CHAPTER 17

Reliability, Availability, and Serviceability (RAS)

This chapter describes the high-level architecture of RAS features designed for persistent memory. Persistent memory RAS features were designed to support the unique error-handling strategy required for an application when persistent memory is used. Error handling is an important part of the program's overall reliability, which directly affects the availability of applications. The error-handling strategy for applications impacts what percentage of the expected time the application is available to do its job.

Persistent memory vendors and platform vendors will both decide which RAS features and how they will be implemented at the lowest hardware levels. Some common RAS features were designed and documented in the ACPI specification, which is maintained and owned by the UEFI Forum (https://uefi.org/). In this chapter, we try to attain a general perspective of these ACPI-defined RAS features and call out vendor-specific details if warranted.

Dealing with Uncorrectable Errors

The main memory of a server is protected using Error Correcting Codes (ECC). This is a common hardware feature that can automatically correct many memory errors that happen due to transient hardware issues, such as power spikes, soft media errors, and so on. If an error is severe enough, it will corrupt enough bits that ECC cannot correct; the result is called an uncorrectable error (UE).

Uncorrectable errors in persistent memory require special RAS handling that differs from how a platform may traditionally handle volatile memory uncorrectable errors.

Persistent memory uncorrectable errors are *persistent*. Unlike volatile memory, if power is lost or an application crashes and restarts, the uncorrectable error will remain on the hardware. This can lead to an application getting stuck in an infinite loop such as:

1. Application starts
2. Reads a memory address
3. Encounters uncorrectable error
4. Crashes (or system crashes and reboots)
5. Starts and resumes operation from where it left off
6. Performs a read on the same memory address that triggered the previous restart
7. Crashes (or system crashes and reboots)
8. …
9. Repeats infinitely until manual intervention

The operating system and applications may need to address uncorrectable errors in three main ways:

- When consuming previously undetected uncorrectable errors during runtime.
- When unconsumed uncorrectable errors that have been detected at runtime.
- When mitigating uncorrectable memory locations detected at boot.

Consumed Uncorrectable Error Handling

When an uncorrectable error is detected on a requested memory address, data poisoning is used to inform the CPU that the data requested is an uncorrectable error. When the hardware detects an uncorrectable memory error, it routes a poison bit along with the data to the CPU. For Intel® architecture, when the CPU detects this poison bit, it sends a processor interrupt signal to the operating system to notify it of this error. This signal is called a Machine Check Exception (MCE). The operating system can then examine the uncorrectable memory error, determine if the software can recover, and perform recovery actions via an MCE handler. Typically, uncorrectable errors fall into three categories:

- Uncorrectable errors that may have corrupted the state of the CPU and require a system reset.

- Only uncorrectable errors that can be recovered by software can be handled during runtime.
- Uncorrectable errors that require no action.

Operating system vendors handle this uncorrectable error notification in different ways, but some common elements exist for all of them.

Using Linux as an example, when the operating system receives a processor interrupt for an uncorrectable error, it proceeds to offline the page of memory where the uncorrectable error occurred and add the error to a list of areas containing known uncorrectable errors. This list of known uncorrectable errors is called the bad block list. Linux will also mark the page that contains the uncorrectable error to be cleared when the page is recycled for use by another application.

The PMDK libraries automatically check the list of pages with uncorrectable errors in the operating system and prevent an application from opening a persistent memory pool if it contains errors. If an offline page of memory is in use by an application, Linux attemps to kill it using the `SIGBUS` mechanism.

At this point, the application developer can decide what to do with this error notification. The simplest way for you to handle uncorrectable errors is to let the application die when it gets a `SIGBUS` So you do not need to write the complicated logic of handling a `SIGBUS` at runtime. Instead, on restart, the application can use PMDK to detect that the persistent memory pool contains errors and repair the data during application initialization. For many applications, this repair can be as simple as reverting to a backup error-free copy of the data.

Figure 17-1 shows a simplified sequence of how Linux can handle an uncorrectable (but not fatal) error that was consumed by an application.

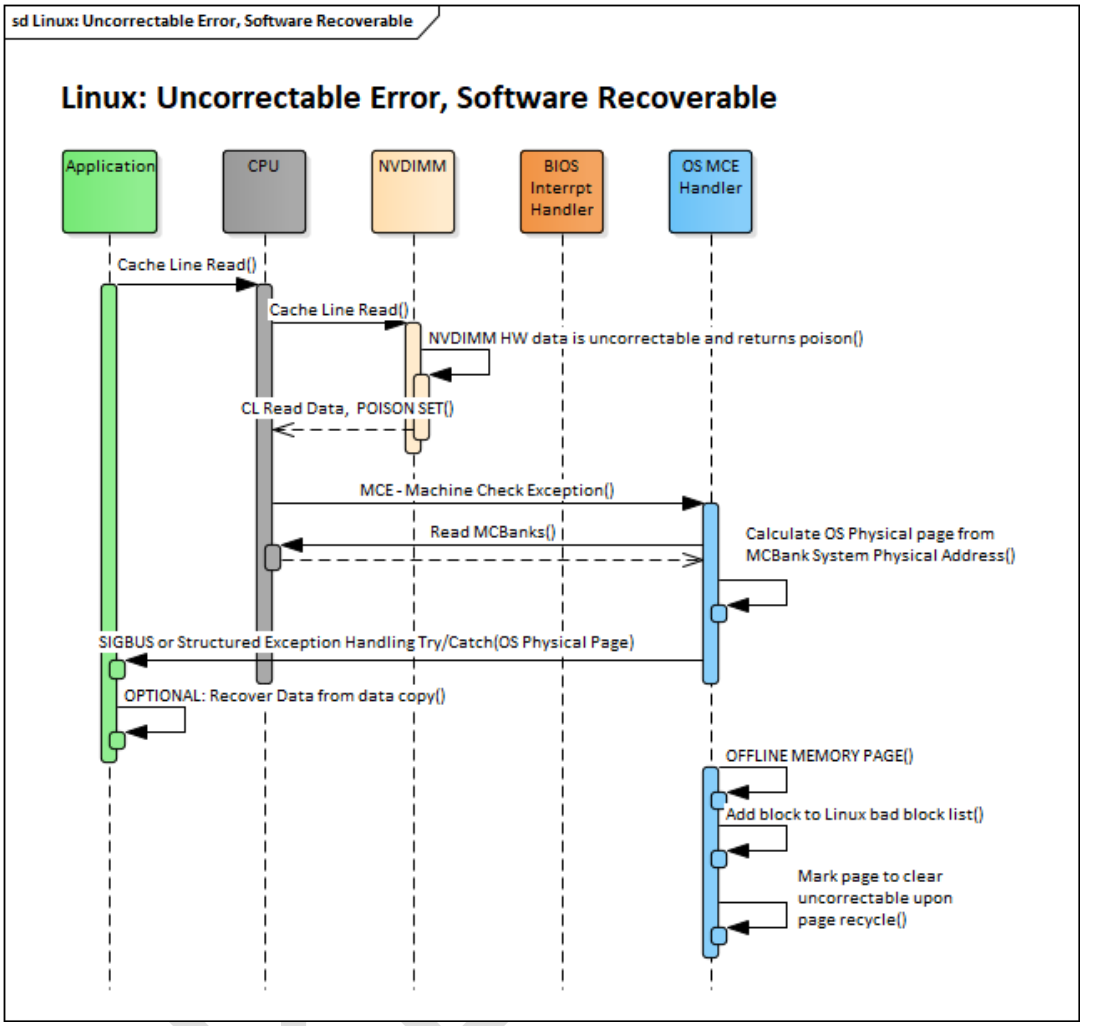

Figure 17-1. Linux consumed uncorrectable error-handling sequence.

Unconsumed Uncorrectable Error Handling

RAS features were defined to inform software of uncorrectable errors that have been discovered on the persistent memory media but have not yet been consumed by software. The goal of this feature is to allow the operating system to opportunistically offline or clear pages with known uncorrectable errors before they can be used by an application. If the address of the uncorrectable error is already in use by an application, the operating system may also choose to notify it of the unconsumed uncorrectable error, or wait until the application consumes the error. The operating system may choose to wait on the chance that the application never tries to access

the affected page and later return the page to the operating system for recycling. At this time, the operating system would clear or offline the uncorrectable error.

Unconsumed uncorrectable error handling may be implemented differently on different vendor platforms, but at the core, there will always be a mechanism to discover the unconsumed uncorrectable error, a mechanism to signal the operating system of an unconsumed uncorrectable error, and a mechanism for the operating system to query information about the unconsumed uncorrectable error. As shown in Figure 17-2, these three mechanisms work together to proactively keep the operating system informed of all discovered uncorrectable errors during runtime.

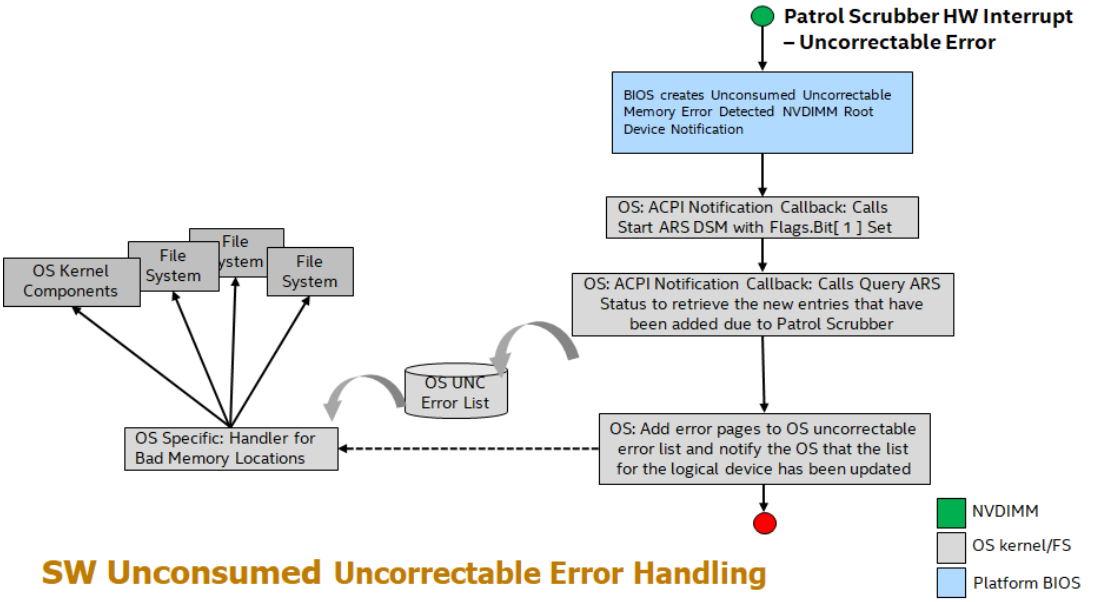

Figure 17-2. Unconsumed uncorrectable error handling.

Patrol Scrub

Patrol scrub (also known as memory scrubbing) is a longstanding RAS feature for volatile memory that can also be extended to persistent memory. It is an excellent example of how a platform can discover uncorrectable errors in the background during normal operation.

Patrol scrubbing is done using a hardware engine, on either the platform or on the memory device, which generates requests to memory addresses on the memory device. The engine generates memory requests at a predefined frequency. Given enough time, it will eventually access every memory address. The frequency in which

patrol scrub generates requests produces no noticeable impact on the memory device's quality of service.

By generating read requests to memory addresses, the patrol scrubber allows the hardware an opportunity to run ECC on a memory address and correct any correctable errors before they can become uncorrectable errors. Optionally, if an uncorrectable error is discovered, the patrol scrubber can trigger a hardware interrupt and notify the software layer of its memory address.

Unconsumed Uncorrectable Memory-Error Persistent Memory Root-Device Notification

The ACPI specification describes a method for hardware to notify software of unconsumed uncorrectable errors called the Unconsumed Uncorrectable Memory-Error Persistent Memory Root-Device Notification. Using the ACPI defined framework, the operating system can subscribe to be notified by the platform whenever an uncorrectable memory error is detected. It is the platform's responsibility to receive notification from persistent memory devices that an uncorrectable error has been detected and take appropriate action to generate a persistent memory root-device notification. Upon receipt of root-device notification, the operating system can then use existing ACPI methods, such as Address Range Scrub (ARS), to discover the address of the newly created uncorrectable memory error and take appropriate actions.

Address Range Scrub

ARS is a device-specific method (_DSM) defined in the ACPI specification. Privileged software can call an ACPI _DSM such as ARS at runtime to retrieve or scan for the locations of uncorrectable memory errors for all persistent memory in the platform. Because ARS is implemented by the platform, each vendor may implement some of the functionality differently.

An ARS accepts a given system address range from the caller and, like patrol scrub, inspects each memory address in that range for memory errors. When ARS completes, the caller is given a list of memory addresses in the given range that contains memory errors. Inspection of each memory address may be handled by persistent memory hardware or by the platform itself. Unlike a patrol scrub, ARS inspects each memory address at a very high frequency. This increased frequency of the scrub may impact the quality of service for the persistent memory hardware. Thus,

ARS can optionally be invoked by the caller to return the results of the previous ARS, sometimes referred to as a short ARS.

Traditionally, the operating system executes ARS in one of two ways to obtain the addresses of uncorrectable errors after a boot. Either a full scan is executed on all the available persistent memory during system boot, or after an unconsumed uncorrectable memory error root-device notification is received. In both instances, the intent is to discover these addresses before they are consumed by applications.

Operating systems will compare the list of uncorrectable errors returned by ARS to their persistent list of uncorrectable errors. If new errors are detected, the list is updated. This list is intended to be consumed by higher-level software, such as the PMDK libraries.

Clearing Uncorrectable Errors

Uncorrectable errors for persistent memory will survive power loss and may require special handling to clear corrupted data from the memory address. When an uncorrectable error is cleared, the data at the requested memory address is modified and the error is cleared. Because hardware cannot silently modify application data, clearing uncorrectable errors is the software's responsibility. Clearing uncorrectable errors is optional, and some operating systems may choose to only offline memory pages that contain memory errors instead of recycling memory pages that contain uncorrectable errors. In some operating systems, privileged applications may have access to clear uncorrectable errors. Nevertheless, an operating system is not required to provide this access.

The ACPI specification defines a Clear Uncorrectable Error DSM for operating systems to instruct the platform to clear the uncorrectable errors. While persistent memory programming is byte-addressable, clearing uncorrectable errors is not. Different vendor implementations of persistent memory may specify the alignment and size of the memory unit that is to be cleared by a Clear Uncorrectable Error. Any internal platform or operating system list of memory errors should also be updated upon successful executing of the Clear Uncorrectable Error DSM command.

Device Health

System administrators may wish to act and mitigate any device health issues before they begin to affect the availability of applications using persistent memory. To that

end, operating systems or management applications will want to discover an accurate picture of persistent-memory device health to correctly determine the reliability of the persistent memory. The ACPI specification defines a few vendor-agnostic health discovery methods, but many vendors choose to implement additional persistent-memory device methods for attributes that are not covered by the vendor-agnostic methods. Many of these vendor-specific health discovery methods are implemented as an ACPI Device Specific Method (_DSM). Applications should be aware of degradation to the quality of service if they call ACPI methods directly, since some platform implementations may impact memory traffic when ACPI methods are invoked. Avoid excessive polling of device health methods when possible.

On Linux, the `ndctl` utility can be used to query the device health of persistent memory modules. Listing 17-1 shows an example output of an Intel® Optane™ DC persistent memory module.

Listing 17-1. Using ndctl to query the health of persistent memory modules.

```
$ sudo ndctl list -DH -d nmem1
[
  {
    "dev":"nmem1",
    "id":"8089-a2-1837-00000bb3",
    "handle":17,
    "phys_id":44,
    "security":"disabled",
    "health":{
      "health_state":"ok",
      "temperature_celsius":30.0,
      "controller_temperature_celsius":30.0,
      "spares_percentage":100,
      "alarm_temperature":false,
      "alarm_controller_temperature":false,
      "alarm_spares":false,
      "alarm_enabled_media_temperature":false,
      "alarm_enabled_ctrl_temperature":false,
      "alarm_enabled_spares":false,
      "shutdown_state":"clean",
      "shutdown_count":1
    }
  }
]
```

Conveniently, `ndctl` also provides a monitoring command and daemon to continually monitor the health of the systems' persistent memory modules. For a list of all the available options, refer to the `ndctl-monitor(1)` man page. Examples for using this monitor method include:

Example 1: Run a monitor as a daemon to monitor DIMMs on bus "nfit_test.1,"

```
$ sudo ndctl monitor --bus=nfit_test.1 --daemon
```

Example 2: Run a monitor as a one-shot command and output the notifications to /var/log/ndctl.log.

```
$ sudo ndctl monitor --log=/var/log/ndctl.log
```

Example 3: Run a monitor daemon as a system service.

```
$ sudo systemctl start ndctl-monitor.service
```

You can obtain similar information using the persistent memory device-specific utility. For example, you can use the `ipmctl` utility on Linux and Windows* to obtain hardware-level data similar to that shown by `ndctl`. Listing 17-2 shows health information for `DIMMID 0x0001` (`nmem1` equivalent in `ndctl` terms).

Listing 17-2. Health information for `DIMMID 0x0001`.

```
$ sudo ipmctl show -sensor -dimm 0x0001

 DimmID | Type                          | CurrentValue
==========================================================
 0x0001 | Health                        | Healthy
 0x0001 | MediaTemperature              | 30C
 0x0001 | ControllerTemperature         | 31C
 0x0001 | PercentageRemaining           | 100%
 0x0001 | LatchedDirtyShutdownCount     | 1
 0x0001 | PowerOnTime                   | 27311231s
 0x0001 | UpTime                        | 6231933s
```

```
 0x0001 | PowerCycles                   | 170
 0x0001 | FwErrorCount                  | 8
 0x0001 | UnlatchedDirtyShutdownCount | 107
```

ACPI Defined Health Functions (_NCH, _NBS)

The ACPI specification includes two vendor-agnostic methods for operating systems and management software to call for determining the health of a persistent memory device.

Get NVDIMM Current Health Information (_NCH) can be called by the operating systems at boot time to get the current health of the persistent memory device and take appropriate action. The values reported by _NCH can change during runtime and should be monitored for changes. _NCH contains health information that shows if:

- The persistent memory requires maintenance,
- the persistent memory device performance is degraded,
- the operating system can assume write persistency loss on subsequent power events, and
- the operating system can assume all data will be lost on subsequent power events.

Get NVDIMM Boot Status (_NBS) allows operating systems a vendor-agnostic method to discover persistent-memory health status that does not change during runtime. The most significant attribute reported by _NBS is Data Loss Count (DLC). Data Loss Count is expected to be used by applications and operating systems to help identify the rare case where a persistent memory dirty shutdown has occurred. See “Unsafe/Dirty Shutdown” later in this chapter for more information how to properly use this attribute.

Vendor Specific Device Health (_DSMs)

Many vendors may want to add further health attributes beyond what exists in _NBS and _NCH. Vendors are free to design their own ACPI persistent memory Device Specific Methods (_DSM) to be called by the operating system and privileged applications. Although vendors implement persistent-memory health discovery differently, a few common health attributes that are likely to exist to determine if a

persistent memory device requires service. These health attributes may include information such as an overall health summary of the persistent memory, current persistent memory temperature, persistent media error counts, and total device lifetime utilization. Many operating systems, such as Linux, include support to retrieve and report the vendor-unique health statistics through tools such as `ndctl`. The Intel® persistent memory _DSM interface document can be found under the "Related Specification" section of https://docs.pmem.io/.

ACPI NFIT Health Event Notification

Due to the potential loss of quality of service, operating systems and privileged applications may not want to actively poll persistent memory devices to retrieve device health. Thus, the ACPI specification has defined a passive notification method to allow the persistent memory device to notify when a significant change in device health has occurred. Persistent-memory device vendors and platform BIOS vendors decide which device health changes are significant enough to trigger an NVDIMM Firmware Interface Table (NFIT) health event notification. Upon receipt of an NFIT health event, a notification to the operating system is expected to call an _NCH or a _DSM attached to the persistent memory device and take appropriate action based on the data returned.

Unsafe/Dirty Shutdown

An unsafe or dirty shutdown on persistent memory means the persistent memory device power-down sequence or platform power-down sequence may have failed to write all inflight data from the system's persistence domain to persistent media. (Chapter 2 describes persistence domains.) A dirty shutdown is expected to be a very rare event, but they can happen due to a variety of reasons such as physical hardware issues, power spikes, thermal events, and so on.

A persistent memory device does not know if any application data was lost as a result of the incomplete power-down sequence. It can only detect if a series of events occurred in which data may have been lost. In the best-case scenario, there might not have been any applications that were in the process of writing data when the dirty shutdown occurred.

The RAS mechanism described here requires the platform BIOS and persistent memory vendor to maintain a persistent rolling counter that is incremented anytime a

dirty shutdown is detected. The ACPI Specification refers to such a mechanism as the Data Loss Count (DLC) that can be returned as part of the Get NVDIMM Boot Status(_NBS) persistent memory device method.

Referring to the output from `ndctl` in Listing 17-1, the "`shutdown_count`" is reported in the health information. Similarly, the output from `ipmctl` in Listing 17-2 reports "`LatchedDirtyShutdownCount`" as the dirty shutdown counter. For both outputs, a value of 1 means no issues were detected.

Application Utilization of Data Loss Count (DLC)

Applications may want to use the DLC counter provided by _NBS to detect if possible data loss occurred while saving data from the system's persistence domain to the persistent media. If such a loss can be detected, applications can perform data recovery or rollback using application-specific features.

The application's responsibilities and possible implementation suggestions for applications are outlined as follows:

1. Application first creates its initial metadata and stores it in a persistent memory file:
    a. Application retrieves current DLC via operating system-specific means for the physical persistent memory that make up the logical volume the applications metadata resides on.
    b. Application calculates the current Logical Data Loss Count (LDLC) as the sum of the DLC for all physical persistent memory that make up the logical volume the applications metadata resides on.
    c. Application stores the current LDLC in its metadata file and ensures the update of the LDLC has been flushed to the system's persistence domain. This is done by using a flush that forces the write data all the way to the persistent memory power fail safe domain. (Chapter 2 contains more information about flushing data to the persistence domain.)

   d. Application determines GUID or UUID for the logical volume the applications metadata resides on, stores this in its metadata file, and ensures the update of the GUID/UUID to the persistence domain. This is used by the application to later identify if the metadata file has been moved to another logical volume, where the current DLC is no longer valid.
   e. Application creates and sets a “clean” flag in its metadata file and ensures the update of the clean flag to the persistence domain. This is used by the application to determine if the application was actively writing data to persistence during dirty shutdown.
2. Every time the application runs and retrieves its metadata from persistent memory:
   a. Application checks the GUID/UUID saved in its metadata with the current UUID for the logical volume the applications metadata resides on. If they match, then the LDLC is describing the same logical volume the app was using. If they do not match, then the DLC is for some other logical volume and no longer applies. The application decides how to handle this.
   b. Application calculates the current LDLC as the sum of the DLC for all physical persistent memory the application’s metadata resides on.
   c. Application compares the current LDLC calculated with the saved LDLC retrieved from its metadata.
   d. If the current LDLC does not match the saved LDLC, then one or more persistent memory have detected a dirty shutdown and possible data loss. If they do match, no further action is required by the application.
   e. Application checks the status of the saved “clean” flag in its metadata; if the clean flag is NOT set, this application was writing at the time of the shutdown failure.
   f. If the clean flag is NOT set, perform software data recovery or rollback using application-specific functionality.

g. Application stores the new current LDLC in its metadata file and ensures the update of the count has been flushed to the system's persistence domain. This may require unsetting the clean flag if it was previously set.
h. Application sets the clean flag in its metadata file and ensures the update of the clean flag has been flushed to the persistence domain.

3. Every time the application will write to the file:
   a. Before the application writes data, it clears the "clean" flag in its metadata file and ensures the flag has been flushed to the persistence domain.
   b. Application writes data to its persistent memory space.
   c. After the application completes writing data, it sets the "clean" flag in its metadata file and ensures the flag has been flushed to the persistence domain.

PMDK libraries make these steps significantly easier and account for interleaving set configurations.

Summary

This chapter describes some of the RAS features that are available to persistent memory devices and that are relevant to persistent memory applications. It should have given you a deeper understanding of uncorrectable errors and how applications can respond to them, how operating systems can detect health status changes to improve the availability of applications, and how applications can best detect dirty shutdowns and use the data loss counter.