



White Paper:

Using Data Poisoning to Contain and Recover from Uncorrected Errors

Authors:

Drew Walton (drewwalton@microsoft.com) with extensive feedback from Vilas Sridharan (AMD), Dean Liberty (Arm), Meeta Saggi (Pure Storage), Charles Slayton (Cisco) and Yogesh Varma (Intel).



This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-sa/4.0/).

Executive Summary

This document is intended to provide an overview of data poisoning, a technique to handle hardware errors, and provide general requirements for generating poison data, logging poison-related errors and reacting to these errors. Correct poison handling is independent of CPU instruction architecture and this document is intended to be generic for all CPU architectures and IO devices.

Compliance with OCP Tenets

A full explanation of the OCP core tenets can be seen [here](#).

Openness

This document is intended to clarify how data poisoning works. It is intended to be an open reference that third parties can use to ensure they implement data poisoning correctly.

Efficiency

By clarifying how computer systems should implement data poisoning, this document will help reduce instances of data corruption and help users diagnose related failures more accurately. These improvements reduce wastes and improve data center efficiency.

Scale

This contribution, by describing best practices for handling data poisoning, enables server fleet operators to scale their workloads, handling uncorrected errors more gracefully and enabling automated responses to errors that cause poisoning.

Sustainability

This contribution helps data center operators understand how to mitigate errors that cause data poisoning, showing how to avoid data corruption and showing how to diagnose the cause of the poison. This improves the efficiency of computing and helps prevent replacement of parts that log an error but have not failed.



Introduction: What is data poisoning?

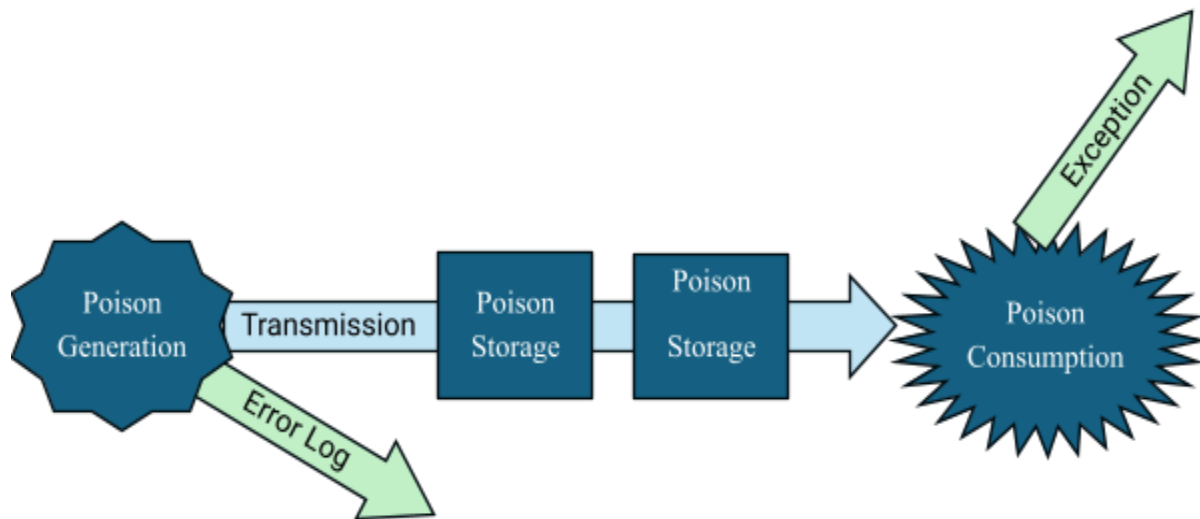
Data poisoning is an error containment tool used when hardware detects uncorrectably corrupted data. Hardware often uses parity checks or Error Correcting Codes (ECC) to detect and correct corrupted data. Only data that cannot be corrected needs to be poisoned. The sources of corruption are out of scope of this paper and include, but are not limited to, high energy radiation and other hardware faults.

When data poisoning is used, hardware that detects uncorrectably corrupted data marks the data as bad (“poison”) so that entities that attempt to consume the data can detect that the data is bad and not use it. The name “poison” was chosen for this error containment mechanism because it reflects that only entities that try to use poisoned data are affected by poison. Poison is frequently used in SoCs that support memory ECC, but poison can be generated in many other places, not just memory.

- Poison is generated when data is discovered to be corrupt, the corruption cannot be corrected, and the scope of corruption is known
 - Running processes are functionally unaffected at the instant poison is generated
 - Software is only affected when it attempts to consume poisoned data
 - Poisoning applies only to data, not metadata such as tags; if something like a packet header or cache tag is corrupted, we no longer know what OS physical address is corrupted and therefore do not know the scope of the corruption.
- Poison is typically an attribute of a granule of tracked memory addresses. That granule is often implemented as a cache-line.
 - Cache lines are not just in CPU caches, memory is accessed in cache line increments, usually 64 bytes.
- A poison indicator is stored with the cache line
 - In caches
 - ECC-protected memory has a way to store a poison signature
- Poison travels with the data
 - Between memory controllers, caches, and other storage locations
 - Any link that transmits poisoned data must have a way to indicate that a transmission is poisoned
 - Using Poison TLPs in PCIe®
 - Using a poison indicator in coherent links between SoCs

- When an engine (such as a CPU or IO device) attempts to consume poisoned data, the engine:
 - Must prevent software from using the poisoned data, e.g. halting the engine before the instruction attempting to use poison retires
 - Must generate an interrupt/exception (such as a Machine Check Exception on x64 or abort (such as a Synchronous error abort on Arm) to notify management software (e.g., operating system or device driver)
 - Sends an interrupt/exception to be serviced by the management software
- When management software gets interrupts from an IO engine for poison, it can decide how to handle the error and whether to terminate the affected process or crash the system
- When a system is notified of poison data (e.g., due to an exception) it must ensure that all engines are prevented from using the poisoned data (including IO devices)
 - For example, even when OS software is interrupted by an exception due to attempted poison consumption, IO device DMA operations may continue and steps must be taken to avoid them consuming corrupted data
 - This will be elaborated further in section IO Error Containment below
- If a system is running virtual machines in a hypervisor, attempts to consume poison will first trigger an interrupt/exception to the hypervisor which can decide how to handle the error, including triggering an interrupt/exception to a VM.
- When poison is transmitted to, and written into memory or a cache, no error is logged but a poison signature is stored with the cache line, such as in the ECC bits
 - The poison was generated elsewhere
 - But we need to remember that the data was corrupted so we can block consumption
- If a poison signature is written to a memory location, the memory controller will transmit a poison indication to anything that uses that memory location but it will not signal a second poison generation event

A simplified example of data poison handling is shown below:



In the diagram above, poison is generated when corruption is detected. This is usually a failure like an uncorrected ECC error, bad parity or a bad CRC on a link. The poison signature is transmitted from the entity that generated it to a consumer. The consumer must not be allowed to complete an operation that uses poisoned data since the poisoned data is known to be corrupt.

Poison generation might not be immediately followed by poison consumption. If a CPU prefetches a cache line that generates poison when it is read from memory, that need not be considered consumption because the speculatively prefetched cache line might be evicted from the cache without ever being used by software.

Poison generation and consumption errors should be treated as separate events in software. Some CPU architectures may have separate interrupts for poison generation and consumption. Sometimes software might request data that has been corrupted and see a poison generation and poison consumption event in close time proximity but it is not safe to assume that will always be the case. It is also possible that data could be poisoned and never consumed.

Poison Example 1: Memory Uncorrected Error

The easiest way to see how data poisoning is used is to walk through an example. Let's look at an uncorrected memory error on a SoC that supports memory ECC.

1. An application reads a memory address that is not in the CPU's cache
2. The CPU requests the data from the appropriate memory controller
3. The memory controller reads memory, including ECC data

4. The memory controller's calculated ECC does not match the ECC that was read, and the error is not correctable
5. The memory controller logs a deferred, uncorrected memory error
 - a. Includes the memory address that was affected
 - b. This is the poison generation error log
 - c. See section Logging Poison Generation for more details
6. The memory controller marks the data as poison in its reply
7. The memory controller also writes a poison signature into memory
 - a. The memory controller should not report new errors when it reads a cache line with a poison signature since reporting errors for previously poisoned memory could cause a storm of uncorrected errors.
 - b. If the poison signature is successfully stored in memory, we only get one poison generation event, even for future reads of the poisoned address
8. The logic connecting the memory controller to the CPU transmits the poison along with the data
9. The CPU stores the poisoned data in the cache
 - a. No error is generated since the poison data is not yet consumed by a process
10. When the application process' read instruction is executed, the CPU takes a synchronous exception that halt process execution at the instruction that would cause poison to be consumed
11. The CPU's interrupt is serviced by an OS hardware error handling Interrupt Service Routine (ISR)
12. The interrupt service routine determines whether to kill the process, kill the system, or take some other action

It is important to note that the scope of the corrupted data was known: the cache line that the memory controller read was corrupted.

There are 3 key things that happened in this flow:

1. Poison was generated by the memory controller for the uncorrected memory error
 - a. The memory controller knows the address of the failure
 - b. The memory controller knows the scope of the corruption (cache line or other poisoning granule)
 - c. The memory controller logs the uncorrected ECC error (tell us the physical location of the hardware fault)
2. The poison marker is transmitted with the data from the memory controller to the CPU core

- a. The protocol that is used for the transmission must maintain a poison indicator
 - b. The transmission or movement of poisoned data is generally not logged as an error
- 3. The CPU attempted to consume the poison and triggered a synchronous interrupt to the OS
 - a. This action prevents the process from using corrupted data
 - b. Poison only directly impacts the consumer
 - c. The poison consumption should generate an error log because the process was affected
 - d. A repair action **should not** be driven by the consumption event
 - i. Sometimes people assume that memory caused the poison to be generated and replace the DIMM that holds the OS physical address. This assumption may not be correct.
 - ii. There are many possible sources of poison other than memory!
 - e. **Repair actions should be driven by the poison generation event**
 - i. There are many sources of poison
 - ii. The entity that originally detected the corruption must log the error
 - iii. Use the poison generation error log to determine what failed
 - iv. Possible sources of poison are covered later in this document

Poison Example 2: CPU Last Level Cache Uncorrected Error

CPU caches can also have uncorrected ECC errors. These errors can be immediately consumed or written back to memory and consumed sometime later. These examples will discuss errors in cache data where we still know the state of the cache line and what address the data in the cache came from. Poison can only be used for cache data and will not work when cache tags are corrupted since we no longer know the address of the failure. When the tags are corrupted, we need a more severe containment mechanism as described in the section titled, “Errors where the scope of corruption is not known.”

The case where a CPU core fetches a cache line from its own cache, and the cache has an uncorrectable ECC error, is straightforward. The poison is generated in the cache, an uncorrected error is logged for the poison generation, and the CPU core that consumes the poison takes a synchronous interrupt to the OS.

In some cases, an uncorrected error in a cache can be fixed by reading the data from a healthy copy in another place. For example, if the uncorrected cache error is in a

write-through cache, the CPU can read the correct data from the next level down in the cache hierarchy.

The case where the poisoned cache line is written back to memory is much more interesting and we will walk through the example in more detail:

1. The last level cache has an error that alters the data stored for a cache line
2. The affected cache line is evicted from the cache
 - a. The cache controller reads the corrupted cache line
 - b. The cache controller detects an uncorrected ECC error
 - c. The cache controller logs the uncorrected ECC error and generates poison
3. The poisoned cache line is sent to the memory controller
4. The memory controller writes a poison signature to memory at the affected address
 - a. This might be an ECC value, a bit, and/or a pattern written to data & ECC bits
 - b. The memory controller does not log an error (it did not generate poison)
 - c. The poison remains stored in memory until it is cleared, usually by a reset
5. Sometime later, another CPU core fetches the address that was poisoned
6. The memory controller reads the address, and forwards poison to the requesters
 - a. The memory controller sees that poison was previously written to memory
 - b. This transmission is not poison generation since no new corruption was detected
 - c. Nothing is logged in the memory controller. For troubleshooting purposes, this is important. Otherwise, memory will be falsely accused.
7. The requesting CPU gets the poison from memory controller, writes it to its cache
8. The core gets the poison cache line from the cache
9. Once the core is certain that software is about to consume poison, it triggers a synchronous interrupt to OS
 - a. Consumption is usually at the instant that the affected instruction is retired

Even though the poison is passed between memory and caches, only 2 things need to be logged: the poison generation when the cache ECC error is detected and the attempted poison consumption when software is interrupted. In this example, system memory contained a poisoned cache line, but the physical memory devices were not the source of the failure, and the memory controller did not log an error.

Poison Example 3: Upstream Poison from a PCIe device

Poison can also be generated by PCIe devices. When PCIe devices generate poison they or the upstream port might create error logs that contain the Bus, Device and Function of the PCIe device that generated poison. The Bus, Device and Function are useful for helping us figure out which FRU needs to be serviced. This is complicated by drivers not having a generic way to report the failure. The PCIe root port can log Poison TLP Received when a device sends poison upstream.

In this scenario, we will consider DMA from a PCIe device to the host.

1. A PCIe device reads its on-board memory and detects an uncorrectable ECC error
 - a. This is commonly High Bandwidth Memory (HBM)
 - b. But there are other possible components that could generate poison
2. The device's memory controller poisons the transaction and logs the poison generation
3. The memory controller writes the poison signature to the memory so that the poison generation is only reported once
4. The transaction is sent to the device's PCIe interface
5. The PCIe interface marks the transaction as a Poison TLP
6. The Root Port creates a poison transaction inside the SoC
7. Since we are DMA'ing into memory the address of this transaction causes it to be directed to the memory controller
8. The memory controller on the host writes a poison signature to DMA's target memory address
9. Up to this point, the host is tracking the poison but not taking any action on it
10. Consumption might happen at a later, indeterminate time
11. If a CPU on the host tries to use the poisoned data, the memory controller will send it poison and it will take a synchronous interrupt
 - a. To the OS, this looks very similar to an uncorrected memory error
 - b. But the source of the poison is the PCIe device, not main memory!

Common Sources of Poison

There are many possible sources of poison in modern servers. Any silicon structure that can detect corruption can signal poison. The largest amount of silicon in most servers is memory and memory tends to be the largest source of poison. But there are other significant sources of poison that should be considered in designing an error containment solution.

Poison Sources:

- Main memory, such as DRAM
 - When an application fetches something from memory
 - When the patrol scrubber scrubs a location in memory
- CPU Caches
- IO devices
 - Including DRAM on the device such as LPDDR, HBM and GDDR
- Other structures in silicon that can detect errors

The next few sections of this document focus on containing corrupt data and later sections cover how to determine the source of poison. For now, assume that attempted poison consumption means a failure somewhere in the system, but do not assume that the poison was from a particular source.

Software Responses to Poisoned Data

A poisoned cache line does not contain correct data. Any operation that attempts to use a poisoned cache line must fail in order to prevent the operation from using bad data. Note that this includes both software and IO operations.

Software Poison Containment

When software, running on a CPU, attempts to use poisoned data, it will trigger an interrupt to the OS which can:

1. Reload application data (for read-only segments like “.text”)
2. Send an error signal to the application in case it can recover from losing the cache line (especially useful for large, in-memory databases)
3. Terminate the application
4. Terminate the VM (if in a VM)
5. Terminate the OS/hypervisor

Note that the list above is in order of increasing blast radius.

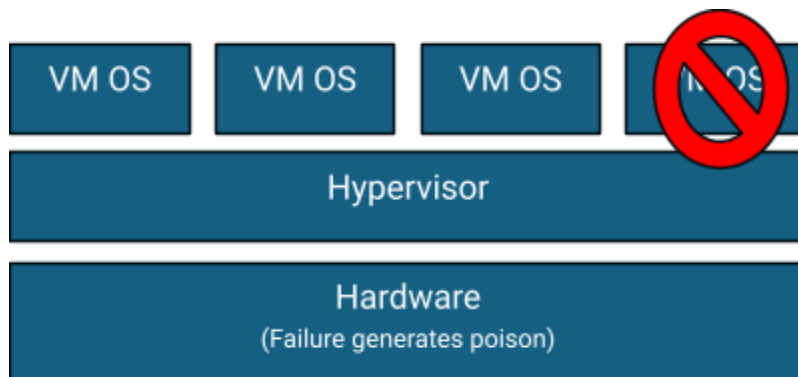
Terminating the OS

This is the simplest response to poison: crash the OS. It is the simplest alternative, since it treats poison consumption like any other uncorrected error, but also has the highest impact on users. It is desirable to avoid crashing the OS for poison consumption, but it may be necessary in some cases, such as when the poison is in a critical kernel, non-fault-tolerant data structure. Fortunately, the amount of memory used for critical kernel data structures is small relative to the amount of memory used by

applications; we can use one of the alternatives listed below to reduce the blast radius of poison for non-kernel data structures.

Terminating the VM

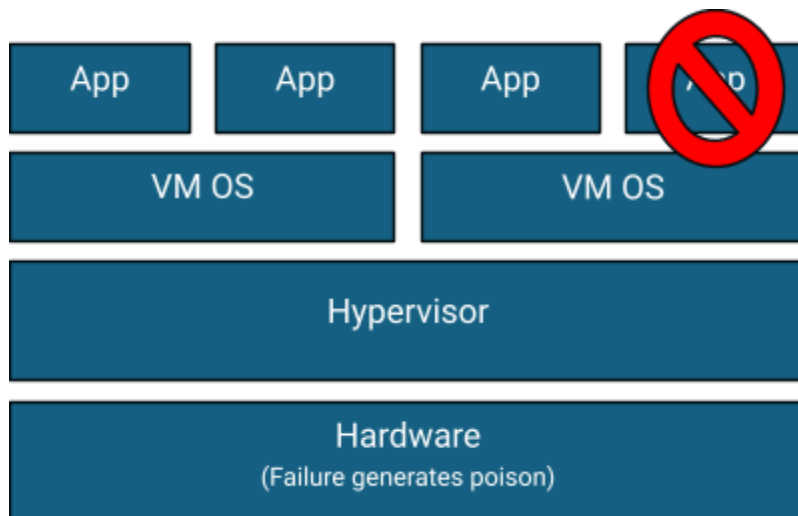
If workloads are running in Virtual Machines, the poison consumption event will trigger an interrupt to the hypervisor. The hypervisor can determine which virtual machine is using the affected address and terminate that virtual machine.



Other virtual machines can continue running though it may be possible and prudent, if faulty hardware is suspected, to trigger live migration and move them off the machine.

Crashing the application

It is also possible to kill only the application/process that consumes poison.



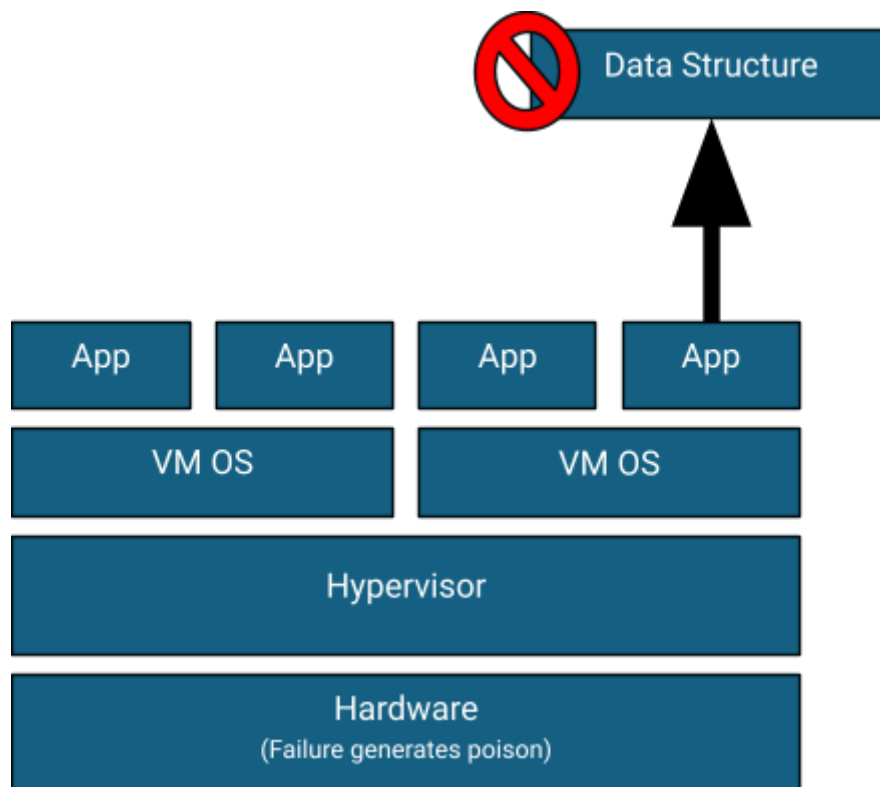
Killing the application reduces the blast radius of the failure but may be a problem for workloads that consist of multiple processes and do not correctly handle a process being killed. It may be prudent to verify that the workload can tolerate a process being killed. Many applications have daemons that monitor processes and restart any that die so this solution will work in many cases.

Reloading a read-only segment

If poison is detected in a read-only memory section, it may be possible for the OS to reload that section, remapping the addresses of the section so that they do not use the poisoned address. For example, Unix-derived OSs use the ELF (Executable and Linkable Format) standard for executable files. This file format puts code and read-only data in the .text and .rodata segments of the executable. If the OS detects poison in these sections of the executable, it may be possible to reload them from disk. The Microsoft Windows PE format also has read-only sections such as .rdata that might be reloaded from storage in a similar manner.

Signal the application so it can recover

It is also possible to signal the application when the failure occurs, and the application can determine if the affected data structure can be reconstructed or determine if the loss of the data structure can be tolerated.



Modern database applications can often tolerate the loss of rows in the database and replay transactions to recreate the missing data. Memory poison recovery is especially important for large, in-memory database applications.

In Linux, [the application is signaled using SIGBUS](#).

In Microsoft Windows®, the application is signaled via [Structured Exceptions](#).

If the application does not handle the signal, it crashes.

Mechanisms to Prevent Software Poison Consumption

The containment mechanisms listed in the previous section can reduce the blast radius of poison, but it is better to avoid attempted poison consumption altogether whenever possible.

Track poisoned memory and avoid it

If a particular memory OS physical address is known to contain poison, the OS can mark the affected memory page as being unusable so that future applications do not use the memory. Since uncorrected errors are relatively rare, poison is also rare, and it is probably better to err on the side of mapping out too many pages rather than too few. As mentioned elsewhere in this document, the memory may contain poison that was generated by another device such as a PCIe device and in these cases, the OS may be able to clear the poison and resume using the memory.

It is desirable to mark poisoned pages as unusable for both poison generation and poison consumption errors. When a memory patrol scrubber generates poison as a result of an uncorrected memory error, the source of the poison is known, and we know that the memory at the address that generated the poison should be avoided and thus marked as bad by the OS. The patrol scrubber may detect poison before any application on the computer and can sometimes allow the OS to respond to the poison earlier, possibly reducing the impact of the failure.

Likewise, when poison is consumed by software, we know that the memory at the affected address is poisoned and software may optionally mark the affected memory page as bad. In general, losing a 4K page is better than crashing.

Suggested flow:

1. OS realizes that a memory address is poisoned
 - a. From attempted consumption
 - b. From the patrol scrubber
2. OS marks the poisoned page as unusable
3. Error-handling software gets error logs and analyzes the error
4. The error-handling software attempts to:
 - a. Determine what generated the poison
 - b. Determine if a FRU should be replaced
 - c. If it is safe for the OS to clear the poison and allow the page to be used

This error-handling software is described in [OCP Fault Management Infrastructure Requirements ver 1.0](#). The section titled Clearing poison, below describes how poison might be cleared in memory.

It may also be possible to correlate the PCIe address sent to a device with host memory, allowing us to map out pages when an IO operation fails as a result of poison being sent to the IO device. For example, a PCIe device may DMA a poisoned cache line from memory and fail the affected operation. In such a case, it would be useful to mark the memory address as bad, assuming there is a way to either remap the address using the IOMMU or to tell the driver to stop using the address.

Marking memory pages as bad can present problems when large or huge pages are used and can cause memory fragmentation, but they are usually preferable to crashing.

Partial Memory Mirroring

Poison recovery as described in the section titled Software Poison Containment, above works when an application or an OS with poison-tolerant data structures tries to consume poison, but many OSs cannot tolerate a failure that poisoned their internal data structures. To avoid poison consumption on such OSs, we might use a hardware feature called partial memory mirroring.

Memory mirroring is a hardware feature that stores data two or more times in memory. When memory is read, the read is directed to one of the copies. If there is an error during the read, the hardware will read one of the other copies of the data. Memory mirroring improves memory reliability but is expensive since it halves the memory available to applications. In addition, there will likely be a performance impact since every write to memory is done in two places.

Partial memory mirroring only mirrors part of memory, reducing the amount of memory lost to mirroring. If an OS implements poison recovery for applications, but cannot recover from errors in data structures, it might use partial memory mirroring to store OS data structures that cannot tolerate an uncorrected error.

IO Error Containment

Preventing software from using poisoned data is critical for containment but insufficient. It is possible for poison to crash an OS while a PCIe device continues to DMA poisoned data from main memory to a network device or to storage; we need a way to prevent IO devices from using poisoned data.

Poison is transmitted to PCIe devices as Poison Transaction Layer Packets (PTLPs), but PCIe devices are inconsistent in the way they handle poison. This is because many

PCIe devices were originally derived from consumer products that do not support as much error detection as data center products. For example, some GPUs will use the data from a PTLP even though the data is known to be corrupted. It is important to test poison handling on PCIe devices and for us to keep track of which ones fail operations when they receive poison (correct behavior) and which ones use the data in the PTLP (incorrect behavior).

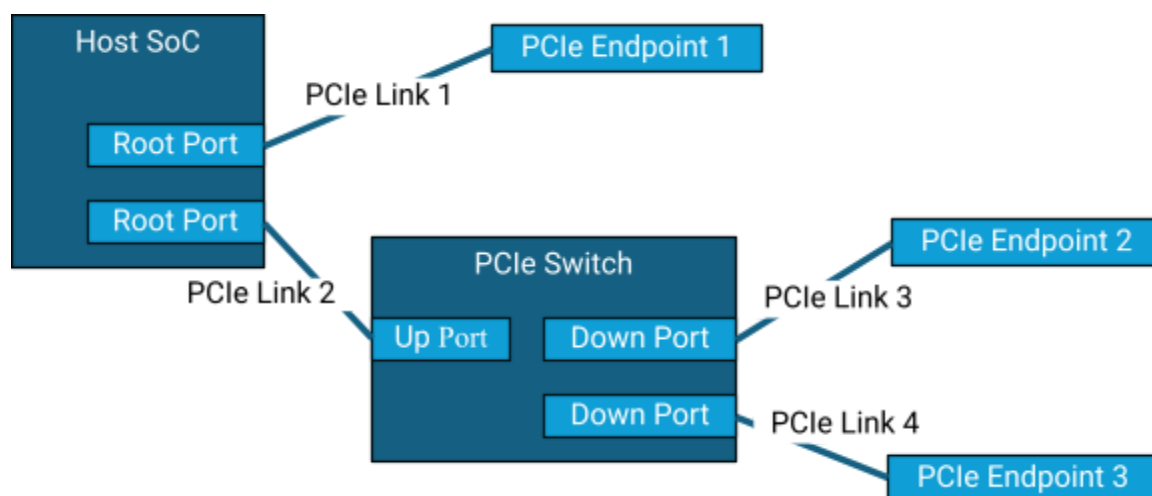
Silicon that is expected to be used by enterprise and cloud service providers should detect PTLPs and fail operations that receive them. This is not always the case today but is critical as data center workloads continue to scale up.

Options for handling poison propagating to a PCIe device

If the cloud service provider is confident that the PCIe device handles poison in a way that meets their requirement, for example, proper containment of corrupt data, they can allow Poison TLPs to reach the device.

When the device does not meet the CSP needs, the CSP may be able to take additional steps to ensure that their requirements are met. One option is to use a feature in the host's Root Port called Downstream Port Containment (DPC) which brings the PCIe link down for fatal errors. The PTLP can be promoted to a fatal error so that DPC prevents the poorly designed PCIe device from ever getting the PTLP. The PCIe link going down will cause the associated operation to fail.

When implementing DPC, it is useful to configure PCIe ports to trigger DPC on the furthest downstream port possible. To see why this is useful consider the following simple server with 3 PCIe endpoints, 2 of which are connected via a PCIe switch.



If there is an error in main memory and a PCIe device reads the failing address, it will cause the Root Port to send a PTLP on the upstream link. Assuming that all of the

PCIe devices do not support poison correctly, we will assume that all the PCIe links are configured to use DPC.

In the case of PCIe link 1, the PTLP triggers DPC and only the PCIe endpoint is affected. But if we look at PCIe link 2, bringing that link down for a PTLP will affect PCIe endpoints 2 and 3 even though the transaction was targeted to only one of them. It would be better to not configure DPC in the Root Port for PCIe link 2 and configure it in the downstream port for PCIe links 3 and 4. That way, only the endpoints that actually get the PTLP are affected. This is especially useful when there are multiple VMs in a machine that each use different PCIe links. Note that this requires PCIe switches that support DPC.

Host-OS/Drivers need to ensure they handle cases where a configuration read fails and returns all 1's such as when doing a read after DPC brings the link down as described in the implementation note titled "DATA VALUE OF ALL 1'S" from the PCIe 6.1 Specification, section 6.2.11, "Downstream Port Containment (DPC)". This requirement is included in prior versions of the specification and is sometimes overlooked, creating the possibility of corrupt data being used.

Sometimes Root Ports can be configured to send Unsupported Requests (URs) instead of PTLPs. URs have the advantage that they do not bring the link down but require PCIe IP and drivers that handle errors well. PCIe devices need to be tested to ensure they handle URs correctly. In many cases, a PCIe device that does not support PTLPs correctly is not likely to notice that it got a UR for a memory read and such devices will treat the UR as valid data. DPC is usually a more reliable way to prevent the consumption of poison.

Upstream PCIe Poison Handling

PCIe devices can generate poison. A common example is an uncorrected ECC error in HBM memory. To contain these errors, the PCIe device must either fail the operation and ensure that the poisoned address never escapes the device or send poison with the data that had the error. In some cases, GPUs that do not correctly support downstream poison will still send poison upstream. It is important to have poison enabled on the host machine. When the upstream PTLP is sent to the host, it will store poison in memory and trigger poison consumption handling when software tries to use the poisoned address.

Upstream poison from PCIe devices can be hard to diagnose since PCIe devices are very inconsistent in how they log the errors and notify the user. The Open Compute Project expects to address this in the RAS API team by having PCIe devices report

errors in CPER logs via the API and this document will be updated with details as the RAS API is published.

It is also prudent for the Root Port to log that it received a PTLP and log the bus, device and function of the source of the PTLP. Even though we don't normally want to log the transmission of poison, logging PTLPs and their source may be the only way to tell that a PCIe device generated poison; we need to be able to detect the failing FRU and replace it.

Clearing poison

Poison needs to be cleared, especially in cases where a device has written poisoned transactions to large ranges of memory. Poison can be cleared by a reboot. When the server boots, the firmware will test memory and then initialize the memory ECC. The act of initializing the ECC clears the poison since the poison signature is stored in memory ECC. In some cases, the poison is never consumed and is overwritten by a cache line write such as when an OS overwrites memory that it is allocating to a new process (assuming memory is zeroed out using atomic cache line writes that do not need to read-modify-write the cache line to zero it out).

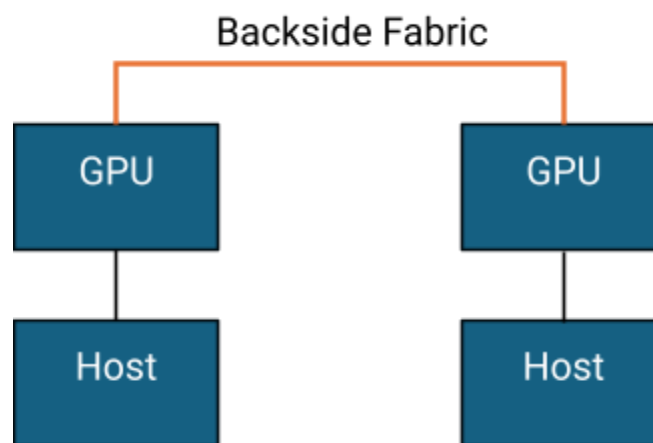
When the OS can recover from poison, it is possible for the number of poisoned cache lines to increase over time. In such cases, it would be ideal to have a way to clear poison from memory cache lines, especially in the case where the poison was not from a memory error. There are several ways to clear poison:

- Reading full cache line and overwriting it
 - CPU will read the cache line for ownership (assuming cache coherency)
 - If you can write the whole cache line with a single instruction there is no consumption and the poison will be overwritten
 - Sub-cache line writes require read-modify-write which is treated as consumption and does not clear poison
 - Can sometimes use SIMD registers to do a whole cache line write
 - Some CPU architectures may have instructions that zero a whole cache line
- Send a special command to the memory controller
 - The memory controller might have a way to overwrite a cache line and reset its ECC to a good state (no poison)

The key to clearing poison is to be sure that no process is using the poisoned cache line and then be able to set the ECC to a good state without triggering the interrupts used to prevent poison consumption.

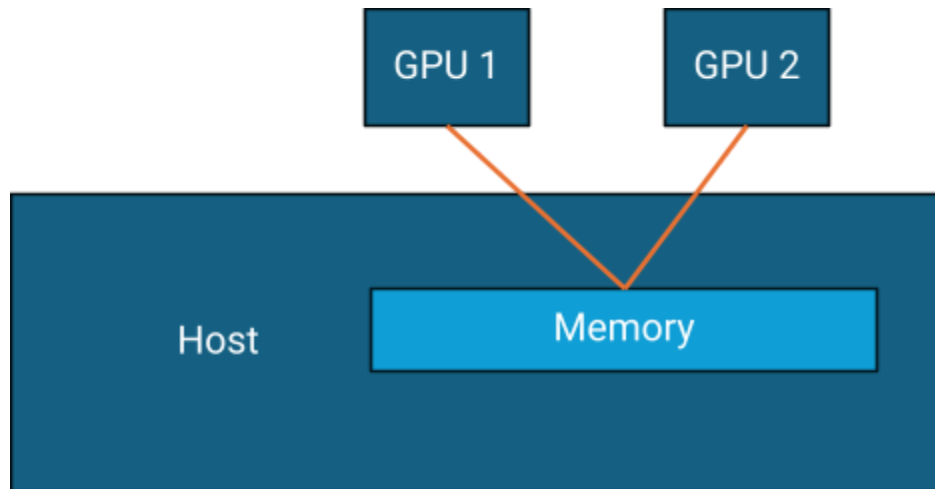
Other Error Containment Considerations

GPUs and other accelerators may have a backside fabric that passes data between GPUs. At the time of this writing, these fabrics rarely support poison correctly, though the Open Compute Project's GPU RAS team is requiring that error containment be supported on backside fabrics. GPUs should be tested to ensure that they fail operations affected by poison on their backside fabrics.



GPU Mem GPU corner case

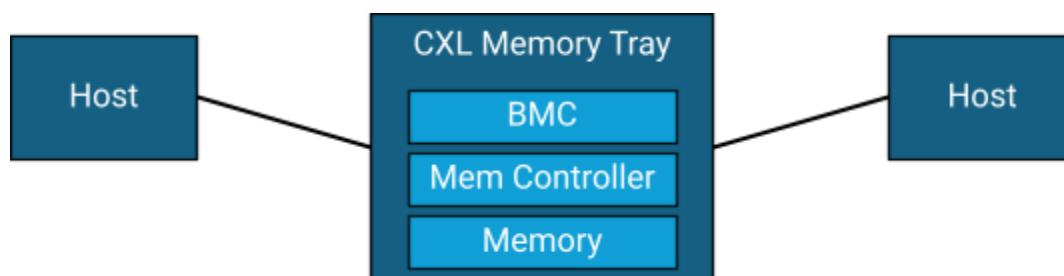
Another potential problem with GPUs that do not handle poison correctly is that the poison may not be detected for a long time. Consider the following system with 2 GPUs that are passing data between one another using the host's main memory over PCIe links, shown in orange in the diagram below.



If GPU 1 DMA's data into memory and that memory is failing, GPU 2 could be configured to read that poisoned memory with no CPU involvement. If the GPUs do not support poison and PCIe DPC is not configured, GPU 2 could be reading corrupted data for an indefinite period with no way to detect it. This is a worst-case silent data corruption scenario and underscores the need for GPUs to fail operations affected by PTLPs.

There are also many corner cases that need to be considered for memory that is attached to the host via CXL®. CXL.mem allows memory to be attached to a host via a CXL link. CXL.mem devices must be tested to ensure they handle poison correctly and must use Downstream Port Containment (DPC) if they do not. Sometimes CXL memory persists after the host is reset. This means that poison in the CXL memory will also persist. CXL memory controllers need a way for the host to clear poison and an OS using CXL memory will need to remember the addresses of poisoned cache lines in CXL memory between boots if the memory is not cleared on a reboot.

Shared memory between servers using CXL memory creates a lot of corner cases that need to be addressed to maintain containment and uptime.



The servers attached to the CXL memory must both support poison and have a way to tell the CXL memory controller when it is safe to clear memory and reinitialize the memory ECC. We strongly recommend that CXL memory controllers have hardware state machines that can clear a range of memory and reinitialize its ECC. Ideally the servers can send a command to the CXL memory BMC telling it to clear memory and the BMC firmware can trigger the state machine. There are also corner cases where memory in one host creates poison that is transferred to CXL memory and from there to the other host. These corner cases must be considered in shared memory designs.

Non-Volatile DIMMs (NVDIMMs) also require special consideration with poison handling. Like CXL.mem, the NVDIMM may not be cleared upon a reboot. We need to remember which pages were poisoned in the NVDIMM across reboots or clear the memory and reinitialize the memory ECC on a reboot or develop a way to clear the poison in the affected cache line. If poison is not successfully cleared or tracked, the host can reconsume the poison with every new boot. Ideally the host will be able to test the NVDIMM upon reboot and tell the OS to avoid a list of bad pages.

Logging Poison-Related Errors

Logging poison-related errors is critical to addressing the failure that caused poison and for understanding the impact of the failure to the user.

Logging Poison Generation

Poison generation is a primary error, that is an error that directly correlates to a silicon block that detected a problem. FRU replacements and RAS actions should be driven by the poison generation event. When poison is generated, we need to log:

- An uncorrected error that is not acted upon immediately (sometimes called a deferred error)
- The reason that poison was generated (e.g. an uncorrected ECC error in a particular memory array)
- The location of the error in the structure that had the failure (e.g. rank, bank, row, column)
- The system physical address of the poison where known (e.g. main memory)
 - Needed for the OS to map out bad pages
 - Needed to correlate a later poison consumption event with the failure that generated poison
- The PCIe local address when the error is on a PCIe link

- We still want the system address if it is possible to calculate it using the IOMMU so we can correlate it with consumption and map out the page

Logging Poison Transmission

Poison transmission is usually not very interesting and should generally not be logged. PCIe poison transmission may be an exception and may need to be logged.

- A DPC event blocks poison transmission and effectively prevents poison consumption
- If a PCIe device writes poison to the host, it may be useful to log that poison as if it was poison generation
 - This allows us to determine that a PCIe device is generating poison even in cases where the PCIe driver does not log the error correctly
 - The PTLP header should be logged since it may tell us what device and function caused the error.
 - There may be no other way to determine that a PCIe device generated poison

Logging Poison Consumption

Poison consumption indicates that the user has been impacted by the failure that originally caused poison to be generated. We need to handle poison consumption in software and need to log that it occurred so that we can understand the scope of the impact to the customer.

When a CPU attempts to consume poison and takes a synchronous interrupt, we should:

- Log an error indicating the CPU was interrupted for a poison consumption
- Log the system address of the error

Having a count of the poison consumption errors can be useful for understanding how much the user is being impacted by uncorrected errors.

The poison consumption error log can be correlated with the poison generation event using the system address.

Poison **consumption** error logs **should not** generally be used to drive part replacements. Even though memory is the most common source of poison, it is possible that the poison might have been generated in another structure or in an IO device. The poison **generation** error logs **should** be used to drive repair actions.

Techniques for Minimizing Impact of Errors where the scope of corruption is not known

Poison is only useful for errors where the scope of corruption is known. There are more severe errors such as a corrupted packet header or a packet timeout where the scope of the corruption is not known. Poison is not useful in these situations because we do not know what to poison. Even worse, these cases represent losses of memory coherency, and we need a more drastic error handling solution like using viral containment.

Here are some common causes of viral containment:

- Cache tag corruption
- Packet header corruption on CPU interconnect
- Coherent links going down
 - e.g. CPU socket interconnects or DDR channel, not PCIe
- Timeouts

With viral containment, the block that sees a problem goes into viral containment mode where it marks all transactions going out of it with a viral signature. When a block receives a viral packet, it goes viral and all its outgoing packets are marked as viral. Where poison requires consumption to drive a crash, viral containment spreads by contact. Any CPU or PCIe interface that sees a viral packet will shut down, containing the corruption caused by the loss of coherency. Viral mode, like poison, should stop IO like RDMA, disk accesses and network traffic, but is different in that once the root port goes viral, all future IO should be blocked. The one exception to this is for the BMC. The BMC is going to need to log the error that triggered viral containment and it should still be able to access the device that has gone viral.

When a SoC enters viral containment, the viral mode blocks IO to PCIe devices. The boot firmware on the SoC might be able to do some logging, but it is probably better to spinloop and log Out of Band. In the past, very sophisticated error handling firmware could crawl out of viral mode, systematically working outwards from the CPU, logging errors and reenabling functionality. At the end of the crawlout, the firmware would “plunge” the system by masking viral propagation, clearing the viral errors and waiting some multiple of the longest transaction timeout to ensure that any in-flight viral transactions reached their destination before reenabling viral mode. This firmware required extraordinary knowledge of the hardware design and was very difficult to write. Modern systems have service processors like the BMC that can log the viral error

without a complicated crawlout, but silicon designers must ensure that the BMC can read critical error logging registers when the system is in viral mode.

AMD has a similar feature called syncflood which provides similar containment with one key difference: syncflood stops forward progress, blocking code execution. With traditional viral containment, the CPU will take an interrupt that firmware running on the SoC can service as mentioned above. With syncflood, the CPU cores are wedged so that Out of Band (OOB) firmware can log the error and reset the machine to clear the syncflood. This is simple to implement and provides strong containment.

In some cases, viral containment may be hard to implement since it requires a way to mark a transaction as viral and requires that every functional block that can receive a viral packet to handle viral containment correctly. In such cases it may be easier to make blocks poison every transaction that passes through them. This is a useful shortcut but, in the future, we generally want the benefit of full viral mode.

Glossary

Error Containment

Preventing corrupted data from detected uncorrectable errors from being used by software and preventing the corrupted data from escaping via IO devices.

Error Correction Code (ECC)

Error Correction Codes store redundant data in a storage array and allow some errors to be in the data corrected and allow some types of corruption to be detected.

Poison

Data is marked as poisoned by silicon that can detect uncorrected errors in the data and knows what data is affected by the errors.

Primary Error

A primary error is an error where the source of the error is known. For example, an uncorrected ECC error in a cache is a primary error. Poison consumption is a secondary error; we know an error occurred but the silicon that logs poison consumption errors does not know where the poison was generated.

Silent Data Corruption



Silent Data Corruption occurs when silicon cannot detect corrupted data. For example, a severe error in memory may flip so many bits that the ECC aliases to the correct value even though the data is corrupt. Parity cannot detect 2 flipped bits. Silent data corruption also occurs when error containment is not implemented correctly, especially when poison handling is not correct and we lose track of which data is corrupted. The corrupted data escapes to a location where it may be used even though it is not valid.

References

[OCP Fault Management Infrastructure Requirements](#)

[hwpoison — The Linux Kernel documentation](#)

[BKDG for AMD Family 15h Models 70h-7Fh Processors](#) – shows how syncflood is used

License - Creative Commons

OCP encourages participants to share their proposals, specifications and designs with the community. This is to promote openness and encourage continuous and open feedback. It is important to remember that by providing feedback for any such documents, whether in written or verbal form, that the contributor or the contributor's organization grants OCP and its members irrevocable right to use this feedback for any purpose without any further obligation.

It is acknowledged that any such documentation and any ancillary materials that are provided to OCP in connection with this document, including without limitation any white papers, articles, photographs, studies, diagrams, contact information (together, "Materials") are made available under the Creative Commons Attribution-ShareAlike 4.0 International License found here: <https://creativecommons.org/licenses/by-sa/4.0/>, or any later version, and without limiting the foregoing, OCP may make the Materials available under such terms.

As a contributor to this document, all members represent that they have the authority to grant the rights and licenses herein. They further represent and warrant that the Materials do not and will not violate the copyrights or misappropriate the trade secret rights of any third party, including without limitation rights in intellectual property. The contributor(s) also represent that, to the extent the Materials include materials protected by copyright or trade secret rights that are owned or created by any third-party, they have obtained permission for its use consistent with the foregoing. They will provide OCP evidence of such permission upon OCP's request. This document and any "Materials" are published on the respective project's wiki page and are open to the



public in accordance with OCP's Bylaws and IP Policy. This can be found at <http://www.opencompute.org/participate/legal-documents/>. If you have any questions please contact OCP.

About Open Compute Foundation

The Open Compute Project (OCP) is a collaborative Community of hyperscale data center operators, telecom, colocation providers and enterprise IT users, working with the product and solution vendor ecosystem to develop open innovations deployable from the cloud to the edge. The OCP Foundation is responsible for fostering and serving the OCP Community to meet the market and shape the future, taking hyperscale-led innovations to everyone. Meeting the market is accomplished through addressing challenging market obstacles with open specifications, designs and emerging market programs that showcase OCP-recognized IT equipment and data center facility best practices. Shaping the future includes investing in strategic initiatives and programs that prepare the IT ecosystem for major technology changes, such as AI & ML, optics, advanced cooling techniques, composable memory and silicon. OCP Community-developed open innovations strive to benefit all, optimized through the lens of impact, efficiency, scale and sustainability. Learn more at www.opencompute.org.

