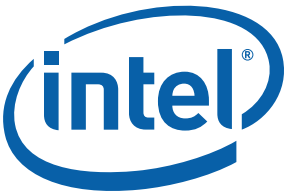

OS Machine Check Recovery on Itanium®-Based Systems

Application Note

August 2008

Document Number: 320482-001

Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

Contents

Figures

Tables

Revision History

Revision Number	Description	Date
–001	Public release.	August 2008

§

1 Introduction

The Intel® Itanium® processor family supports an advanced machine architecture, which allows processors to cooperate with the chipset, firmware, and operating system to contain, signal, correct, and log machine check errors. Most errors are corrected by processor or chipset hardware, but multilevel error handling allows Processor Abstraction Layer (PAL) firmware and System Abstraction Layer (SAL) firmware to provide additional error correction capabilities. To further enhance system availability and reliability, errors that cannot be corrected by hardware or firmware are handed off to the operating system for recovery.

This document primarily deals with *recoverable* machine check aborts (MCAs) and describes the actions SAL firmware and the operating system can take to successfully handle recoverable MCAs. This document explains how to distinguish between different MCAs, describes some MCA conditions that can be successfully recovered at the OS level, and provides guidelines for recovery. High-level pseudocode for the OS_MCA handler is provided to illustrate the OS actions needed to recover from an MCA. This document can benefit OS developers implementing MCA handling and recovery. SAL developers can benefit from better understanding the OS actions needed for recovery.

1.1 References

This document should be read in conjunction with the following documents:

- *Intel® Itanium® Architecture Software Developer's Manual* available at http://developer.intel.com.
- *Itanium® Processor Family System Abstraction Layer Specification* available at http://developer.intel.com.
- *Itanium® Processor Family Error Handling Guide* available at http://developer.intel.com.

1.2 Glossary

This section defines some of the basic terminology used in this document.

BERR#

Bus error signal. On most platforms, BERR# is routed globally. Processors may drive BERR# on contained fatal errors to bring the system into MCA. Unlike BINIT#, processor-asserted BERR# permits more complete logging than BINIT#, since bus agents do not lose internal state. The platform can assert BERR# to bring processors into a "corrected" MCA.

BINIT#

Bus initialization signal. The processor or platform may assert this signal to indicate a fatal machine check condition that requires bus agents to drop in-flight transactions to preserve error containment. This signal is routed globally and results in a global MCA.

bus_check

A structure within the processor error section of the SAL error record describing errors related to the system bus.

cache_check

A structure within the processor error section of the SAL error record describing cache errors.

Continuable

Error severity indicating that the error was isolated and contained and microarchitectural state captured at the time of the MCA allows the interrupted process to be resumed if any necessary corrective action has been taken.

Corrected

Error severity indicating the error (if any) has been corrected by the processor or platform hardware or firmware. The error may have been corrected without interruption to the executing process. If the processor or platform were configured to bring the affected processor(s) into MCA on corrected errors, the currently executing process may be resumed without taking any corrective action.

Data Poisoning

Upon identifying an uncorrectable memory error, processor or platform hardware may use ECC encoding to indicate that data has an uncorrectable error and defer MCA handling until the error is consumed.

Fatal

Error severity indicating the error cannot be corrected by hardware, firmware, or the operating system. Error logs can be stored to NVRAM, but the system must be rebooted to restore the system to a known good state.

Global MCA

An MCA which brings all processors in the system into MCA.

Hard Fail Response

System bus response for a transaction failure.

IPI

Interprocessor interrupt.

PEM

Processor experiencing machine check.

PL0

Operating system privilege level 0 – operating system execution.

PL3

Operating system privilege level 3 – application execution.

POM

Processor observing machine check.

PRM

Processor rendezvoused during machine check.

Recoverable

Error severity indicating the error has not been corrected by hardware or processor firmware. Firmware or the operating system must take corrective action such as using redundancy to restore state or terminating the currently executing application process.

Rendezvous

As an alternative to global MCA, SAL can bring slave processors into rendezvous spin-loop for global error handling. PAL or the OS may request that SAL bring processors into rendezvous. See the *Itanium® Processor Family System Abstraction Layer Specification* for more information.

reg_file_check

A structure within the processor error section of the SAL error record describing register errors.

tlb_check

A structure within the processor error section of the SAL error record describing TLB errors.

TC

Translation cache portion of the translation lookaside buffer.

TR

Translation register portion of the translation lookaside buffer.

uarch_check

A structure within the processor error section of the SAL error record describing microarchitectural errors.

§

2 Understanding Recoverable MCAs

This section describes MCA severity levels, local versus global machine check, and other concepts relevant to error recovery.

2.1 MCA Error Severities

The OS_MCA handler should first check the error severity that SAL reports in the SAL error record header ERR_SEVERITY field:

- Corrected
- Recoverable
- Fatal

If OS_MCA determines the error is recoverable, it can determine whether the error is continuable by examining the Processor State Parameter (PSP) software recovery bits, which are defined in Table 11-8 of the *Intel® Itanium® Architecture Software Developer's Manual, Volume 2.* An explanation of PSP interpretation for error recovery is provided in Table 2-1, "Processor State Parameter Machine Check Values".

Table 2-1. Processor State Parameter Machine Check Values

PAL Severity	PSP					Expected SAL Action	Expected OS Action
	cm	us	ci	co	sy		
Corrected	1	0	0	0	0	Unless promoted, corrected errors do not result in MCA hand-off to SAL.	Log error upon CMCI or poll for errors.
Recoverable, Continuable	0	0	1	1	0	Log error in NVRAM. Hand off to OS_MCA if the handler has been registered.	If possible, correct the error and resume the interrupted context or terminate applications to contain the error. Otherwise, shut down the system.
Recoverable, Not Continuable	0	0	1	0	0	Log error in NVRAM. Hand off to OS_MCA if the handler has been registered.	Terminate applications to contain the error or shut down the system.
Fatal	0	1	1	0	0	Log error in NVRAM. If the system has adequate functioning resources, SAL may hand off to OS_MCA. Otherwise, reset the system.	If SAL hands off to OS_MCA, report the error to the user and reset the system.

For platform errors, SAL determines the error severity by examining the platform error logging registers in addition to the record header ERR_SEVERITY field and the PSP.

Section B.2.2 of the SAL specification defines a Section Header field, ERROR_RECOVERY_INFO, that provides additional information about recoverable errors.

Even when SAL classifies an MCA as recoverable, the OS outcome depends upon the type of MCA that occurred, the context of the MCA, and the capabilities of the OS_MCA handler. In some cases, the OS can correct the error and continue execution of the interrupted process, changing the recoverable error to a corrected error. In other cases, the OS can terminate the affected application processes to allow the OS kernel to continue execution.

In cases where the OS is not able to recover, the error must be treated as fatal. Some examples of errors classified as recoverable by the firmware for which OS recovery may not be feasible are:

- The error is consumed in the OS kernel, and the OS is unable to recover.
- System bus hard fail response on outgoing IPI: The processor and platform may not be able to signal the MCA on the instruction that caused the MCA. If visible stores have occurred after the IPI instruction, the OS may be unable to restart the instruction sequence from the failed IPI instruction.
- The processor hardware logs provide physical addresses, but the OS needs the virtual address to affect recovery. The OS may not have accurate physical to virtual mapping for all addresses, which may prevent recovery from some MCAs.
- The MCA is caused by a software programming error (for example, TR purge) – not a hardware error.

2.2 Local versus Global Machine Check

A global MCA occurs when all the processors in the system enter the MCA flow. (A machine check rendezvous involves all processors, but does not result in all processors starting MCA flows.)

Bus-based Itanium processors provide bus initialization (BINIT#) and bus error (BERR#) signals to signal global MCAs. Although BINIT# assertion allows errors to be logged, the condition is always fatal and global. BERR# is routed globally on most systems, but this is not required architecturally. BERR# has two different uses. Itanium processors assert BERR# to bring processors into MCA to maintain error containment without reinitializing the bus. Platforms may assert BERR# to bring processors into a "corrected" MCA – the processors are brought into MCA, but no corrective action needs to be taken for the processors. Thus, a bus_check structure will always be present when a global MCA occurs. However, the presence of a bus_check structure on one processor does not imply a global MCA event.

In some MCA recovery situations, the OS may want to know whether the MCA is local to the processor or global so it can anticipate which processors will enter the handler and ensure that processors do not exit the OS_MCA handler until logging and handling is complete.

The bus_check.ib field indicates that either an error occurred on an internal bus or the processor asserted a BERR# or BINIT#. The bus_check.eb field is set when a processor receives a BERR#, BINIT#, or hard-fail bus response.

The implementation-specific bus_check.bsi allow the OS to determine whether a BERR#, BINIT#, or hard fail bus response occurred:

Table 2-2. PAL_MC_ERROR_INFO bus_check.bsi Values

Value	Description
0	Unknown/unclassified
1	BERR#
2	BINIT#
3	Hard Fail response received on the bus

If two of the defined bus_check.bsi conditions occur together, the condition with the highest encoding is reported. For example, if a BINIT# and a Hard Fail response occur together, a Hard Fail response is reported.

If BERR# or BINIT# is observed on the bus and the processor also has a bus error to report (in addition to the observation of BERR# or BINIT# on the bus), two bus checks are reported. The observation of BERR# or BINIT# is reported first and the other next.

2.3 Rendezvous

PAL can branch to SALE_ENTRY with a non-zero return vector address in GR19 to indicate that SAL should rendezvous slave processors and send the monarch back to PALE_CHECK to provide error containment without reinitializing bus agents for a fatal MCA.

If BERR# is not routed globally, SAL may signal a rendezvous interrupt to the slave processors on a platform-asserted BERR# to bring slave processors into rendezvous for global error handling.

The SAL to OS handoff indicates the rendezvous status in GR11. The handoff values are:

- –1 = Rendezvous unsuccessful
- 0 = Rendezvous not required
- 1 = Rendezvous successful using rendezvous interrupt
- 2 = Rendezvous successful using a combination of rendezvous interrupt and INIT

The OS may specify that SAL rendezvous slave processors for all MCAs by using the *rz_always* flag argument during the invocation of the SAL_MC_SET_PARAMS procedure. OS_MCA can check GR11 to determine whether rendezvous was successful and it gained control of all processors for MCA handling.

2.4 Error Containment

In most cases, Itanium processors check ECC or parity on data accesses and will correct a 1-bit error before putting a transaction on the bus. In some performance critical cases, the processor will check ECC or parity in parallel with forwarding the transaction to the bus; if a data error is detected in this situation, Itanium processors maintain error containment by asserting BINIT#. On BINIT#, processor and memory controller bus agents are responsible for dropping in-flight transactions and resetting their bus state.

Since Itanium processors and platform hardware maintains error containment, system firmware and operating systems do not have any error containment responsibilities.

2.5 Min-State Save Area I-Resources and X-Resources

On an interruption (either PAL-based or IVA-based), the processor stores architectural state to the I-resources (IIP, IPSR, IIM, and IFS). During interrupt handling, interrupt collection is masked with PSR.ic = 0, but PSR.mc = 1 and machine check aborts can be delivered.

To permit error recovery when PSR.ic = 0, current Itanium processor implementations provide optional X-resources (XIP, XPSR, XFS, XR0 – XR4). (Availability of X-resources on a processor implementation can be identified using PAL_PROC_GET_FEATURE bits 41 and 42.) If an MCA occurs while PSR.ic = 0, the I-resources are saved to the X-resources and the processor state at the time of the MCA is stored to the I-resources.

The PAL MCA handler will copy I-resources and X-resources to the min-state save area. SAL_CHECK saves the min-state save area to NVRAM in the processor error section and provides the error record to OS_MCA when SAL_GET_STATE_INFO is called. OS_MCA can determine if an interruption was in progress at the time of the MCA by examining IPSR.ic. If IPSR.ic = 0, the X-resources provide information about the processor state at the time the original interruption was taken. If IPSR.ic = 1, the X-resources are undefined.

After successful error recovery, OS_MCA can return to SAL_CHECK with an indication to resume execution of the MCA-interrupted context. SAL_CHECK will call PAL_MC_RESUME, which will restore X-resources to the I-resources and will allow any interruption that was in progress when the MCA was taken to continue execution.

2.6 IA-32 Instruction Execution

If the IPSR.is =1, indicating IA-32 instruction execution at the time of the machine check, MCAs are not continuable and the error will be handed off as recoverable, not continuable.

Note that the OS-based IA-32 Execution Layer operates with ISPR.is = 0, so MCAs that occur when IA-32 applications are running can be handed off as continuable.

§

3 Recoverable Processor Errors

This section describes recoverable processor errors and how to identify them.

3.1 Translation Register and Translation Cache Errors

On Itanium processors, the translation lookaside buffer consists of translation caches (TCs) co-managed by hardware/software and software-controlled translation registers.

If a translation results in multiple hits in the TRs and/or TCs, which could be due to hardware errors or misprogrammed TRs, the processor initiates an MCA. For example, if one of the bits in a TR gets corrupted and the OS attempts to install a TC, the erroneous TR may result in multiple hits in the TLB and an MCA. A similar problem may be caused by the OS programming the TCs or TRs incorrectly. Processor error logging does not allow software to determine whether the multi-hit collision was due to a hardware error or an OS error.

The dual-core Itanium processor also provides parity protection on TRs and TCs. An MCA will be initiated if a parity error is detected on the TRs or TCs. Note that processor error logging does not allow software to determine whether the TLB errors were caused by multi-hit collisions or parity errors.

Hardware TLB errors can be addressed by flushing the TCs and reloading TRs from a redundant copy maintained by the OS to restore the TLB to a known good state. This document assumes that multi-hit collisions due to OS programming errors would be addressed by correcting the faulty OS code. If a TLB MCA due to OS programming occurs, flushing the TCs and reloading TRs will not correct the error and a subsequent MCA will occur after execution resumes.

Table 3-1 and Table 3-2 specify how to identify recoverable TLB errors:

Table 3-1. Itanium® Processor Recoverable TLB Errors

Event	Severity	Check Field	Check Field Values
ITLB2 Data Parity error OR Multiple Hits - Errors On TRs (Ifetch, prefetch)[1]	recov.	tlb	level=1, itr=1, op=3, is=IPSR.is, iv=1
ITLB2 - TR purge (any operation that can cause a TR to be purged)[1, 2]	recov.	tlb	level=1, itr=1, op=8, is=IPSR.is, iv=1
DTLB2- TR purge (any operation that can cause a TR to be purged)[1, 2]	recov.	tlb	level=1, dtr=1, op=8, is=IPSR.is, iv=1
DTLB2 - Page mask overflow (localpurge)[1, 2]	recov.	tlb	level=1, dtr=1, dtc=1, op=8, is=IPSR.is, iv=1

Notes:

1. Recoverable if OS can re-install the TRs.
2. The TR is purged. Recoverable only if the OS can re-install the TRs. However, the firmware will return to the interrupted context and re-execute the offending instruction and the MCA will recur. This pattern will likely continue indefinitely until the code is fixed.

Table 3-2. Dual-Core Itanium® Processor Recoverable TLB Errors

Event	Severity	Check Field	Check Field Values
DTLB2- Parity error OR Multiple Hits - Error on either TCs or TRs - caused by active thread (detected on lookups)[1]	recov.	tlb	level=1, dtr=1, dtc=1, is=IPSR.is, iv=1, pl=IPSR.cpl, pv=1, pi=1
DTLB2- Illegal TR purge caused by active thread (detected on any operation that can cause a TR to be purged)	cont.	tlb	level=1, dtr=1, op=8, is=IPSR.is, iv=1
DTLB2- Page mask overflow caused by active thread (detected on local purge)	cont.	tlb	level=1, dtr=1, op=8, is=IPSR.is, iv=1
ITLB2 Data Parity error OR Multiple Hits - Errors on TRs[1]	cont.	tlb	level=1, itr=1, op=3, is=IPSR.is, iv=1, pl=IPSR.cpl, pv=1, pi=1
ITLB2- Illegal TR purge caused by active thread (detected on any operation that can cause a TR to be purged)	cont.	tlb	level=1, itr=1, op=8, is=IPSR.is, iv=1

Notes:

1. Not possible to log whether TR or TC affected.

Since the error is TLB related, the OS must avoid virtual addressing during recovery to avoid causing a nested TLB machine check abort.

OS_MCA can use the following steps to recover from a TLB error:

1. Examine the PSP.tc bit and the reported severity to determine if a recoverable TLB error occurred.
2. Purge all TCs by executing the `PTC.e` instruction in a loop, using parameters returned by the PAL_PTCE_INFO procedure. Note that current PAL implementations perform this operation during firmware error handling before transferring control to the OS MCA handler.
3. The OS_MCA handler must then purge and reinstall all the TRs for the context from a redundant copy.
4. Switch to virtual mode.
5. Check the detailed information in the TLB error record to confirm that a recoverable TLB error occurred. If not, return to SAL_CHECK with an uncorrected status value and an indication to halt or reboot the system.
6. Switch back to physical mode.
7. Return back to SAL_CHECK with a “corrected status” indicating that SAL should resume execution of the interrupted context.

To perform recovery from TR errors, the OS needs to maintain a data structure containing a redundant copy of TR values. Here are the recommended steps for revising the OS data structure, however, not all operating systems require these steps:

1. Remove the TR information for the old context from the OS data structures.
2. Execute a memory fence instruction to ensure visibility of the stores.
3. Remove the TRs from the processor TLBs using the `PTR.i/d` instruction.
4. Store the TR information for the new context into the OS data structures.
5. Execute a memory fence instruction to ensure visibility of the stores.
6. Install TRs into the processor using the `ITR.i/d` instruction.

3.2 Register File Errors

Dual-Core Itanium processors provide register file parity protection. If the register error MCA occurs in an application context, OS_MCA can terminate the application to maintain system availability.

For register parity errors, the MCA the interrupted context will be the precise context that caused the MCA to occur.

Table 3-3. Dual-Core Itanium® Processor Recoverable Register File Errors

Event	Severity	Check Field	Check Field Values
Parity error on GR[1]	recov.	reg_file	id=1,2; is=IPSR.is, iv=1, pl=IPSR.cpl
Parity error on FR	recov	reg_file	id=3, is=IPSR.is, iv=1, pl=IPSR.cpl, pv=1, pi=1

Notes:

1. General register NaT bits are not protected.

3.3 Recoverable Cache and Memory Errors

Errors related to memory may arise during loads from memory as well as during stores to memory.

3.3.1 Data Poisoning

Itanium processors implement a data poisoning model for handling uncorrectable multibit errors, which allows for deferred handling.

Itanium processors provide ECC on the system bus, L3 cache, and L2 data cache. These processor structures will correct 1-bit errors. For multibit errors, the observation of the multibit error will be signaled to the operating system using a CMCI with the bus_check.dp or cache_check.dp bit set. If processor execution results in transfer of the multibit error to a register or lower-level cache that only has parity encoding and cannot maintain the data poisoning encoding, the error is "consumed" and an MCA will occur.

Inbound multibit errors from the processor bus may either be directly consumed or transferred to L3 cache. Multibit errors in the L3 cache are not directly consumed, but they can be transferred to lower levels of cache. Multibit errors can be directly consumed from the L2 data cache, resulting in an MCA.

Note that by default data poisoning events are signaled using CMCI and handed off showing a "Corrected" state to indicate that no immediate action is required, however, the error is actually uncorrected. PAL_PROC_SET_FEATURES bit 53, introduced on the dual-core Itanium processor, changes the signaling of data poisoning detection from CMCI to MCA.

Data poisoning CMCI to MCA promotion was originally defined to change the signaling but hand off the *error unchanged as a "Corrected" error. Specification Change 17 published in the Intel® Itanium® Architecture Software Developer's Manual Specification Update, June 2008* will change the PAL_PROC_SET_FEATURES bit 53 definition to an MCA hand-off with a "Recoverable/Continuable" status on future Itanium processors.

The MCAs due to consuming poisoned data will be signaled at or before the use of the load. For example, in the code sequence below:

```
LabelA:     ld8 r15 = [r16]

LabelB:     mov r17 = r18

LabelC:     add r19 = r20, r21;;

…

…

LabelD:     mov r22 = r15       // MCA is signaled at or before this instruction
```

If the data pointed to by register GR16 is poisoned in memory, a local MCA will surface at any point during the interval from instruction LabelA through LabelD, with the data being consumed at LabelD, since the application registers only have parity protection and are unable to maintain the data poisoning encoding.

3.3.2 Precise and Interrupted Contexts

Memory operations have a long latency relative to processor execution, and the OS may switch contexts before the error is consumed. It is important to distinguish between the "precise" context, which is the context that caused the MCA to occur and the "interrupted" context, which is the context that was running when the MCA was raised. To recover from a multibit memory error, OS_MCA should terminate the precise context – not the interrupted context. The precise context and the interrupted context are independent, and they may even have different privilege levels.

Current Itanium processors do not report the precise instruction pointer of the instruction that caused the MCA to occur (LabelA: in the above example code) or the precise privilege level.

Itanium processors report the target address associated with the MCA (r16 value in the above example), but the target address generally can't be used to identify the precise context, since register values may have changed since the memory operation was initiated.

Itanium processors report information about the interrupted context in the I-resources. Although the interrupted context is independent of the precise context, OS_MCA can make conservative decisions about how to recovery from multibit memory errors using an understanding of the OS design.

On a context switch, the OS kernel must store all process register state to memory, which would cause any latent memory errors to be consumed. If unconsumed user data is consumed by the OS kernel during a context switch, OS_MCA can be conservative and treat the condition as a fatal MCA. If unconsumed kernel data gets consumed after a context switch to an application, we may be able to assume that the kernel didn't expect to consume the data and OS_MCA can conservative by killing the consuming application. Although the interrupted application wasn't at fault in this case, killing the application is safe and maintains system availability.

3.3.3 Errors During Stores to Memory

When poisoned data from cache is written back to memory with a multibit error, the processor will signal a CMCI, and the platform should poison the data in memory. Some chipsets may signal a CPEI on receipt of the poisoned memory data.

Any consumers of the data will incur a local MCA. Current Itanium processors provide data poison promotion which is configurable by SAL. If data poisoning promotion is enabled, data poison entering or leaving the processor will signal an MCA instead of a CMCI.

3.3.4 Errors During Loads and Fetches from Memory

SAL will report multibit errors in data loaded from memory as local MCAs. Memory reads may occur due to data loads or instruction fetches. Data loads may also occur during stores if the processor uses write-allocate caching.

Note that data poisoning detection will be signaled by a bus_check CMCI, and data poisoning consumption will be reported as a cache_check MCA. The data poisoning detection CMCI will be reported as an external bus error in the Bus_Check structure within the SAL processor error section (Bus_Check.eb = 1). The type of transaction should be a "full line read" (Bus_Check.type = 3) or "partial read" (Bus_Check.type=1).

The physical address of the load will be marked as valid in the Bus_Check structure (Bus_Check.tv = 1), and the physical address will exist in the SAL error record (MOD_TARGET_IDENTIFIER field in the BUS MOD_ERROR_INFO_STRUCT structure). The OS must verify that the physical address of the load is within the memory range to ensure that the operation was a load from memory, rather than from a memory-mapped device.

Following the processor error section, the SAL error record may have a memory error section indicating the physical address of the error logged by the memory controller. This section will also have the field replaceable unit information of the failing memory component (card, DIMM, bank, row, column, and so on). The field replaceable unit information is important if the OS maintains threshold statistics.

There are situations where the memory error section may not be present. It is possible for two local MCAs to surface simultaneously in a multiprocessor configuration due to loads by independent applications. If SAL handles these MCAs in sequence, SAL might first retrieve all the errors logged by the chipset. Many chipsets have two levels of error logs. SAL might report two memory error sections for the first local MCA and none for the second local MCA. The OS, using the processor error section, can terminate both the affected applications.

Table 3-4 and Table 3-5 document how to identify MCAs resulting consumption of memory errors. Note that memory errors consumed directly from the bus will be reported as an L2D cache error.

Table 3-4. Itanium® Processor Recoverable Cache/Memory Errors

Event	Severity	Check Field	Check Field Values
Poison filled to L2 cache from the bus[1]	recov.	bus	op=δ[2], level=1, dl=1, dc=1, way=δ, wiv=1, index=δ, is=IPSR.ir, iv=1

Notes:

1. Signalled by L2 logic, but reported as a bus error
2. δ = don't care

Table 3-5. Dual-Core Itanium® Processor Recoverable Cache/Memory Errors

Event	Severity	Check Field	Check Field Values
L2D data multibit error on consumed poison	recov.	cache	op=1,3; level=1, dl=1, dc=1, way=δ[1], wiv=1, index=δ, is=IPSR.is, iv=1, tv=1
L2D data multibit error on non-WB/non-snoop, not on 4-byte aligned store, line has not been replaced	recov.	cache	op=1,3; level=1, dl=1, dc=1, way=δ, wiv=1, index=δ, is=IPSR.is, iv=1, tv=1
L2D data multibit error on non-WB/ non-snoop, not on 4-byte aligned store, line has been replaced[2] or no valid min state save pointer	recov.	cache	op=1,3; level=1, dl=1, dc=1, way=δ, wiv=1, index=δ, is=IPSR.is, iv=1
L1I poison consumption - happening twice	recov.	cache	dl=1 if data error; tl=1 if tag error; ic=1; is=IPSR.is, iv=1, pl=IPSR.cpl, pv=1, pi= 1

Notes:

1. δ = don't care
2. The target address is not reported.

The OS keeps track of the pages that have poisoned memory. When there are no applications referring to the page with poisoned memory, the OS may clear the poisoned page and recycle the page for use by other applications. These steps are described in Section 4.7, "Cleaning a Poisoned Memory Page".

Until the poisoned page is cleared, the OS can avoid additional MCAs from arising from the poisoned memory page by marking the poisoned page as not eligible for I/O write. This would prevent that page from being written to backing store, which would generate another MCA. This step is unnecessary if the OS or device driver can recover from MCAs during transfer of data from the poisoned memory page to the device.

3.4 Other Recoverable Errors

Table 3-6. Other Itanium® Processor Recoverable Errors

Event	Severity	Check Field	Check Field Values
FSB hard failure response: outgoing ptc.g[1]	cont.	bus	size=δ[2], eb=1, type=7, bsi=3, is=IPSR.is, iv=1, tv=1
FSB hard failure response: I/O port space read[3]	recov.	bus	size=δ, eb=1, type=9, bsi=3, is=IPSR.is, iv=1, tv=1
FSB hard failure response: I/O port space write[3]	recov.	bus	size=δ, eb=1, type=10, bsi=3, is=IPSR.is, iv=1, tv=1
FSB hard failure response: outgoing IPI, incoming IPI not directed at detecting processor[3]	recov.	bus	size=δ, eb=1, type=11, bsi=3, is=IPSR.is, iv=1, tv=1; (size, type, tv cannot be safely reported if incoming IPI)
FSB hard failure response: read, rfo, ifetch (prefetch, demand)[2, 3]	recov.	bus	size=δ, eb=1, type=3, bsi=3, is=IPSR.is, iv=1, tv=1
Watchdog timer half-expiration[4]	recov.	uarch	sid=1, op=4, is=IPSR.is, iv=1
Illegal ISA transfer[5]	recov.	uarch	op=3, is=1, iv=1

Notes:

1. Recoverable if OS flushes all TCs in the system
2. δ = don't care
3. System needs to decide whether its feasible to restart the transaction or application.
4. The timeout counter is reset every time an instruction is retired. It continues to count down after the MCA is signalled; if it expires before the first instruction of PALE_CHECK is retired, the BINIT# event (error "Timeout counter expiration") occurs. In most cases, if the half-expiration MCA occurs, a deadlock condition has been encountered that would probably prevent PALE_CHECK from being fetched; the BINIT# would therefore most probably occur.
5. A change between Itanium® Architecture and IA-32 execution that does not comply with the rules described in the *Intel® Itanium® Architecture Software Developer's Manual*.

Table 3-7. Other Dual-Core Itanium® Processor OS Recoverable Errors

Event	Severity	Check Field	Check Field Values
FSB Hard Fail response: outgoing ptc.g[1]	cont.	bus	eb=1, type = 7, bsi=3, is=IPSR.is, iv=1
FSB hard failure response: I/O port space read[2]	recov.	bus	size=δ[3], eb=1, type=9, bsi=3, is=IPSR.is, iv=1, tv=1
FSB hard failure response: I/O port space write[2]	recov.	bus	size=δ, eb=1, type=10, bsi=3, is=IPSR.is, iv=1, tv=1
FSB Hard Fail response: outgoing IPI, incoming IPI not directed at detecting processor[2]	recov.	bus	size=δ, eb=1, type=11, bsi=3, IPSR.is=is, iv=1, tv=1; (size, type, tv cannot be safely reported if incoming IPI)
Timeout counter half expiration[4]	recov.	uarch	sid=1, op=4, is=IPSR.is, iv=1
Illegal ISA transfer[5]	recov.	uarch	op=3, is=1, iv=1

Notes:

1. Continuable if OS flushes all TCs in the system.
2. System needs to decide whether feasible to restart the application.
3. δ = don't care.
4. The timeout counter is reset every time an instruction is retired. It continues to count down after the MCA is signalled; if it expires before the first instruction of PALE_CHECK is retired, the BINIT# event (error "Timeout counter expiration") occurs.
5. Recoverable by terminating the affected application, if it can be determined without a target address. This is intended mainly for recovering from livelock in the midst of an IA- 32 instruction. If the timeout counter expires during execution of an Itanium instruction, it is highly unlikely that the MCA would break the locking condition.

§

4 OS_MCA Example

This chapter describes a sample OS_MCA implementation to illustrate the actions an OS_MCA handler must consider during recovery. This sample implementation considers multiprocessor error recovery, which increases complexity. OS_MCA implementations may choose to limit error recovery to local MCAs, which simplifies recovery.

4.1 Processor Roles During Recovery

The OS may classify processors involved in an MCA recovery event using the SAL error records to determine their role in the recovery.

- **Processors Experiencing the Machine Check** (PEMs)

  PEMs are processors that initiated a machine check abort. If the platform initiated the machine check abort by asserting BERR# or BINIT#, there will be no PEMs. If concurrent errors occurred, there may be multiple PEMs. The SAL error record for PEMs would have:

  — One or more of tlb_check, cache_check, reg_file_check, or uarch_check structures or

  — A bus_check structure with the bus_check.ib field set to 1, indicating an internal bus error

- **Processors Observing the Machine Check** (POMs)

  POMs are either processors that received a BERR# or BINIT# signaled by another processor. The SAL error record for POMs meet the following requirements:

  — A processor error section with a bus_check.eb = 1 and bus_check.bsi = 1 (BERR#) or = 2 (BINIT#)

- **Processors Rendezvoused during Machine Check** (PRMs)

  PRMs are processors rendezvoused using the SAL rendezvous interrupt or the INIT IPI. The OS typically maintains a list of such processors during the processing of the rendezvous interrupt, so that the PRMs may be woken up at the end of the MCA recovery.

  The PRMs wait in a loop within the SAL handler with machine checks unmasked, in order to recognize a subsequent global MCA event.

- **Processors that could not be rendezvoused successfully**

  If SAL cannot successfully rendezvous the processors, rendezvous failure will be indicated at the handoff to the OS_MCA handler. Operating systems may consider this to be a fatal condition.

Table 4-1, “MP Synchronization – OS_MCA Recovery” provides an overview OS_MCA handler actions on PEMs and POMs.

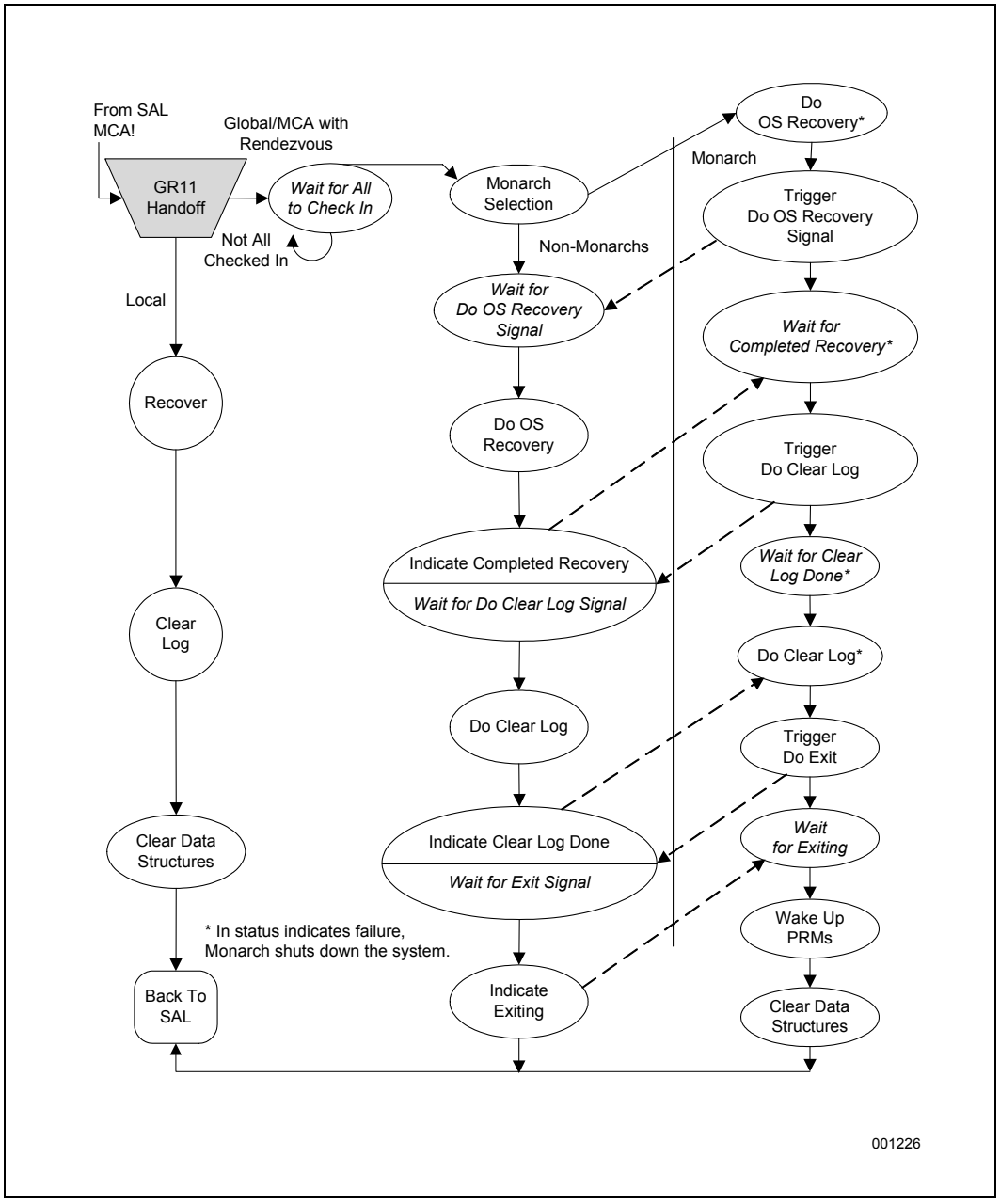

Figure 4-1. MP Synchronization – OS_MCA Recovery

All PEMs and POMs will enter the OS_MCA handler. Note that OS_MCA cannot identify the SAL monarch from the order in which processors enter OS_MCA. In a rendezvous situation, the PRMs will be waiting in a SAL rendezvous loop. The OS handles the rendezvous interrupt sent by SAL. If the OS maintains a data structure indicating which processors received the rendezvous interrupt, it can determine the identity of the other processors that will enter the OS_MCA entry point. Such a list of rendezvoused processors is maintained in an array called PRMCpuList[cr.lid].

The Itanium architecture follows the release consistency memory ordering model. In the discussion below, multiple processors will need to manipulate global data structures in writeback memory. The OS_MCA code must follow appropriate fencing steps to

ensure visibility of loads and stores across the system. In general, loads should have acquire semantics, and stores should follow the release semantics. If a data variable needs to be revised atomically, the `FetchAdd` instruction should be used.

Suggested data structures below are shown using an array index "[cr.lid]." This does not imply the need to support all possible combinations of processor ID and EID. The OS may implement a sparse matrix based on the maximum supported system configuration.

4.2 Handling Multiple Simultaneous MCAs

This section describes the SAL and OS requirements for handling multiple simultaneous MCAs. SAL and the OS must be capable of handling different MCA conditions such as: one or more simultaneous local MCAs, a local MCA followed by a global MCA, a global MCA followed by a local MCA or another global MCA, MCAs involving rendezvous, and so on.

An MCA involving rendezvous affects all processors, but some of the processors (PRMs) may be in the SAL rendezvous loop waiting for the wake-up signal from the OS monarch processor. If a machine check occurs during such a wait, the PRM processor will begin execution at the machine check layer within PAL, and its MCA type no longer fits the PRM classification.

If a PRM were to experience a BINIT#, the current MCA would be interrupted and the processor would be steered to a new MCA for the BINIT#. However BINIT# is global, and the PEMs and POMs would also experience the BINIT# and their execution would also be steered to a new MCA. The SAL and the OS on the PEMs and POMs would process the BINIT# condition, resulting in a system reset.

The handling of simultaneous machine check events is described in Table 4-1, "Simultaneous MCAs".

Table 4-1. Simultaneous MCAs

First Event	Second Event/Comments		
	Local MCA On Another Processor	**BERR#**	**MCA On Another Processor Involving Rendezvous**
Local MCA on Processor P1	Handle as independent MCA events.	1. SAL waits for check-in by all processors. P1 will not check into SAL yet. 2. If timeout in SAL, jump to OS_MCA with fatal handoff status. 3. If no timeout, and current PEM (P1) exits out of PAL, the global MCA is recognized on P1, and P1 will re-enter PAL MCA. 4. SAL MCA executes, SAL to OS transition indicates Global MCA.	1. SAL waits for check-in by all processors. P1 will not check into SAL_MC_RENDEZ yet. 2. If timeout in SAL, jump to OS_MCA with rendezvous failure status. 3. If no timeout, and current PEM (P1) exits out of PAL, the rendezvous/INIT interrupt is recognized on P1 and P1 will check into SAL_MC_RENDEZ. 4. SAL MCA executes, SAL to OS transition indicates successful rendezvous.
BERR#	Nested MCA handled when current MCA concludes.	Nested MCA handled when current MCA concludes.	Nested MCA handled when current MCA concludes.
MCA Involving Rendezvous	1. If PEM/POM, nested MCA handled when current MCA concludes. 2. If PRM: • Recognize MCA as occurring during rendezvous. • Wait in SAL until previous MCA is completed on other PEMs/POMs, then jump to OS_MCA or • Jump to OS_MCA with Fatal error status.	1. If PEM/POM, nested MCA handled when current MCA concludes. 2. If PRM: • Recognize MCA as occurring during rendezvous. • Wait in SAL until previous MCA is completed on other PEMs/POMs, then jump to OS_MCA or • Jump to OS MCA with Fatal error status.	1. If PEM/POM, nested MCA handled when current MCA concludes. 2. If PRM: • Recognize MCA as occurring during rendezvous. • Wait in SAL until previous MCA is completed on other PEMs/POMs, then jump to OS_MCA or • Jump to OS MCA with Fatal error status

4.3 Processor Check-In

Each processor entering the OS_MCA entrypoint (that is, the PEMs and POMs) saves its state in a unique save area.

Each processor stores its identification in an array, CpuInMcaList[cr.lid]. This is an array of structures with fields for ID, EID, Local/Global indication, Error Severity, type of MCA, state of MCA processing, status of OS_MCA recovery, status of Clear Log, and so on. An example organization of this structure is depicted in Table 4-2, "CpuInMcaList Data Structure".

Table 4-2. CpuInMcaList Data Structure

Valid Bits (0...n)	Proc. ID	Proc. EID	MCA Scope	Error Severity	Type of MCA	State of MCA Processing	OS Recovery Status	Clear Log Status
Value of 1 Indicates Field Presence	From CR.lid	From CR.lid	0 = Invalid 1 = Local 2 = Global	0 = Invalid 1 = Recoverable 2 = Fatal	0 = Invalid 1 = POM 2 = PEM	0 = Invalid 1 = Waiting for Monarch Selection *(SyncPoint-1)* 2 = Waiting to do OS Recovery *(SyncPoint-2)* 3 = Proceed to OS Recovery 4 = Completed OS Recovery, Waiting to do Clear Log *(SyncPoint-3)* 5 = Do Clear Log 6 = Clear Log complete, Waiting for Exit signal *(SyncPoint-4)* 7 = Exit from OS_MCA 8 = Exiting from OS_MCA *(SyncPoint-5)*	0 = Invalid 1 = Success 2 = Failure	0 = Invalid 1 = Success 2 = Failure

Initially, the ID, EID, and local/global indication (from GR11) are stored to indicate the check-in to the other processors in the OS_MCA handler. Valid bits are set as fields are stored into the table. The type of MCA and the error severity from the SAL error record will be filled after parsing the SAL error record.

Each processor retrieves its SAL error record by calling SAL_GET_STATE_INFO. The SAL_CLEAR_STATE_INFO procedure may be invoked immediately or deferred, depending on the OS implementation. If the error severity is fatal, some operating systems may choose to retrieve the error record from SAL during the next reboot and hence will not invoke the SAL_CLEAR_STATE_INFO procedure in the OS_MCA path. The OS image on each processor parses the SAL error record and fills in the error severity and the type of MCA (PEM/POM) fields into the CpuInMcaList[cr.lid] array. The processors set their MCA processing status to “Waiting for Monarch Selection” (*SyncPoint-1*).

4.4 Recovery Steps for Local MCA

If the MCA is local, there is no need for coordination with other processors. The OS recovery code will recover from the MCA, clear the SAL error log, and return to the interrupted context. If the recovery is unsuccessful, the OS may display an error message and should return to SAL_CHECK with an uncorrected status value to indicate whether SAL should halt or reboot the system.

4.5 Recovery Steps for Global MCA

If the MCA is global, each processor examines the CpuInMcaList array and waits for all PEM/POM classifications and error severities to be registered.

If a processor is classified as a non-OS monarch POM, it sets its MCA processing status to *"Completed OS Recovery, Waiting to Do Clear Log"* (see *SyncPoint-3* below). The POMs then wait for the OS monarch to change the MCA processing status to *"Do Clear Log."*

4.5.1 OS Monarch Selection

All PEM processors examine the error severity of all the PEMs and elect one with the highest severity to be the OS monarch. If the MCA was a platform-asserted BERR# or BINIT# and no PEM exists, the OS monarch will be elected from the POMs. The OS monarch then acquires the semaphore OS_Monarch_Semaphore. Since fatal and recoverable are only two severity levels, if a fatal error is noticed on any of the PEMs, that PEM would become the OS monarch, display an error message and shut down the system.

There are some situations where the error severity must be promoted to fatal. The OS monarch needs to examine register GR11 at SAL to OS handoff. If this value is –1, rendezvous was unsuccessful. The OS monarch must treat this condition as a fatal MCA regardless of the error severity in the SAL error records.

If the MCA is recoverable, the OS monarch proceeds with the recovery. The non-OS monarch PEMs revise their MCA processing status to *"Waiting to Do OS Recovery"* and then wait for the OS monarch to change this status to *"Proceed to OS Recovery"* (SyncPoint-2).

4.5.2 Error Recovery

At this point, the OS monarch proceeds with OS recovery. Error records of all the processors are available, and the OS monarch can decide on the recovery steps by examining all the error records. For the recoverable cases that we have identified, there is no need for such interprocessor coordination.

The OS monarch takes necessary recovery steps, then revises the OS_Recovery_Status field to indicate the completion status of its recovery. The OS monarch then revises the state of MCA processing field of PEMs to *"Proceed to OS Recovery,"* one PEM at a time to cause the PEMs to go through the OS recovery steps appropriate for each PEM's SAL error record. The OS monarch must monitor the PEM's state of MCA processing field to be changed to *"Completed OS Recovery, Waiting to Do Clear Log."*

The OS monarch and each PEM takes the appropriate recovery steps including setting the new context for resumption from MCA. At the end, the PEM stores the completion status for the recovery in the OS_Recovery_Status field of the CpuInMcaList[cr.lid] array. Different values indicate the success or failure of the recovery operation. The non-OS monarch PEMs then change their state of MCA processing to *"Completed OS Recovery, Waiting to Do Clear Log"* and wait on their state to be changed by the OS monarch to *"Do Clear Log"* (SyncPoint-3).

4.5.3 Check Status of Recovery

When the monarch and the PEMs have finished their recovery and reached *SyncPoint-3*, the OS monarch examines the OS_Recovery_Status fields of the monarch and PEMs to determine if recovery was successful. If unsuccessful, the OS monarch can display an error message using one of the error records that caused the failure in the OS_MCA and then shut down the system.

4.5.4 Perform Error Logging

If all the PEMs report a successful MCA recovery, the OS monarch revises the state of MCA processing field of other PEMs and POMs to *"Do Clear Log"* to direct them to perform the logging functions. This may be done on one PEM/POM at a time or

simultaneously on all. All PEMs and POMs must invoke the SAL_CLEAR_STATE_INFO procedure to clear the SAL error record, although the OS may not choose to log POM error records.

If the OS_MCA is not capable of logging the MCA event to disk from the OS_MCA context, it may copy the SAL error record to a separate buffer and trigger an OS event for saving the buffer to disk. For some operating systems, this may be just moving the error record to a separate OS buffer and triggering the write later.

The processors deposit the completion status of their Clear Log into the CpuInMcaList[cr.lid] array, revise their state of MCA processing to *"Clear Log complete, Waiting for Exit signal"* and then wait for this state to be changed by the OS monarch to *"Exit from OS_MCA"* (*SyncPoint-4*). The OS monarch performs similar logging functions.

4.5.5 Direct Other Processors to Exit OS_MCA or SAL

When all the PEMs and POMs have finished their logging and reached *SyncPoint-4*, the OS monarch examines the status of their Clear Log for success/failure of logging operation. If unsuccessful, the OS monarch can display an error message and shut down the system.

If the error recovery and logging are successful, the OS monarch revises the state of MCA processing field to "*Exit from OS_MCA*" to direct them to exit from the OS_MCA handler through SAL and PAL. The PEMs and POMs revise their state of MCA processing to *"Exiting from OS_MCA"* as they exit to indicate their progress of MCA processing (*SyncPoint-5*).

When the OS monarch verifies that all other PEMs and POMs have left the OS_MCA handler, the OS monarch wakes up all the PRMs using the wakeup interrupt (IPI or memory semaphore specified in SAL_MC_SET_PARAMS procedure).

4.5.6 Reinitialize Data Structures for the Next MCA

The OS monarch reinitializes the data structures for use during the next MCA. The variables to be initialized include (more may be needed depending on OS_MCA implementation):

```
OS_Monarch_Semaphore

CpuInMcaList[cr.lid]

PRMCpuList[cr.lid]
```

4.6 Terminating the Precise Thread

Once the OS_MCA handler has made a decision to terminate the precise thread, it must return to the SAL_MCA handler with the address of a new min-state save area pointer that should be supplied to the PAL_MC_RESUME procedure.

PAL_MC_RESUME allows the OS to resume to a new context and set new values for both the IIP (IP at the time of MCA) and the XIP (IIP at the time of MCA) registers within the new min-state save area without losing state information. Since only the affected thread (PL3 application) is being terminated, and at the time of MCA, the application would have been executing with the PSR.ic set to 1. In such a context, the application's IIP and IPSR values are volatile.

The OS needs to call PAL_PROC_GET_FEATURES to determine whether the processor implements the min-state save area X-resources (XFS, XPSR, and XIP). (The Itanium and dual-core Itanium processors implement X-resources.)

On exit from the machine check, the OS can transfer control to a fault handler by setting values in the new min-state save structure as follows:

- The IIP will point to the OS fault handler.
- The IPSR will have values appropriate for the fault handler. The IPSR.i and IPSR.ic must be 0 to ensure that the fault handler is not preempted. The IPSR.bn, in most OS implementations, would be 0 to enable use of Banked GRs 24-31 without performing any register saves.
- The XIP will point to an address within the offending application.
- The XPSR will have the value of the PSR at the time of MCA.

The normal termination code within the OS may be employed to terminate the precise thread.

4.7 Cleaning a Poisoned Memory Page

Once the OS has terminated the applications that had a mapping to the poisoned memory page, it may choose to keep the page offline or recycle the page for use by other applications.

The OS generally cannot determine whether an error occurring in a memory page is due to a transient error resulting from a random radiation strike or a persistent failure due to a manufacturing error or wear-out phenomena. In most cases, the uncorrectable memory error rate is low enough that poisoned memory pages can be left offline for the remaining OS uptime and the pages can be examined at reboot self-test.

If an OS does choose to recycle poisoned memory pages, care must be taken to successfully clear the error and avoid an inadvertent secondary MCA. If the OS were to attempt a store of zeroes to the problem memory area, there will be a read of the cache line from poisoned memory resulting in another local MCA on processors with write-allocate caches. The recommended procedure is to first change the memory attribute of the problem page from writeback to uncacheable and then storing zeroes to the poisoned memory area. Please refer to the description of "Disabling Prefetch and Removing Cacheability" in the *Intel® Itanium® Architecture Software Developer's Manual* for the detailed steps. This procedure requires execution of the PAL_PREFETCH_VISIBILITY and the PAL_MC_DRAIN procedure calls on all the processors within the system. Poisoned cache lines that are modified are flushed to memory, however the flush generates only a Corrected Machine Check Interrupt (CMCI) and does not result in another MCA.

The OS must allow for the fact that ECC methods differ across platforms. For example, the memory controller on the E8870 chipset uses single device data correction ECC (12 check bits cover 32 bytes), while the Itanium processor system bus uses a different number of check bits and covered bytes. An uncorrectable memory error will always have larger footprint than 8 bytes, whether the source of the error is a real multibit error or data poisoning. The OS must clear a minimum of 4K bytes at the poisoned memory location.

When the page has been cleared of poison, the OS can revise its data structures for poisoned memory pages. Some OS implementations may choose not to recycle such pages based on thresholding statistics. SAL may provide such an indication in the SAL error record using the "Error threshold exceeded" bit in the ERROR_RECOVERY_INFO field of the memory error section.

4.8 Maintaining Error Statistics for Error Prediction

The OS may maintain memory error statistics and avoid using pages that generate too many corrected errors. This prevents corrected single bit errors from degenerating to uncorrectable multibit errors. OS-based management applications and the platform controller may share this information and avoid use of such memory pages during the next reboot.

It is recommended that the SAL implementation not probe platform error information during an MCA condition unless the PSP.bc bit is set (BusCheck presence). This minimizes the commingling problem between an MCA and a CPE.

§