Machine check handling on Linux

Andi Kleen
SUSE Labs
ak@suse.de

Aug 2004

Abstract

The number of transistors in common CPUs and memory chips is growing each year. Hardware busses are getting faster. This increases the chances of data corruption by arbitrary bit flips in hardware. Modern chips can detect and sometimes correct such events using ECC checksums and other techniques, but there are cases the hardware can't hide such problems completely and software has to handle it. Such an event is called an machine check (MC).

As these events become more common, it's becoming more and more important that Linux recovers as well as possible from them.

The paper discusses some generic issues in handling MCEs in software and covers the new recently rewritten x86-64 machine check handler.

1 What is a machine check

This paper is about machine checks. A machine check is the hardware's way to tell you about some internal error. Traditionally when something goes wrong in hardware the machine crashes. With a machine check, software has a chance to do something better.

The main focus is the x86/x86-64 platform, in particular on the AMD Opteron (because that is the platform the author has most experience on with machine checks). Most of the discussion applies in general terms to non-x86 architectures too, although some details can differ.

There are two main kinds of machine check: machine check exceptions (MCEs) and silent machine check. A machine check exception happens when there is an error that the hardware cannot correct. It will cause the CPU to interrupt the current program and call a special exception handler.

With a silent machine check the hardware was able to correct the error, but logged the event to internal registers. There the event can be read by the operating system or the firmware later. Silent machine checks don't need immediate software or administrator action, but it is useful to log and analyze them to get early cues about hardware problems.

2 Why are they important?

Modern hardware has internal self checking, like internal checksums and error detecting and correcting codes for caches and busses. But the number of transistors is growing and feature size is shrinking with each chip generation, which both increases error rates.

Clustering Linux machines into clusters for high performance scientific computing becomes more and more popular[beowulf]. In these clusters it is important to gather information about machines failing so that corrective action can be taken by the administrator. With a lot of machines the mean time between failures is significantly decreased, which means error handling becomes more important.

When an hardware error occurs on a node the task that ran on it should fail to prevent silent errors from being introduced into the computation. One way to detect these problems would be self checks in the software (like checksums over memory buffers or algorithms with internal sanity checks for results). But this is not always possible and requires a lot of effort from the programmer. Another way is to rely on the hardware error detection. When the kernel logs an uncorrected hardware error the cluster software can take corrective action, like rerunning the task on another node and reporting the failure to the administrator.

The same issues apply on servers and high availability clusters.

Logging hardware errors makes it possible to predict failures early.

Even on a desktop silent errors should be avoided. It is better to tell the user that something went wrong due to a hardware issue instead of silently giving wrong results or crashing randomly.

Sources of machine checks can be the CPU, PCI IO[1], memory, caches, internal busses. The errors can be corrected errors (only logged to registers, no exception) or uncorrected errors (exception happens, software must react).

When PCI IO errors are enabled machine checks could be also caused by software bugs in drivers[2]

3 A quick overview of the x86 machine check architecture

The original IBM PCs had parity memory and caused Non Maskable Interrupts (NMIs) when a memory error occurred. Later PCs dropped parity memory, but still reported some hardware errors

Then with the Intel Pentium, basic machine check handling was added to the CPU again. With the Pentium Pro Intel defined a new generic x86 machine architecture[intelsys]. This architecture is implemented by modern x86 CPUs from Intel and AMD. It consists of a standard exception (interrupt 18) for machine checks and some standardized Machine Specific Registers (MSRs). The common registers allow software to check if an machine check occurred, to enable and disable them, check whether the error was corrected or corrupted the CPU state and some other things.

In addition there are some more registers for each bank. A bank is a group of errors generated by a specific subsystem (like CPU, bus unit, cache, north bridge). The number and meaning of banks is CPU dependent.

Each bank has a number of sub-errors that can be enabled or disabled individually. Normally a generic machine check handler enables all errors and all banks [3] A machine check bank also has a register for the address associated with the error.

Some CPUs like the Intel Pentium 4 also have extensions over the standard registers[intelsys].

The advantage of this generic architecture is that a single machine check handler can work on many different CPUs. When an machine check is detected, the kernel

---

[1] Normally not enabled on current PCs.

[2] But this normally isn't the case on current x86 machines.

[3] Sometimes some particular machine check errors are not reliable in hardware, but it is assumed that the BIOS disables the broken ones.

reads all the generic machine check registers and the registers from any banks that signaled an error.

The actual decoding and interpretation of the different errors is CPU dependent and up to the user. Some generic handling can be done; for example when a bank has a valid error address, the handler can assume that the memory at this address got corrupted. Also the handler can take different action depending on if the error was corrected or not and if the error corrupted the CPU context.

Modern Intel CPUs have special thermal errors that happen when the CPU overheats and gets throttled. This normally only needs to be logged.

Some chipsets can be configured to trigger NMIs on various PCI or other bus errors.

4 Why is it hard to write a machine check handler

Cannot use any normal kernel services. Normally kernel code can be in process context or in interrupt context. Interrupt context can do less than process context; it can only call functions that properly protect their data structure against parallel occurring interrupts. Such "safe" functions are called "interrupt-safe".

Machine check exceptions can trigger all the time, even in a critical section when all normal interrupts are disabled. This implies that the machine check handler cannot even use interrupt-safe functions, otherwise it would risk deadlocking on kernel spin locks.

For silent machine checks, undefined interrupt state isnt a problem because they normally run from the timer interrupt, which honors normal interrupt exclusion rules. However to make the code simpler the silent checking and the exception handling share the same code paths, which means that these problems apply to some extent to the silent event check too.

It is also important to handle the machine check quickly (because the machine may be already unstable after an hardware failure). When the handling is delayed to bring the kernel into a easier to handle state first there is a risk that the event cannot be handled at all. Also when another machine check occurs on the same bank in this time window it would overwrite the old event and become un-handleable.

For more complex event like an RAM error however there may be no other choice than to delay handling, because they must synchronize with kernel locks.

Unlike other exceptions, machine checks are asynchronous. This means the CPU core does not take care of reporting them at the exact instruction that caused the failure, but they may be reported only hundreds of cycles later. This makes handling less reliable as discussed below.

5 Logging machine checks

Traditionally machine checks were logged by the firmware[4]. When the operating system does not have an machine check handler, the MC registers will never be cleared. After the next warm boot[5] the BIOS finds the information from the last machine check and logs it to an event log. This method has obvious shortcomings: the logging only happens when the machine is rebooted, it cannot log multiple errors in the same bank and it is hard to collect this information in a network or save it to disk.

---

[4] BIOS on PCs.

[5] On IA64 and PPC64 machines this can be handled without rebooting by PAL code or the hypervisor.

Moving the logging to the operating system can avoid all these problems. Even then it is still difficult. Most Linux users have the X server running and the console is invisible. This means that the handler could log a fatal machine check, but the user wouldn't see it and just see a frozen X. One way to avoid this right now is to log the error after the next reboot only. This also allows us to save it to disk, which makes it possible to check it later by support personnel.

It is important to clearly separate machine check logs from other software errors (like oopses). Most users cannot distingush them and they will ask their software vendors about it, when they should really contact the hardware vendor. Experience has shown that the only good way to do this is to separate the log mechanisms completely.

6 x86-64 rewrite

The original x86-64 machine check handler in Linux 2.4 was derived from the i386 version, which was originally written by Alan Cox. One of the first enhancements was a text decoder of the AMD Opteron specific banks[6], in particular for memory errors. This decoder code unfortunately had a few bugs and turned out to be a design mistake. It is better to do such decoding in user space.

The handler also had a few problems inherited from the i386 version, in particular it used printk directly from the handler, which could deadlock. There were also a few other problems, which triggered a rewrite from scratch of the x86-64 handler during Linux 2.5.

It closely follows the recommendations given by Intel[intelsys] and AMD[amdsys] for machine check handlers. One important change is that the new handler makes some attempts to distinguish between uncorrected errors and errors that corrupt the processor context. In the first case only the process is killed when it is safe. The old handler would always panic.

This killing can be slightly risky when the process was in kernel mode, because it could have been holding locks and that deadlock on process exit.

By default the kernel will always panic on a MC in the kernel to avoid this deadlock. The rationale is that a panic can be handled better than a deadlock, especially in a cluster. [7]

The new handler doesn't have any CPU specific code any more [8], it handles everything using the generic x86 machine check architecture.

It has a new lockless binary logging system. All machine check events (silent and exceptions) will be logged to a special buffer. This buffer isn't a ring buffer, if the buffer fills up new entries are discarded. It is completely separated from the normal printk log and can be accessed from user space using the /dev/mcelog character device. This device should be read in a regular cronjob by the mcelog[mcelog] program. mcelog decodes the event and logs it into a special log file. It could also notify administrators about the event.

The log buffer also has a special signature in memory that could be used by an external debugger or special firmware to look for hardware errors after reboot.

On a panic the bank causing the fatal error is not cleared to allow firmware or the kernel to log the error after an warm reboot to permanent storage[9].

It has a regular polling timer that reads silent machine checks and logs them.

---

[6] The x86-64 port ran only on Opteron at this time so it was a natural idea.

[7] This heuristic is not completely reliable right now due to the asynchronous nature of machine checks.

[8] Except for one BIOS bug workaround

[9] As of 2.6.8 this feature is not in mainline yet

```
/* A machine check record */
struct mce {
        __u64 status;   /* bank status register */
        __u64 misc;     /* misc register (always 0 right now) */
        __u64 addr;     /* address or 0 */
        __u64 mcgstatus; /* global MC status register */
        __u64 rip;      /* Program counter or 0 for silent error */
        __u64 tsc;      /* cpu time stamp counter */
        __u64 res1;     /* for future extension */
        __u64 res2;     /* dito. */
        __u8  cs;       /* code segment */
        __u8  bank;     /* machine check bank */
        __u8  cpu;      /* cpu that raised the error */
        __u8  finished; /* entry is valid */
        __u32 pad;
};
```

7 Configuring the new x86-64 handler

The new handler can be configured at system run time by reading or writing the control files in */sys/devices/system/machinecheck/machinecheck0/* [10] Valid fields are:

- *tolerant* Tolerance level. The higher this level the more risk the machine check handler takes to keep the machine running.

  Valid levels are:

0	always panic on uncorrected errors.
1	panic if deadlock possible
2	try to avoid panic at slight deadlock risk
3	never panic or exit (for testing only)

  Specifying oops=panic on the kernel command line implies zero tolerance.

  For a cluster setting tolerant to zero may be best, together with panic=10 to force an reboot.

- *check_interval* Interval in seconds to check for silent machine check events. Default 5 minutes. 0 disables background checking.

- *bank0ctl ... bankNctl* Binary mask of errors enabled in bank N. Default is to enable all errors in each bank. An disabled error will be ignored. For details on the banks and their sub-errors for AMD and Intel CPUs see [opteron] and [intelsys].

8 Future work: New RAM/cache error handling

RAM errors are the most common sources of machine check events. The memory controller runs asynchronously from the CPU core, which results in errors getting reported imprecisely. The MCE handler assumes that the error occurred in the process that was active at exception time and checks if it was in kernel or user mode. It then uses this information to decide which process to kill or if it should panic. When the error happened shortly before a kernel call or a context switch this information may be stale. A more reliable alternative would be to use the physical

---

[10] machinecheck0 applies to all CPUs in the system.

error address provided in the MCn_ADDR register and use VM data structures to look up to which process the memory address belong. This could be multiple processes for shared memory.

The handler would first need to synchronize to process state because VM locks are not interrupt safe. It could first go into interrupt context by forcing an self interrupt[11] on the same CPU (this would delay execution to the next local interrupt enable and a standard interrupt context). Then this interrupt handler could set up a work queue item to run a callback in one of the event processes on the local CPU.

This callback could use the mem_map and rmap data structures offered in Linux 2.6 to look up the owner of the failed page. There are various cases to distinguish in the kernel page cache:

Free page	Ignore and clear error
Clean page	Free page and reread contents from disk
Dirty page	Kill process owning or force IO error for unmapped file cache data
Kernel page	Panic or kill process depending on tolerance level

This approach could also be possible to handle uncorrected cache errors. In the future it may be also possible to give an application a chance to react to a machine check error by sending it a signal with the failed address as payload instead of unconditionally killing it[12]. The program could then decide how to handle the corrupted memory. For example a database server with a lot of data cache which is backed by the disk could just drop a corrupted cache page and reread it.

9 Future work: Handling IO errors on PCs

Some non-PC platforms like HP zX or IBM PPC64 chipsets raise machine checks on PCI IO bus aborts. On PCs these errors are normally silently ignored. Some chipsets can be configured to raise an NMI in this case. It would be possible to write chipset specific drivers that look up the PCI bridge error registers on NMI and try to figure out what device caused the error. Then disable the PCI device to prevent further corruption[13] and log an error to the user. This would be useful for driver debugging and could potentially protect the kernel against failing PCI cards. To do the latter properly it would also need a full IOMMU.

PCI Express[pcie] has an optional but standardized advanced error report capability in its bridge configuration space that may be useful here.

One problem is that there is no well-defined way to find the source of an NMI because it is used by other subsystems like oprofile.

Another problem is that changing the PCI bridge programming of the firmware (e.g. to enable additional error reporting using NMI) has always some risk.

Still it might be worth it because handling PCI errors better could potentially increase Linux/x86 reliability longer term. Short term it would uncover some more driver bugs, although many of those should be already fixed from testing on PPC64 and IA64. I am not quite sure it will be possible to implement this generally on PCs, but it would be at least worth a try.

10 Work to do

NMI handling is still broken. Currently it reads some IO ports and handles them based on what they did on IBM AT, which is not very useful anymore on modern machines. It also still uses printk and should use the lockless logging framework.

---

[11] This may require using APIC mode or alternatively put a check into the timer handler

[12] The MCE cannot currently be caught by a signal handler.

[13] On the HP and IBM workstation chipsets this is done by firmware

Add a proper thermal handler on x86-64.

The improved x86-64 machine check handler should be ported to x86.

Mcelog doesn't decode AMD Opteron specific errors so far. Currently it only dumbs the registers as hexadecimal. It would be more user friendly to show the individual banks as text, with the various error bits decoded. Intel decoding support should also eventually be added.

11 Acknowledgments

The original i386 Linux MCE handler was written by Alan Cox. Dave Jones and Zwane Mwaikumbo also worked on the i386 handler in Linux 2.5. Paul DeVriendt, Richard Brunner, David Boles gave valuable feedback on the x86-64 rewrite and helped me to understand the problem better. Special thanks to Eric Morton for a lot of review and patches of the code and to Randy Dunlap for reviewing this paper.

References

[beowulf] Becker Donald, Sterling Thomas *Beowulf: A Parallel Workstation For Scientific Computation*

[intelsys] Intel corporation

IA-32 Intel Architecture Software Developer's Manual Volume 3:

System Programming Guide

http://developer.intel.com

[opteron] AMD

BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors

http://developer.amd.com

[amdsys] AMD

AMD64 Architecture Programmer's Manual Volume 2: System Programming

http://developer.amd.com

[mcelog] Kleen

mcelog utility tarball

ftp://ftp.x86-64.org/pub/linux-x86_64/tools/mcelog/

[pcie] PCI SIG

PCI Express Base Spec

http://www.pcisig.org