Machine Check Recovery for Linux on Itanium® Processors

Tony Luck
Intel Corporation
Software and Solutions Group
`tony.luck@intel.com`

Abstract

The Itanium[1] processor architecture provides a machine check abort mechanism for reporting and recovering from a variety of hardware errors that may be detected by the processor or chip set. Simple errors such as single bit ECC may be corrected transparently to the operating system by hardware and firmware, but more complex errors where data has been lost require OS intervention. In cases where the OS can reconstruct the lost data, then execution can continue transparently to the application layer, otherwise the OS may decide to sacrifice affected user processes to allow the system to continue. This paper describes how Linux can recover from TLB errors without affecting applications, and also how Linux can recover from certain memory errors at the expense of terminating user processes.

1 Introduction

Server systems are not just about increased speed and capacity. They must also provide better reliability than their desktop and mobile cousins. The Intel Itanium architecture includes a machine check architecture that provides the mechanism to detect, contain and in many cases correct processor and platform errors.

2 Source of errors

There are several sources of errors within a computer system:

1. Electrical supply line fluctuations

   Can be mitigated with a high quality power supply with surge suppression capability.

2. Static electricity

   Effects can be lessened by good enclosure design and special static reducing floor covering.

3. Heat

   Reduced by good thermal design of system enclosure and air-conditioning of the computer room. Hardware may also detect excess temperature and automatically switch to mode where less power is dissipated (e.g. a lower clock speed and/or voltage, or a reduction in the number of available functional units for retiring instructions).

[1] Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Other names and brands may be claimed as the property of others.

4. Interaction with high energy particles due to radioactive decay

   Can be reduced by careful selection of the materials used to build the system (e.g. use of ultra pure dopants consisting of only stable isotopes), and addition of shielding.

5. Interaction with high energy particles from cosmic ray showers in the earth's atmosphere

   Reduce by shielding (locate computer room in the basement) and avoiding high altitude locations for computer systems (intensity in Denver, altitude 5280 feet, is over four times higher than at sea level locations).

Why is this important? Other aspects of hardware are getting more reliable, but as feature size is reduced they become more susceptible to particles (as lower energy particles are capable of flipping bits). Software is getting more reliable too: we cannot just blame crashes on the OS. Clusters of computers multiply the error rates. A mean time between failure of several years for a processor isn't too bad of a problem if you only have one processor, when you build a cluster of several thousand processors, then you have a big problem.

For each of the error sources listed above I have suggested methods by which the error rate may be reduced, but it may be impractical or too expensive to reduce all of these errors to insignificant levels, hence computer systems must be designed to detect, isolate and recover from errors when they do occur.

3 Itanium Machine Check Architecture

The Itanium machine check architecture provides a framework in which diverse types of errors can be handled in a logical and consistent way.

3.1 Error severity

Errors are divided into three categories:

1. Corrected errors are those that are repaired by hardware or by firmware. In either case an interrupt (CMCI[2] for processor errors, CPEI[3] for platform errors) may be raised for logging purposes.

   Examples of this type of error are a correctable single-bit ECC error in the processor cache or a correctable single-bit ECC error on the system bus.

2. Recoverable errors involve some loss of state. They require operating system intervention to determine whether it is possible for the system to continue operation.

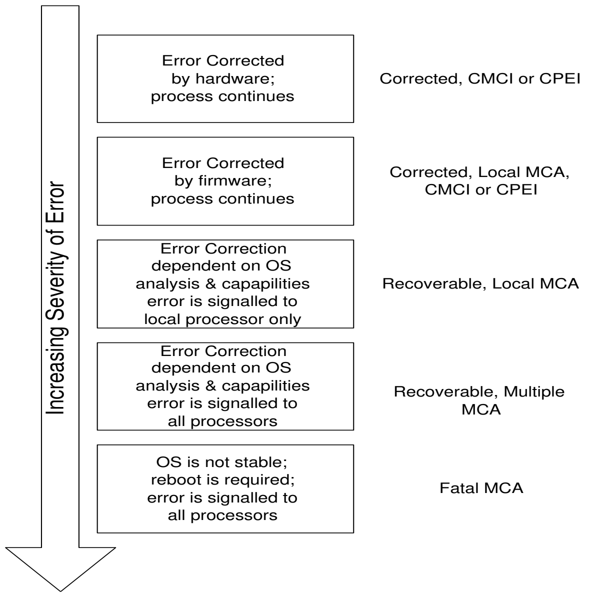
   An example of a recoverable error is one where incorrect data is about to be passed to a processor register (e.g. from a load from memory with a multi-bit ECC error).

3. Fatal errors cannot be corrected. A system reboot is required.

   On this kind of error, the processor generates a signal that is broadcast on the system bus (called BINIT#) that causes the processor to discard all in-flight transactions to prevent error propagation. The error is fatal because there is no way to recover the state of the discarded bus transaction, hence the need for a system reboot.

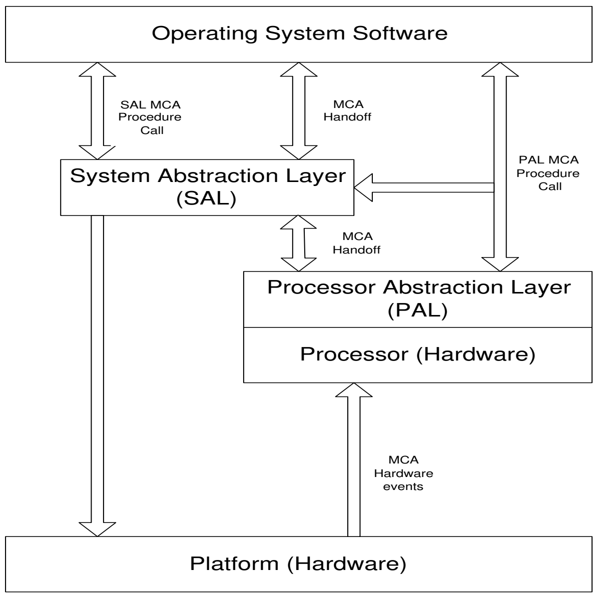
   An example of a fatal error is a processor time-out (when the processor has not retired any instructions after a certain time period).

[2]Corrected Machine Check Interrupt
[3]Corrected Platform Error Interrupt


Increasing Severity of Error
Error Corrected by hardware; process continues
Corrected, CMCI or CPEI
Error Corrected by firmware; process continues
Corrected, Local MCA, CMCI or CPEI
Error Correction dependent on OS analysis & capapilities error is signalled to local processor only
Recoverable, Local MCA
Error Correction dependent on OS analysis & capapilities error is signalled to all processors
Recoverable, Multiple MCA
OS is not stable; reboot is required; error is signalled to all processors
Fatal MCA

3.2 Control flow

This diagram shows how the hardware, processor, processor abstraction layer, system abstraction layer and operating system interact to handle machine checks:



At the hardware level, hardware redundancy (parity and ECC) is used to detect and possibly correct errors during program execution. In some correctable cases the hardware may simply fix the error on the fly and raise a corrected error interrupt to allow logging of the event. In other cases the processor will save all machine state and pass control to the PAL (Processor Abstraction Layer) code at the PAL_MCA entry point for analysis and further processing by firmware. Different members of the Itanium processor family may make different choices about whether to fix errors in hardware or pass responsibility up to firmware.

Entry to the PAL is made in physical mode with caches disabled. This provides the option for the PAL to handle some types of errors detected in the cache, which is useful since on chip caches make up so much of the die area of modern processors that they are statistically one of the most likely forms of errors (from the processor itself…memory errors from a large array of DIMMS are probably the most likely system-wide source of errors). If the error is corrected at this point, then execution can be resumed, but again an interrupt is raised to allow the error to be logged.

Next layer in the firmware stack is the SAL (System Abstraction Layer) which is responsible for components outside of the processor itself (e.g. chip set, memory, I/O bus bridges etc.). The processor is still executing in physical mode (MMU disabled), we may have enabled the caches by this point (depending on whether the SAL machine check entry point is located in cacheable or uncacheable memory). SAL code examines the details of the error and determines whether it can fix it without causing loss or corruption of any data. As above, if the error is fixed, then execution resumes with a pended interrupt to log the details.

At the highest level is the operating system. Errors that have not been corrected at lower levels, but which leave the processor in an internally consistent state, are still recoverable. These can be passed to the operating system if it is interested in trying to recover. An operating system indicates it is capable and willing to handle machine check errors by registering an entry point with the SAL using the SAL_SET_VECTORS call. Note that the operating system entry point must be a physical address because it is possible that the error to be handled is in the memory management H/W, hence the MMU must still be disabled when the SAL transfers control to the operating system entry point. The physical entry point means that the code to service machine check abort must either be position independent, or at least very aware of relocation issues since it will not be executing at the kernel's linked address. Also the SAL_SET_VECTORS call allows the operating system to provide the length and a simple byte checksum of the code so that the SAL may validate that the routine has not been corrupted.

4 Reporting corrected errors

As mentioned above, errors that are corrected by hardware, PAL, or SAL may be reported to the operating system by means of a CMCI (Corrected Machine Check Interrupt) for errors inside the processor, or a CPEI (Corrected Platform Error Interrupt) for system errors outside of the processor. The operating system may choose to disable these interrupts and periodically poll the SAL to see if any errors have been corrected. It might do this to avoid being swamped by corrected error interrupts (e.g. a stuck data line causing hard single-bit memory errors across a wide range of physical addresses, we would like to ensure that the operating system can continue to make forward progress).

Whether the operating system takes interrupts or polls, once a corrected error record is found the OS can retrieve the whole record, parse it to find any useful information, and then output details to its own log. As a final step in error reporting the operating system must request the SAL to clear the error record from non-volatile memory (to ensure that space is available for future errors to be logged).

User level tools could be written to analyze the operating system logs to check for patterns that may be predictive of future hard errors in components that generate a high level of soft errors.

5 Poisoned memory

Another feature of the machine check architecture is the concept of "poisoned memory." This allows a platform the option of deferring error processing in some circumstances. Suppose a modified cache line is being written back to memory when an uncorrectable error is detected in the data contained in the cache line. The current execution state of the processor probably has no connection with this data, so signaling a machine check abort at this time may be an over-reaction to the situation. Instead, the data can be written to memory together with an indication that it is corrupt (the "poison" flag, typically indicated with bad ECC bits). A regular CMC interrupt is then raised, and the OS is allowed to examine the situation later. Deferring the error in this case may be useful, because it is not certain that the corrupted data will ever be needed (e.g., the page to which the cache line belongs may already have been freed by the operating system, or the word that was corrupted in the cache line may have only been present in the cache because of false sharing).

Note that in the "read" case data poisoning does not apply, the processor will immediately begin MCA processing.

If the operating system writer is concerned that the poisoned data may be consumed before the interrupt is processed, there is an option to promote CMCI to MCA to allow immediate action to be taken (though this option applies to all CMCI, not just to those caused by poison data).

6 Operating system examples

Here are some example cases of recoverable errors where the operating system can intervene to recover from an error that has been detected and reported using the above mechanism.

6.1 TLB translation register error

The Translation lookaside buffer in the Itanium processor is not only divided into separate structures for instruction and data access, it is also conceptually divided into two types of entries.

1. Translation cache entries can freely be replaced as new mappings are added.
2. Translation register entries are locked into the TLB. These are used to "pin" translations for critical regions to ensure that a TLB miss will never occur for a virtual address mapped by a locked entry.

Errors in translation cache can be trivially handled by H/W or PAL by simply discarding corrupted entries from the cache. This will only affect performance, if the discarded entry is needed, the processor will simply reload using the normal TLB miss execution path. However, if an error is detected in one of the translation registers, then the fault cannot be handled by F/W, PAL or SAL, since the entry cannot just be dropped and the firmware does not have enough information to reconstruct the damaged entry. So the error is propagated up to the OS which needs to reload the errant register. The ia64 Linux implementation uses the following TR registers:

ITR(0) maps one large kernel page[4] as the kernel text (code)

DTR(0) maps one large kernel page as the kernel data

ITR(1) maps one granule[5] for PAL

DTR(1) maps one page[6] for per-cpu area

DTR(2) maps one granule for kernel stack

The first four of these are loaded during kernel initialization, and are never changed, so it is a simple matter to add code to save the correct values for these registers in memory, so that the MCA handler can reload when needed. The last, mapping the kernel stack, can potentially be reloaded on every context switch (actual reloads occur when the task structure for the new process is in a different large kernel page). The Linux kernel uses one of the supervisor mode registers (ar.k4) to keep track of which large kernel page is currently mapped, and this register can be used to reconstruct the DTR(2) value during the MCA handler.

Although the SAL error record generated for an error in a translation register provides information on which register(s) have errors, it would require a large amount of code to retrieve and parse the error record to determine exactly which registers need to be reloaded. All this code would have to run in physical mode (remember that SAL passes control to the operating system MCA entry point in physical mode, and it is not possible to transition to virtual mode for this particular error, since we know that one or more of the translation registers are corrupt). The simple solution to this issue is to have the MCA TLB recovery code purge and reload all of the TR registers in the

[4]Kernel is always mapped with a single 64MB page

[5]another type of large kernel page, configurable as either 16MB or 64MB

[6]PAGE_SIZE on 2.4, 64k on 2.5

physical mode code. Then we can safely transition to virtual mode to retrieve and examine the record from the SAL error log to report the actual register(s) that were affected by the error.

6.2 Multi bit ECC error in memory

In this case some data has been irretrievably lost, so the operating system cannot escape from this situation unscathed. The basic strategy for the OS is to identify the address of the memory that reported the error. If the memory is owned by the kernel, this will currently be reported as a fatal error by the Linux MCA handler and the OS will reboot (to be strictly accurate the Linux MCA handler will return to SAL with a request to reboot, and the error will be reported by Linux after the reboot when the SAL error record is retrieved from NVRAM). If the memory is allocated to one or more user processes, the processes can be sacrificed to allow the system to continue running (just as the OOM killer will terminate processes when Linux runs out of memory). Life is rarely that simple. In this case a complication is that machine checks are not reported synchronously to the instructions that trigger them, they may be deferred for a long time (in the worst case until the values are consumed). The processor precisely identifies the location at which the fault is detected, but does not provide information about the point at which the fault occurred. The processor does not automatically [7] raise machine checks across privilege transitions from user to kernel mode and vice versa, so it is possible that an error caused in one privilege state will be reported in the other state. Easy cases:

a) fault triggered in user mode is reported in user mode—kill process

b) fault triggered in kernel mode is reported in kernel mode—system reboot

Harder cases:

c) fault triggered in kernel mode is reported in user mode—this is a very subtle case. At first sight it appears that our error handler cannot do the right thing. This case is indistinguishable from case ‘a’ above, since we only have precise information about where the error was detected. But if the error occured in some kernel data, just killing the process is not the correct action. It would leave the kernel running with a corrupted data structure! However, we are saved by the fact that the error is detected when the user process tries to consume the data, and we know that only a buggy kernel would leak details of a kernel data structure to user mode. So we can assume that any such error that happened in kernel mode and was detected in user mode must have occurred when the kernel was restoring registers that belonged to the user process. Thus killing the process is sufficient to contain the error.

d) fault triggered in user mode is reported in kernel mode—sadly this will cause a reboot because it isn’t possible to distinguish this from case ‘b’ above (though it might be possible to eliminate many of these cases by a special case for faults reported during the code that saves user registers).

6.3 PCI errors

The Itanium processor family machine check architecture provides a framework for reporting platform errors, such as PCI bus errors. From an OS perspective, these may be far more complex to handle. Issues are:

[7] There is a PAL call PAL_MC_DRAIN to do this, but it would be a major performance issue to use this on every transition

1. We would like a framework that is minimally invasive to the existing driver model.
2. Linux is just starting to get support for hot-add and hot-remove of devices, but it is a big step from there to support surprise removal of devices when errors occur.
3. Even in the case of transient errors, recovery may be complicated by the firmware on the card, which typically has been written with an expectation that the system will reboot after an error.

7 Acknowledgments

Thanks to all the people at Intel who spent time reviewing this paper and providing invaluable feedback.

References

[Menyhárt] Z. Menyhárt and D. Song, *OS Machine Check Recovery on Itanium Architecture-base Platforms*, Intel Developer Forum, Fall 2002

[Ziegler] J.F. Ziegler, *Terrestrial cosmic ray intensities*, IBM Journal of Research and Development, Volume 42, Number 1, 1998

[SDV] Intel, *Intel Itanium Architecture Software Developer's Manual, Volume 1–3*

[EHG] Intel, *Itanium Processor Family Error Handling Guide*, August 2001

[SAL] Intel, *Itanium Processor Family System Abstraction Layer (SAL) Specification*, November 2002

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada