemented, the corresponding feature is considered unsupported. For example, if mmio_enabled() and resume() aren't there, then it is assumed that the driver is not doing any direct recovery and requires a slot reset. Typically a driver will want to know about a slot_reset().

The actual steps taken by a platform to recover from a PCI error event will be platform-dependent, but will follow the general sequence described below.

STEP 0: Error Event

A PCI bus error is detected by the PCI hardware. On powerpc, the slot is isolated, in that all I/O is blocked: all reads return 0xffffffff, all writes are ignored.

STEP 1: Notification

Platform calls the error_detected() callback on every instance of every driver affected by the error.

At this point, the device might not be accessible anymore, depending on the platform (the slot will be isolated on powerpc). The driver may already have "noticed" the error because of a failing I/O, but this is the proper "synchronization point", that is, it gives the driver a chance to cleanup, waiting for pending stuff (timers, whatever, etc...) to complete; it can take semaphores, schedule, etc... everything but touch the device. Within this function and after it returns, the driver shouldn't do any new IOs. Called in task context. This is sort of a "quiesce" point. See note about interrupts at the end of this doc.

All drivers participating in this system must implement this call. The driver must return one of the following result codes:

- PCI_ERS_RESULT_CAN_RECOVER
  Driver returns this if it thinks it might be able to recover the HW by just banging IOs or if it wants to be given a chance to extract some diagnostic information (see mmio_enable, below).
- PCI_ERS_RESULT_NEED_RESET
  Driver returns this if it can't recover without a slot reset.
- PCI_ERS_RESULT_DISCONNECT
  Driver returns this if it doesn't want to recover at all.

The next step taken will depend on the result codes returned by the drivers.

If all drivers on the segment/slot return PCI_ERS_RESULT_CAN_RECOVER, then the platform should re-enable IOs on the slot (or do nothing in particular, if the platform doesn't isolate slots), and recovery proceeds to STEP 2 (MMIO Enable).

If any driver requested a slot reset (by returning PCI_ERS_RESULT_NEED_RESET), then recovery proceeds to STEP 4 (Slot Reset).

If the platform is unable to recover the slot, the next step is STEP 6 (Permanent Failure).

Note

The current powerpc implementation assumes that a device driver will *not* schedule or semaphore in this routine; the current powerpc implementation uses one kernel thread to notify all devices; thus, if one device sleeps/schedules, all devices are affected. Doing better requires complex multi-threaded logic in the error recovery implementation (e.g. waiting for all notification threads to "join" before proceeding with recovery.) This seems excessively complex and not worth implementing.

The current powerpc implementation doesn't much care if the device attempts I/O at this point, or not. I/O's will fail, returning a value of 0xff on read, and writes will be dropped. If more than EEH_MAX_FAILS I/O's are attempted to a frozen adapter, EEH assumes that the device driver has gone into an infinite loop and prints an error to syslog. A reboot is then required to get the device working again.

STEP 2: MMIO Enabled

The platform re-enables MMIO to the device (but typically not the DMA), and then calls the mmio_enabled() callback on all affected device drivers.

This is the "early recovery" call. IOs are allowed again, but DMA is not, with some restrictions. This is NOT a callback for the driver to start operations again, only to peek/poke at the device, extract diagnostic information, if any, and eventually do things like trigger a device local reset or some such, but not restart operations. This callback is made if all drivers on a segment agree that they can try to recover and if no automatic link reset was performed by the HW. If the platform can't just re-enable IOs without a slot reset or a link reset, it will not call this callback, and instead will have gone directly to STEP 3 (Link Reset) or STEP 4 (Slot Reset)

Note

The following is proposed; no platform implements this yet: Proposal: All I/O's should be done _synchronously_ from within this callback, errors triggered by them will be returned via the normal pci_check_whatever() API, no new error_detected() callback will be issued due to an error happening here. However, such an error might cause IOs to be re-blocked for the whole segment, and thus invalidate the recovery that other devices on the same segment might have done, forcing the whole segment into one of the next states, that is, link reset or slot reset.

The driver should return one of the following result codes:

- PCI_ERS_RESULT_RECOVERED
  Driver returns this if it thinks the device is fully functional and thinks it is ready to start normal driver operations again. There is no guarantee that the driver will actually be allowed to proceed, as another driver on the same segment might have failed and thus triggered a slot reset on platforms that support it.
- PCI_ERS_RESULT_NEED_RESET
  Driver returns this if it thinks the device is not recoverable in its current state and it needs a slot reset to proceed.
- PCI_ERS_RESULT_DISCONNECT
  Same as above. Total failure, no recovery even after reset driver dead. (To be defined more precisely)

The next step taken depends on the results returned by the drivers. If all drivers returned PCI_ERS_RESULT_RECOVERED, then the platform proceeds to either STEP3 (Link Reset) or to STEP 5 (Resume Operations).

If any driver returned PCI_ERS_RESULT_NEED_RESET, then the platform proceeds to STEP 4 (Slot Reset)

STEP 3: Link Reset

The platform resets the link. This is a PCI-Express specific step and is done whenever a fatal error has been detected that can be "solved" by resetting the link.

STEP 4: Slot Reset

In response to a return value of PCI_ERS_RESULT_NEED_RESET, the platform will perform a slot reset on the requesting PCI device(s). The actual steps taken by a platform to perform a slot reset will be platform-dependent. Upon completion of slot reset, the platform will call the device slot_reset() callback.

Powerpc platforms implement two levels of slot reset: soft reset(default) and fundamental(optional) reset.

Powerpc soft reset consists of asserting the adapter #RST line and then restoring the PCI BAR's and PCI configuration header to a state that is equivalent to what it would be after a fresh system power-on followed by power-on BIOS/system firmware initialization. Soft reset is also known as hot-reset.

Powerpc fundamental reset is supported by PCI Express cards only and results in device's state machines, hardware logic, port states and configuration registers to initialize to their default conditions.

For most PCI devices, a soft reset will be sufficient for recovery. Optional fundamental reset is provided to support a limited number of PCI Express devices for which a soft reset is not sufficient for recovery.

If the platform supports PCI hotplug, then the reset might be performed by toggling the slot electrical power off/on.

It is important for the platform to restore the PCI config space to the "fresh poweron" state, rather than the "last state". After a slot reset, the device driver will almost always use its standard device initialization routines, and an unusual config space setup may result in hung devices, kernel panics, or silent data corruption.

This call gives drivers the chance to re-initialize the hardware (re-download firmware, etc.). At this point, the driver may assume that the card is in a fresh state and is fully functional. The slot is unfrozen and the driver has full access to PCI config space, memory mapped I/O space and DMA. Interrupts (Legacy, MSI, or MSI-X) will also be available.

Drivers should not restart normal I/O processing operations at this point. If all device drivers report success on this callback, the platform will call resume() to complete the sequence, and let the driver restart normal I/O processing.

A driver can still return a critical failure for this function if it can't get the device operational after reset. If the platform previously tried a soft reset, it might now try a hard reset (power cycle) and then call slot_reset() again. If the device still can't be recovered, there is nothing more that can be done; the platform will typically report a "permanent failure" in such a case. The device will be considered "dead" in this case.

Drivers for multi-function cards will need to coordinate among themselves as to which driver instance will perform any "one-shot" or global device initialization. For example, the Symbios sym53cxx2 driver performs device init only from PCI function 0:

```
+       if (PCI_FUNC(pdev->devfn) == 0)
+               sym_reset_scsi_bus(np, 0);
```

Result codes:

- PCI_ERS_RESULT_DISCONNECT Same as above.

Drivers for PCI Express cards that require a fundamental reset must set the needs_freset bit in the pci_dev structure in their probe function. For example, the QLogic qla2xxx driver sets the needs_freset bit for certain PCI card types:

```
+       /* Set EEH reset type to fundamental if required by hba  */
+       if (IS_QLA24XX(ha) || IS_QLA25XX(ha) || IS_QLA81XX(ha))
+               pdev->needs_freset = 1;
```

+

Platform proceeds either to STEP 5 (Resume Operations) or STEP 6 (Permanent Failure).

Note

The current powerpc implementation does not try a power-cycle reset if the driver returned PCI_ERS_RESULT_DISCONNECT. However, it probably should.

STEP 5: Resume Operations

The platform will call the resume() callback on all affected device drivers if all drivers on the segment have returned PCI_ERS_RESULT_RECOVERED from one of the 3 previous callbacks. The goal of this callback is to tell the driver to restart activity, that everything is back and running. This callback does not return a result code.

At this point, if a new error happens, the platform will restart a new error recovery sequence.

STEP 6: Permanent Failure

A "permanent failure" has occurred, and the platform cannot recover the device. The platform will call error_detected() with a pci_channel_state_t value of pci_channel_io_perm_failure.

The device driver should, at this point, assume the worst. It should cancel all pending I/O, refuse all new I/O, returning -EIO to higher layers. The device driver should then clean up all of its memory and remove itself from kernel operations, much as it would during system shutdown.

The platform will typically notify the system operator of the permanent failure in some way. If the device is hotplug-capable, the operator will probably want to remove and replace the device. Note, however, not all failures are truly "permanent". Some are caused by over-heating, some by a poorly seated card. Many PCI error events are caused by software bugs, e.g. DMA's to wild addresses or bogus split transactions due to programming errors. See the discussion in powerpc/eeh-pci-error-recovery.txt for additional detail on real-life experience of the causes of software errors.

Conclusion; General Remarks

The way the callbacks are called is platform policy. A platform with no slot reset capability may want to just "ignore" drivers that can't recover (disconnect them) and try to let other cards on the same segment recover. Keep in mind that in most real life cases, though, there will be only one driver per segment.

Now, a note about interrupts. If you get an interrupt and your device is dead or has been isolated, there is a problem :) The current policy is to turn this into a platform policy. That is, the recovery API only requires that:

- There is no guarantee that interrupt delivery can proceed from any device on the segment starting from the error detection and until the slot_reset callback is called, at which point interrupts are expected to be fully operational.
- There is no guarantee that interrupt delivery is stopped, that is, a driver that gets an interrupt after detecting an error, or that detects an error within the interrupt handler such that it prevents proper ack'ing of the interrupt (and thus removal of the source) should just return IRQ_NOTHANDLED. It's up to the platform to deal with that condition, typically by masking the IRQ source during the duration of the error handling. It is expected that the platform "knows" which interrupts are routed to error-management capable slots and can deal with temporarily disabling that IRQ number during error processing (this isn't terribly complex). That means some IRQ latency for other devices sharing the interrupt, but there is simply no other way. High end platforms aren't supposed to share interrupts between many devices anyway :)

Note

Implementation details for the powerpc platform are discussed in the file Documentation/powerpc/eeh-pci-error-recovery.rst

As of this writing, there is a growing list of device drivers with patches implementing error recovery. Not all of these patches are in mainline yet. These may be used as "examples":

- drivers/scsi/ipr
- drivers/scsi/sym53c8xx_2
- drivers/scsi/qla2xxx
- drivers/scsi/lpfc
- drivers/next/bnx2.c
- drivers/next/e100.c
- drivers/net/e1000
- drivers/net/e1000e
- drivers/net/ixgb
- drivers/net/ixgbe
- drivers/net/cxgb3

- drivers/net/s2io.c

The End