Reliability, Availability and Serviceability

RAS concepts

Reliability, Availability and Serviceability (RAS) is a concept used on servers meant to measure their robustness.

Reliability
: is the probability that a system will produce correct outputs.

- Generally measured as Mean Time Between Failures (MTBF)
- Enhanced by features that help to avoid, detect and repair hardware faults

Availability
: is the probability that a system is operational at a given time

- Generally measured as a percentage of downtime per a period of time
- Often uses mechanisms to detect and correct hardware faults in runtime;

Serviceability (or maintainability)
: is the simplicity and speed with which a system can be repaired or maintained

- Generally measured on Mean Time Between Repair (MTBR)

Improving RAS

In order to reduce systems downtime, a system should be capable of detecting hardware errors, and, when possible correcting them in runtime. It should also provide mechanisms to detect hardware degradation, in order to warn the system administrator to take the action of replacing a component before it causes data loss or system downtime.

Among the monitoring measures, the most usual ones include:

- CPU â€“ detect errors at instruction execution and at L1/L2/L3 caches;
- Memory â€“ add error correction logic (ECC) to detect and correct errors;
- I/O â€“ add CRC checksums for transferred data;
- Storage â€“ RAID, journal file systems, checksums, Self-Monitoring, Analysis and Reporting Technology (SMART).

By monitoring the number of occurrences of error detections, it is possible to identify if the probability of hardware errors is increasing, and, on such case, do a preventive maintenance to replace a degraded component while those errors are correctable.

Types of errors

Most mechanisms used on modern systems use technologies like Hamming Codes that allow error correction when the number of errors on a bit packet is below a threshold. If the number of errors is above, those mechanisms can indicate with a high degree of confidence that an error happened, but they can't correct.

Also, sometimes an error occur on a component that it is not used. For example, a part of the memory that it is not currently allocated.

That defines some categories of errors:

- **Correctable Error (CE)** - the error detection mechanism detected and corrected the error. Such errors are usually not fatal, although some Kernel mechanisms allow the system administrator to consider them as fatal.
- **Uncorrected Error (UE)** - the amount of errors happened above the error correction threshold, and the system was unable to auto-correct.
- **Fatal Error** - when an UE error happens on a critical component of the system (for example, a piece of the Kernel got corrupted by an UE), the only reliable way to avoid data corruption is to hang or reboot the machine.
- **Non-fatal Error** - when an UE error happens on an unused component, like a CPU in power down state or an unused memory bank, the system may still run, eventually replacing the affected hardware by a hot spare, if available.

  Also, when an error happens on a userspace process, it is also possible to kill such process and let userspace restart it.

The mechanism for handling non-fatal errors is usually complex and may require the help of some userspace application, in order to apply the policy desired by the system administrator.

Identifying a bad hardware component

Just detecting a hardware flaw is usually not enough, as the system needs to pinpoint to the minimal replaceable unit (MRU) that should be exchanged to make the hardware reliable again.

So, it requires not only error logging facilities, but also mechanisms that will translate the error message to the silkscreen or component label for the MRU.

Typically, it is very complex for memory, as modern CPUs interlace memory from different memory modules, in order to provide a better performance. The DMI BIOS usually have a list of memory module labels, with can be obtained using the `dmidecode` tool. For example, on a desktop machine, it shows:

```
Memory Device
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 16384 MB
        Form Factor: SODIMM
        Set: None
        Locator: ChannelA-DIMM0
        Bank Locator: BANK 0
        Type: DDR4
        Type Detail: Synchronous
        Speed: 2133 MHz
        Rank: 2
        Configured Clock Speed: 2133 MHz
```

On the above example, a DDR4 SO-DIMM memory module is located at the system's memory labeled as "BANK 0", as given by the *bank locator* field. Please notice that, on such system, the *total width* is equal to the *data width*. It means that such memory module doesn't have error detection/correction mechanisms.

Unfortunately, not all systems use the same field to specify the memory bank. On this example, from an older server, `dmidecode` shows:

```
Memory Device
        Array Handle: 0x1000
        Error Information Handle: Not Provided
        Total Width: 72 bits
        Data Width: 64 bits
        Size: 8192 MB
        Form Factor: DIMM
        Set: 1
        Locator: DIMM_A1
        Bank Locator: Not Specified
        Type: DDR3
        Type Detail: Synchronous Registered (Buffered)
        Speed: 1600 MHz
        Rank: 2
        Configured Clock Speed: 1600 MHz
```

There, the DDR3 RDIMM memory module is located at the system's memory labeled as "DIMM_A1", as given by the *locator* field. Please notice that this memory module has 64 bits of *data width* and 72 bits of *total width*. So, it has 8 extra bits to be used by error detection and correction mechanisms. Such kind of memory is called Error-correcting code memory (ECC memory).

To make things even worse, it is not uncommon that systems with different labels on their system's board to use exactly the same BIOS, meaning that the labels provided by the BIOS won't match the real ones.

ECC memory

As mentioned in the previous section, ECC memory has extra bits to be used for error correction. In the above example, a memory module has 64 bits of *data width*, and 72 bits of *total width*. The extra 8 bits which are used for the error detection and correction mechanisms are referred to as the *syndrome*[1][2].

So, when the cpu requests the memory controller to write a word with *data width*, the memory controller calculates the *syndrome* in real time, using Hamming code, or some other error correction code, like SECDED+, producing a code with *total width* size. Such code is then written on the memory modules.

At read, the *total width* bits code is converted back, using the same ECC code used on write, producing a word with *data width* and a *syndrome*. The word with *data width* is sent to the CPU, even when errors happen.

The memory controller also looks at the *syndrome* in order to check if there was an error, and if the ECC code was able to fix such error. If the error was corrected, a Corrected Error (CE) happened. If not, an Uncorrected Error (UE) happened.

The information about the CE/UE errors is stored on some special registers at the memory controller and can be accessed by reading such registers, either by BIOS, by some special CPUs or by Linux EDAC driver. On x86 64 bit CPUs, such errors can also be retrieved via the Machine Check Architecture (MCA)[3].

[1] Please notice that several memory controllers allow operation on a mode called "Lock-Step", where it groups two memory modules together, doing 128-bit reads/writes. That gives 16 bits for error correction, with significantly improves the error correction mechanism, at the expense that, when an error happens, there's no way to know what memory module is to blame. So, it has to blame both memory modules.

[2] Some memory controllers also allow using memory in mirror mode. On such mode, the same data is written to two memory modules. At read, the system checks both memory modules, in order to check if both provide identical data. On such configuration, when an error happens, there's no way to know what memory module is to blame. So, it has to blame both memory modules (or 4 memory modules, if the system is also on Lock-step mode).

[3] For more details about the Machine Check Architecture (MCA), please read Documentation/x86/x86_64/machinecheck.rst at the Kernel tree.

EDAC - Error Detection And Correction

> **Note**
>
> "bluesmoke" was the name for this device driver subsystem when it was "out-of-tree" and maintained at http://bluesmoke.sourceforge.net. That site is mostly archaic now and can be used only for historical purposes.
>
> When the subsystem was pushed upstream for the first time, on Kernel 2.6.16, it was renamed to `EDAC`.

Purpose

The `edac` kernel module's goal is to detect and report hardware errors that occur within the computer system running under linux.

Memory

Memory Correctable Errors (CE) and Uncorrectable Errors (UE) are the primary errors being harvested. These types of errors are harvested by the `edac_mc` device.

Detecting CE events, then harvesting those events and reporting them, **can** but must not necessarily be a predictor of future UE events. With CE events only, the system can and will continue to operate as no data has been damaged yet.

However, preventive maintenance and proactive part replacement of memory modules exhibiting CEs can reduce the likelihood of the dreaded UE events and system panics.

Other hardware elements

A new feature for EDAC, the `edac_device` class of device, was added in the 2.6.23 version of the kernel.

This new device type allows for non-memory type of ECC hardware detectors to have their states harvested and presented to userspace via the sysfs interface.

Some architectures have ECC detectors for L1, L2 and L3 caches, along with DMA engines, fabric switches, main data path switches, interconnections, and various other hardware data paths. If the hardware reports it, then a edac_device device probably can be constructed to harvest and present that to userspace.

PCI bus scanning

In addition, PCI devices are scanned for PCI Bus Parity and SERR Errors in order to determine if errors are occurring during data transfers.

The presence of PCI Parity errors must be examined with a grain of salt. There are several add-in adapters that do **not** follow the PCI specification with regards to Parity generation and reporting. The specification says the vendor should tie the parity status bits to 0 if they do not intend to generate parity. Some vendors do not do this, and thus the parity bit can "float" giving false positives.

There is a PCI device attribute located in sysfs that is checked by the EDAC PCI scanning code. If that attribute is set, PCI parity/error scanning is skipped for that device. The attribute is:

```
broken_parity_status
```

and is located in `/sys/devices/pci<XXX>/0000:XX:YY.Z` directories for PCI devices.

Versioning

EDAC is composed of a "core" module (`edac_core.ko`) and several Memory Controller (MC) driver modules. On a given system, the CORE is loaded and one MC driver will be loaded. Both the CORE and the MC driver (or `edac_device` driver) have individual versions that reflect current release level of their respective modules.

Thus, to "report" on what version a system is running, one must report both the CORE's and the MC driver's versions.

Loading

If `edac` was statically linked with the kernel then no loading is necessary. If `edac` was built as modules then simply modprobe the `edac` pieces that you need. You should be able to modprobe hardware-specific modules and have the dependencies load the necessary core modules.

Example:

```
$ modprobe amd76x_edac
```

loads both the `amd76x_edac.ko` memory controller module and the `edac_mc.ko` core module.

Sysfs interface

EDAC presents a `sysfs` interface for control and reporting purposes. It lives in the /sys/devices/system/edac directory.

Within this directory there currently reside 2 components:

mc	memory controller(s) system
pci	PCI control and status system

Memory Controller (mc) Model

Each `mc` device controls a set of memory modules [4]. These modules are laid out in a Chip-Select Row (`csrowX`) and Channel table (`chX`). There can be multiple csrows and multiple channels.

[4] Nowadays, the term DIMM (Dual In-line Memory Module) is widely used to refer to a memory module, although there are other memory packaging alternatives, like SO-DIMM, SIMM, etc. The UEFI specification (Version 2.7) defines a memory module in the Common Platform Error Record (CPER) section to be an SMBIOS Memory Device (Type 17). Along this

> document, and inside the EDAC subsystem, the term "dimm" is used for all memory modules, even when they use a different kind of packaging.

Memory controllers allow for several csrows, with 8 csrows being a typical value. Yet, the actual number of csrows depends on the layout of a given motherboard, memory controller and memory module characteristics.

Dual channels allow for dual data length (e. g. 128 bits, on 64 bit systems) data transfers to/from the CPU from/to memory. Some newer chipsets allow for more than 2 channels, like Fully Buffered DIMMs (FB-DIMMs) memory controllers. The following example will assume 2 channels:

CS Rows	**Channels**	
	ch0	**ch1**
	DIMM_A0	**DIMM_B0**
csrow0	rank0	rank0
csrow1	rank1	rank1
	DIMM_A1	**DIMM_B1**
csrow2	rank0	rank0
csrow3	rank1	rank1

In the above example, there are 4 physical slots on the motherboard for memory DIMMs:

DIMM_A0	DIMM_B0
DIMM_A1	DIMM_B1

Labels for these slots are usually silk-screened on the motherboard. Slots labeled `A` are channel 0 in this example. Slots labeled `B` are channel 1. Notice that there are two csrows possible on a physical DIMM. These csrows are allocated their csrow assignment based on the slot into which the memory DIMM is placed. Thus, when 1 DIMM is placed in each Channel, the csrows cross both DIMMs.

Memory DIMMs come single or dual "ranked". A rank is a populated csrow. In the example above 2 dual ranked DIMMs are similarly placed. Thus, both csrow0 and csrow1 are populated. On the other hand, when 2 single ranked DIMMs are placed in slots DIMM_A0 and DIMM_B0, then they will have just one csrow (csrow0) and csrow1 will be empty. The pattern repeats itself for csrow2 and csrow3. Also note that some memory controllers don't have any logic to identify the memory module, see `rankX` directories below.

The representation of the above is reflected in the directory tree in EDAC's sysfs interface. Starting in directory `/sys/devices/system/edac/mc`, each memory controller will be represented by its own `mcX` directory, where `X` is the index of the MC:

```
..../edac/mc/
           |
           |->mc0
           |->mc1
           |->mc2
           ....
```

Under each `mcX` directory each `csrowX` is again represented by a `csrowX`, where `X` is the csrow index:

```
.../mc/mc0/
        |
        |->csrow0
        |->csrow2
        |->csrow3
        ....
```

Notice that there is no csrow1, which indicates that csrow0 is composed of a single ranked DIMMs. This should also apply in both Channels, in order to have dual-channel mode be operational. Since both csrow2 and csrow3 are populated, this indicates a dual ranked set of DIMMs for channels 0 and 1.

Within each of the `mcX` and `csrowX` directories are several EDAC control and attribute files.

`mcX` directories

In `mcX` directories are EDAC control and attribute files for this `X` instance of the memory controllers.

For a description of the sysfs API, please see:

> Documentation/ABI/testing/sysfs-devices-edac

`dimmX` or `rankX` directories

The recommended way to use the EDAC subsystem is to look at the information provided by the `dimmX` or `rankX` directories [5].

A typical EDAC system has the following structure under `/sys/devices/system/edac/`[6]:

```
/sys/devices/system/edac/
â”œâ”€â”€ mc
â”‚Â Â  â”œâ”€â”€ mc0
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ce_count
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ce_noinfo_count
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm0
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_ce_count
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_dev_type
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_edac_mode
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_label
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_location
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_mem_type
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_ue_count
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ size
â”‚Â Â  â”‚Â Â  â”‚Â Â  â””â”€â”€ uevent
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ max_location
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ mc_name
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ reset_counters
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ seconds_since_reset
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ size_mb
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ue_count
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ue_noinfo_count
â”‚Â Â  â”‚Â Â  â””â”€â”€ uevent
â”‚Â Â  â”œâ”€â”€ mc1
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ce_count
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ce_noinfo_count
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm0
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_ce_count
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_dev_type
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_edac_mode
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_label
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_location
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_mem_type
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ dimm_ue_count
â”‚Â Â  â”‚Â Â  â”‚Â Â  â”œâ”€â”€ size
â”‚Â Â  â”‚Â Â  â”‚Â Â  â””â”€â”€ uevent
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ max_location
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ mc_name
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ reset_counters
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ seconds_since_reset
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ size_mb
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ue_count
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ ue_noinfo_count
â”‚Â Â  â”‚Â Â  â”œâ”€â”€ uevent
â”‚Â Â  â”‚Â Â  â””â”€â”€ uevent
â””â”€â”€ uevent
```

In the `dimmX` directories are EDAC control and attribute files for this `X` memory module:

- `size` - Total memory managed by this csrow attribute file

This attribute file displays, in count of megabytes, the memory that this csrow contains.

- `dimm_ue_count` - Uncorrectable Errors count attribute file

  This attribute file displays the total count of uncorrectable errors that have occurred on this DIMM. If panic_on_ue is set this counter will not have a chance to increment, since EDAC will panic the system.

- `dimm_ce_count` - Correctable Errors count attribute file

  This attribute file displays the total count of correctable errors that have occurred on this DIMM. This count is very important to examine. CEs provide early indications that a DIMM is beginning to fail. This count field should be monitored for non-zero values and report such information to the system administrator.

- `dimm_dev_type` - Device type attribute file

  This attribute file will display what type of DRAM device is being utilized on this DIMM. Examples:

  - x1
  - x2
  - x4
  - x8

- `dimm_edac_mode` - EDAC Mode of operation attribute file

  This attribute file will display what type of Error detection and correction is being utilized.

- `dimm_label` - memory module label control file

  This control file allows this DIMM to have a label assigned to it. With this label in the module, when errors occur the output can provide the DIMM label in the system log. This becomes vital for panic events to isolate the cause of the UE event.

  DIMM Labels must be assigned after booting, with information that correctly identifies the physical slot with its silk screen label. This information is currently very motherboard specific and determination of this information must occur in userland at this time.

- `dimm_location` - location of the memory module

  The location can have up to 3 levels, and describe how the memory controller identifies the location of a memory module. Depending on the type of memory and memory controller, it can be:

  - *csrow* and *channel* - used when the memory controller doesn't identify a single DIMM - e. g. in `rankX` dir;
  - *branch*, *channel*, *slot* - typically used on FB-DIMM memory controllers;
  - *channel*, *slot* - used on Nehalem and newer Intel drivers.

- `dimm_mem_type` - Memory Type attribute file

  This attribute file will display what type of memory is currently on this csrow. Normally, either buffered or unbuffered memory. Examples:

  - Registered-DDR
  - Unbuffered-DDR

[5] On some systems, the memory controller doesn't have any logic to identify the memory module. On such systems, the directory is called `rankX` and works on a similar way as the `csrowX` directories. On modern Intel memory controllers, the memory controller identifies the memory modules directly. On such systems, the directory is called `dimmX`.

[6] There are also some `power` directories and `subsystem` symlinks inside the sysfs mapping that are automatically created by the sysfs subsystem. Currently, they serve no purpose.

`csrowX` directories

When CONFIG_EDAC_LEGACY_SYSFS is enabled, sysfs will contain the `csrowX` directories. As this API doesn't work properly for Rambus, FB-DIMMs and modern Intel Memory Controllers, this is being deprecated in favor of `dimmX` directories.

In the `csrowX` directories are EDAC control and attribute files for this `X` instance of csrow:

- `ue_count` - Total Uncorrectable Errors count attribute file

  This attribute file displays the total count of uncorrectable errors that have occurred on this csrow. If panic_on_ue is set this counter will not have a chance to increment, since EDAC will panic the system.

- `ce_count` - Total Correctable Errors count attribute file

  This attribute file displays the total count of correctable errors that have occurred on this csrow. This count is very important to examine. CEs provide early indications that a DIMM is beginning to fail. This count field should be monitored for non-zero values and report such information to the system administrator.

- `size_mb` - Total memory managed by this csrow attribute file

  This attribute file displays, in count of megabytes, the memory that this csrow contains.

- `mem_type` - Memory Type attribute file

  This attribute file will display what type of memory is currently on this csrow. Normally, either buffered or unbuffered memory. Examples:

  - Registered-DDR
  - Unbuffered-DDR

- `edac_mode` - EDAC Mode of operation attribute file

  This attribute file will display what type of Error detection and correction is being utilized.

- `dev_type` - Device type attribute file

  This attribute file will display what type of DRAM device is being utilized on this DIMM. Examples:

  - x1
  - x2
  - x4
  - x8

- `ch0_ce_count` - Channel 0 CE Count attribute file

  This attribute file will display the count of CEs on this DIMM located in channel 0.

- `ch0_ue_count` - Channel 0 UE Count attribute file

  This attribute file will display the count of UEs on this DIMM located in channel 0.

- `ch0_dimm_label` - Channel 0 DIMM Label control file

  This control file allows this DIMM to have a label assigned to it. With this label in the module, when errors occur the

output can provide the DIMM label in the system log. This becomes vital for panic events to isolate the cause of the UE event.

DIMM Labels must be assigned after booting, with information that correctly identifies the physical slot with its silk screen label. This information is currently very motherboard specific and determination of this information must occur in userland at this time.

- `ch1_ce_count` - Channel 1 CE Count attribute file

  This attribute file will display the count of CEs on this DIMM located in channel 1.

- `ch1_ue_count` - Channel 1 UE Count attribute file

  This attribute file will display the count of UEs on this DIMM located in channel 0.

- `ch1_dimm_label` - Channel 1 DIMM Label control file

  This control file allows this DIMM to have a label assigned to it. With this label in the module, when errors occur the output can provide the DIMM label in the system log. This becomes vital for panic events to isolate the cause of the UE event.

  DIMM Labels must be assigned after booting, with information that correctly identifies the physical slot with its silk screen label. This information is currently very motherboard specific and determination of this information must occur in userland at this time.

System Logging

If logging for UEs and CEs is enabled, then system logs will contain information indicating that errors have been detected:

```
EDAC MC0: CE page 0x283, offset 0xce0, grain 8, syndrome 0x6ec3, row 0, channel 1 "DIMM_B1": amd76x_edac
EDAC MC0: CE page 0x1e5, offset 0xfb0, grain 8, syndrome 0xb741, row 0, channel 1 "DIMM_B1": amd76x_edac
```

The structure of the message is:

Content	Example
The memory controller	MC0
Error type	CE
Memory page	0x283
Offset in the page	0xce0
The byte granularity or resolution of the error	grain 8
The error syndrome	0xb741
Memory row	row 0
Memory channel	channel 1
DIMM label, if set prior	DIMM B1
And then an optional, driver-specific message that may have additional information.	

Both UEs and CEs with no info will lack all but memory controller, error type, a notice of "no info" and then an optional, driver-specific error message.

PCI Bus Parity Detection

On Header Type 00 devices, the primary status is looked at for any parity error regardless of whether parity is enabled on the device or not. (The spec indicates parity is generated in some cases). On Header Type 01 bridges, the secondary status register is also looked at to see if parity occurred on the bus on the other side of the bridge.

Sysfs configuration

Under `/sys/devices/system/edac/pci` are control and attribute files as follows:

- `check_pci_parity` - Enable/Disable PCI Parity checking control file

  This control file enables or disables the PCI Bus Parity scanning operation. Writing a 1 to this file enables the scanning. Writing a 0 to this file disables the scanning.

  Enable:

  ```
  echo "1" >/sys/devices/system/edac/pci/check_pci_parity
  ```

  Disable:

  ```
  echo "0" >/sys/devices/system/edac/pci/check_pci_parity
  ```

- `pci_parity_count` - Parity Count

  This attribute file will display the number of parity errors that have been detected.

Module parameters

- `edac_mc_panic_on_ue` - Panic on UE control file

  An uncorrectable error will cause a machine panic. This is usually desirable. It is a bad idea to continue when an uncorrectable error occurs - it is indeterminate what was uncorrected and the operating system context might be so mangled that continuing will lead to further corruption. If the kernel has MCE configured, then EDAC will never notice the UE.

  LOAD TIME:

  ```
  module/kernel parameter: edac_mc_panic_on_ue=[0|1]
  ```

  RUN TIME:

  ```
  echo "1" > /sys/module/edac_core/parameters/edac_mc_panic_on_ue
  ```

- `edac_mc_log_ue` - Log UE control file

  Generate kernel messages describing uncorrectable errors. These errors are reported through the system message log system. UE statistics will be accumulated even when UE logging is disabled.

  LOAD TIME:

  ```
  module/kernel parameter: edac_mc_log_ue=[0|1]
  ```

  RUN TIME:

  ```
  echo "1" > /sys/module/edac_core/parameters/edac_mc_log_ue
  ```

- `edac_mc_log_ce` - Log CE control file

  Generate kernel messages describing correctable errors. These errors are reported through the system message log system. CE statistics will be accumulated even when CE logging is disabled.

  LOAD TIME:

  ```
  module/kernel parameter: edac_mc_log_ce=[0|1]
  ```

  RUN TIME:

  ```
  echo "1" > /sys/module/edac_core/parameters/edac_mc_log_ce
  ```

- `edac_mc_poll_msec` - Polling period control file

  The time period, in milliseconds, for polling for error information. Too small a value wastes resources. Too large a value might delay necessary handling of errors and might loose valuable information for locating the error. 1000 milliseconds (once each second) is the current default. Systems which require all the bandwidth they can get, may increase this.

  LOAD TIME:

  ```
  module/kernel parameter: edac_mc_poll_msec=[0|1]
  ```

  RUN TIME:

  ```
  echo "1000" > /sys/module/edac_core/parameters/edac_mc_poll_msec
  ```

- `panic_on_pci_parity` - Panic on PCI PARITY Error

  This control file enables or disables panicking when a parity error has been detected.

  module/kernel parameter:

  ```
  edac_panic_on_pci_pe=[0|1]
  ```

  Enable:

  ```
  echo "1" > /sys/module/edac_core/parameters/edac_panic_on_pci_pe
  ```

  Disable:

  ```
  echo "0" > /sys/module/edac_core/parameters/edac_panic_on_pci_pe
  ```

EDAC device type

In the header file, edac_pci.h, there is a series of edac_device structures and APIs for the EDAC_DEVICE.

User space access to an edac_device is through the sysfs interface.

At the location `/sys/devices/system/edac` (sysfs) new edac_device devices will appear.

There is a three level tree beneath the above `edac` directory. For example, the `test_device_edac` device (found at the http://bluesmoke.sourceforget.net website) installs itself as:

```
/sys/devices/system/edac/test-instance
```

in this directory are various controls, a symlink and one or more `instance` directories.

The standard default controls are:

log_ce	boolean to log CE events
log_ue	boolean to log UE events
panic_on_ue	boolean to `panic` the system if an UE is encountered (default off, can be set true via startup script)
poll_msec	time period between POLL cycles for events

The test_device_edac device adds at least one of its own custom control:

test_bits	which in the current test driver does nothing but show how it is installed. A ported driver can add one or more such controls and/or attributes for specific uses. One out-of-tree driver uses controls here to allow for ERROR INJECTION operations to hardware injection registers

The symlink points to the 'struct dev' that is registered for this edac_device.

Instances

One or more instance directories are present. For the `test_device_edac` case:

```
test-instance0
```

In this directory there are two default counter attributes, which are totals of counter in deeper subdirectories.

ce_count	total of CE events of subdirectories
ue_count	total of UE events of subdirectories

Blocks

At the lowest directory level is the `block` directory. There can be 0, 1 or more blocks specified in each instance:

```
test-block0
```

In this directory the default attributes are:

ce_count	which is counter of CE events for this `block` of hardware being monitored
ue_count	which is counter of UE events for this `block` of hardware being monitored

The `test_device_edac` device adds 4 attributes and 1 control:

test-block-bits-0	for every POLL cycle this counter is incremented
test-block-bits-1	every 10 cycles, this counter is bumped once, and test-block-bits-0 is set to 0
test-block-bits-2	every 100 cycles, this counter is bumped once, and test-block-bits-1 is set to 0
test-block-bits-3	every 1000 cycles, this counter is bumped once, and test-block-bits-2 is set to 0
reset-counters	writing ANY thing to this control will reset all the above counters.

Use of the `test_device_edac` driver should enable any others to create their own unique drivers for their hardware systems.

The `test_device_edac` sample driver is located at the http://bluesmoke.sourceforge.net project site for EDAC.

Usage of EDAC APIs on Nehalem and newer Intel CPUs

On older Intel architectures, the memory controller was part of the North Bridge chipset. Nehalem, Sandy Bridge, Ivy Bridge, Haswell, Sky Lake and newer Intel architectures integrated an enhanced version of the memory controller (MC) inside the CPUs.

This chapter will cover the differences of the enhanced memory controllers found on newer Intel CPUs, such as `i7core_edac`, `sb_edac` and `sbx_edac` drivers.

> **Note**
>
> The Xeon E7 processor families use a separate chip for the memory controller, called Intel Scalable Memory Buffer. This section doesn't apply for such families.

1. There is one Memory Controller per Quick Patch Interconnect (QPI). At the driver, the term "socket" means one QPI. This is associated with a physical CPU socket.

Each MC have 3 physical read channels, 3 physical write channels and 3 logic channels. The driver currently sees it as just 3 channels. Each channel can have up to 3 DIMMs.

The minimum known unity is DIMMs. There are no information about csrows. As EDAC API maps the minimum unity is csrows, the driver sequentially maps channel/DIMM into different csrows.

For example, supposing the following layout:

```
Ch0 phy rd0, wr0 (0x063f4031): 2 ranks, UDIMMs
  dimm 0 1024 Mb offset: 0, bank: 8, rank: 1, row: 0x4000, col: 0x400
  dimm 1 1024 Mb offset: 4, bank: 8, rank: 1, row: 0x4000, col: 0x400
Ch1 phy rd1, wr1 (0x063f4031): 2 ranks, UDIMMs
  dimm 0 1024 Mb offset: 0, bank: 8, rank: 1, row: 0x4000, col: 0x400
Ch2 phy rd3, wr3 (0x063f4031): 2 ranks, UDIMMs
  dimm 0 1024 Mb offset: 0, bank: 8, rank: 1, row: 0x4000, col: 0x400
```

The driver will map it as:

```
csrow0: channel 0, dimm0
csrow1: channel 0, dimm1
csrow2: channel 1, dimm0
csrow3: channel 2, dimm0
```

exports one DIMM per csrow.

Each QPI is exported as a different memory controller.

2. The MC has the ability to inject errors to test drivers. The drivers implement this functionality via some error injection nodes:

   For injecting a memory error, there are some sysfs nodes, under `/sys/devices/system/edac/mc/mc?/`:

   - `inject_addrmatch/*`:

     Controls the error injection mask register. It is possible to specify several characteristics of the address to match an error code:

     ```
     dimm = the affected dimm. Numbers are relative to a channel;
     rank = the memory rank;
     channel = the channel that will generate an error;
     bank = the affected bank;
     page = the page address;
     column (or col) = the address column.
     ```

     each of the above values can be set to "any" to match any valid value.

     At driver init, all values are set to any.

     For example, to generate an error at rank 1 of dimm 2, for any channel, any bank, any page, any column:

     ```
             echo 2 >/sys/devices/system/edac/mc/mc0/inject_addrmatch/dimm
             echo 1 >/sys/devices/system/edac/mc/mc0/inject_addrmatch/rank

        To return to the default behaviour of matching any, you can do::

             echo any >/sys/devices/system/edac/mc/mc0/inject_addrmatch/dimm
             echo any >/sys/devices/system/edac/mc/mc0/inject_addrmatch/rank
     ```

   - `inject_eccmask`:

     specifies what bits will have troubles,

   - `inject_section`:

     specifies what ECC cache section will get the error:

     ```
     3 for both
     2 for the highest
     1 for the lowest
     ```

   - `inject_type`:

     specifies the type of error, being a combination of the following bits:

     ```
     bit 0 - repeat
     bit 1 - ecc
     bit 2 - parity
     ```

   - `inject_enable`:

     starts the error generation when something different than 0 is written.

   All inject vars can be read. root permission is needed for write.

   Datasheet states that the error will only be generated after a write on an address that matches inject_addrmatch. It seems, however, that reading will also produce an error.

   For example, the following code will generate an error for any write access at socket 0, on any DIMM/address on channel 2:

   ```
   echo 2 >/sys/devices/system/edac/mc/mc0/inject_addrmatch/channel
   echo 2 >/sys/devices/system/edac/mc/mc0/inject_type
   echo 64 >/sys/devices/system/edac/mc/mc0/inject_eccmask
   echo 3 >/sys/devices/system/edac/mc/mc0/inject_section
   echo 1 >/sys/devices/system/edac/mc/mc0/inject_enable
   dd if=/dev/mem of=/dev/null seek=16k bs=4k count=1 >& /dev/null
   ```

   For socket 1, it is needed to replace "mc0" by "mc1" at the above commands.

   The generated error message will look like:

   ```
   EDAC MC0: UE row 0, channel-a= 0 channel-b= 0 labels "-": NON_FATAL (addr = 0x0075b980, socket=0, Dimm=0, Channel=2, syndrome=0x0
   ```

3. Corrected Error memory register counters

   Those newer MCs have some registers to count memory errors. The driver uses those registers to report Corrected Errors on devices with Registered DIMMs.

   However, those counters don't work with Unregistered DIMM. As the chipset offers some counters that also work with UDIMMs (but with a worse level of granularity than the default ones), the driver exposes those registers for UDIMM memories.

   They can be read by looking at the contents of `all_channel_counts/`:

   ```
   $ for i in /sys/devices/system/edac/mc/mc0/all_channel_counts/*; do echo $i; cat $i; done
      /sys/devices/system/edac/mc/mc0/all_channel_counts/udimm0
      0
      /sys/devices/system/edac/mc/mc0/all_channel_counts/udimm1
      0
      /sys/devices/system/edac/mc/mc0/all_channel_counts/udimm2
      0
   ```

   What happens here is that errors on different csrows, but at the same dimm number will increment the same counter. So, in this memory mapping:

   ```
   csrow0: channel 0, dimm0
   csrow1: channel 0, dimm1
   csrow2: channel 1, dimm0
   csrow3: channel 2, dimm0
   ```

   The hardware will increment udimm0 for an error at the first dimm at either csrow0, csrow2 or csrow3;

   The hardware will increment udimm1 for an error at the second dimm at either csrow0, csrow2 or csrow3;

   The hardware will increment udimm2 for an error at the third dimm at either csrow0, csrow2 or csrow3;

4. Standard error counters

The standard error counters are generated when an mcelog error is received by the driver. Since, with UDIMM, this is counted by software, it is possible that some errors could be lost. With RDIMM's, they display the contents of the registers

Reference documents used on `amd64_edac`

`amd64_edac` module is based on the following documents (available from http://support.amd.com/en-us/search/tech-docs):

1. **Title:** BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors
   AMD publication #: 26094
   Revision: 3.26
   Link: http://support.amd.com/TechDocs/26094.PDF
2. **Title:** BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors
   AMD publication #: 32559
   Revision: 3.00
   Issue Date: May 2006
   Link: http://support.amd.com/TechDocs/32559.pdf
3. **Title:** BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors
   AMD publication #: 31116
   Revision: 3.00
   Issue Date: September 07, 2007
   Link: http://support.amd.com/TechDocs/31116.pdf
4. **Title:** BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 30h-3Fh Processors
   AMD publication #: 49125
   Revision: 3.06
   Issue Date: 2/12/2015 (latest release)
   Link: http://support.amd.com/TechDocs/49125_15h_Models_30h-3Fh_BKDG.pdf
5. **Title:** BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 60h-6Fh Processors
   AMD publication #: 50742
   Revision: 3.01
   Issue Date: 7/23/2015 (latest release)
   Link: http://support.amd.com/TechDocs/50742_15h_Models_60h-6Fh_BKDG.pdf
6. **Title:** BIOS and Kernel Developer's Guide (BKDG) for AMD Family 16h Models 00h-0Fh Processors
   AMD publication #: 48751
   Revision: 3.03
   Issue Date: 2/23/2015 (latest release)
   Link: http://support.amd.com/TechDocs/48751_16h_bkdg.pdf

Credits

- Written by Doug Thompson <dougthompson@xmission.com>
  - 7 Dec 2005
  - 17 Jul 2007 Updated
- © Mauro Carvalho Chehab
  - 05 Aug 2009 Nehalem interface
  - 26 Oct 2016 Converted to ReST and cleanups at the Nehalem section
- EDAC authors/maintainers:
  - Doug Thompson, Dave Jiang, Dave Peterson et al,
  - Mauro Carvalho Chehab
  - Borislav Petkov
  - original author: Thayne Harbaugh