## Contents



# What is MCA/MCE about

The machine-check mechanism allows the processor to detect and report a variety of hardware errors. Software can enable the processor to report machine-check errors through the machine-check exception (MCE). Most machine-check error conditions do not allow reliable restarting of interrupted programs. System software instead uses the machine-check mechanism to report the source of hardware problems for possible servicing.

MCA stands for Machine Check Architecture.
MCE stands for Machine Check Exception.

## Developer's point of view

With MCA enabled, the CPU informs Xen about hardware errors. Xen installs an exception handler triggered by an MCE. It handles the MCE. Linux dumps the machine check MSR values on the console. Use mcelog to decode them.

## System Administrator's point of view

System administrators do not know what a machine check exception is. When Linux reports an MCE they have almost no idea what went wrong. Most sysadmins also do not know mcelog. Futher, if Dom0 is a Solaris system, there are other tools than mcelog.

The sysadmin relies on the Dom0 to do some analysis and tell him what has happened.

# Machine Check Architecture en détail

## AMD

Machine check support has been inherited from Intel and appeared in AMD K7 CPUs the first time.

The machine check errors are reported via MSRs. There are three common global MSRs. The global MSRs enable/disable the error reporting in general, provide capability information (i.e. number of available banks) and keep the information if the instruction pointer is related to the error and if the machine check handling is in progress (MCIP). There are also five banks each describing a class of errors:

```
0. Data Cache
1. Instruction Cache
2. Bus Unit
3. Load/Store Unit
4. Northbridge
```

A sixth bank has been introduced in AMD Family10h:

```
5. Fixed Issue Reorder Buffer
```

Each bank consists of a group of four MSRs. The first in the group is the control MSR. It allows to enable/disable the reporting of certain errors. The second MSR contains the status. It contains the bits if an error occurred in this class and if the third or fourth MSRs contain valid values.

The third MSR contains the address of the failure. The value read out of the MSR is only valid if the status MSR reports so.

The fourth MSR contains misc information. It contains model specific error codes which may vary even between the CPU models. Example: In K8 NPT, the northbridge misc MSR contains some additional bits that were reserved before.

With the introduction of the AMD K8, the northbridge registers have been moved to the PCI config space. The MSRs are aliases for those registers.

In K8 NPT hw thresholding has been added to the northbridge misc MSR. It counts the number of correctable errors. When the counter reached a certain limit, it raises an interrupt if enabled in the LAPIC.

In Family10h three new northbridge registers have been added. These report more detailed information on the DRAM, on the hypertransport link and on the L3 cache. These registers are also mirrored three new MSRs in the northbridge MISC.

# Intel

The P6 processor has the five banks:

```
0. Data Cache
1. Instruction Cache
2. Bus Unit
4. Load/Store
3. Northbridge
```

The Pentium 4 and Xeon Processors additionally the a sixth bank:

```
5. unknown
```

Note, that Intel counts differently than AMD. Additionally, the Pentium 4 and Xen Processors provide one MSR for each GPR, stack pointer, instruction pointer and flags.

According to Intel, those can be used by a debugger or a handler routine to analyze error using the register values.

# NMI vs. MCE

NMIs and MCEs have the same priority from the hardware. This is a fatal hardware bug by design as they can interrupt each other.

The concequences:

When a MCE interrupts a NMI, then iret at end of MCE handler unblocks further NMI and allows NMIs to become nested.
When a NMI interrupts MCE, then NMI handler can do something causing another MCE.

The MCE handler to the point where it clears the MCIP is the critical section. It reads the error telemetry and any interruption may malform the error telemetry.

A NMI can detect if it interrupted a MCE by reading the MCG_STATUS MSR and checks if MCIP (bit 2) is set. As a NMI does not know what caused the MCE and should assume the worst case: its memory accesses and instruction it runs may generate another MCE. It has to return immediately.

An effictive workaround is to suppress NMIs by disabling the LAPIC on the core running the MCE. However, this stops the LAPIC timer and causes several problems like "Timer went backward".

## Machine Check Context

The CPU is in machine check context when the MCIP (bit 2) in MCG_STATUS (global status) MSR is set. The CPU sets the bit when it throws a machine check exception. Software is responsible to clear the bit at the end of the exception handler.

If a MCE occurs while the CPU is in machine check context, the CPU enters shut down state. A NMI, RESET or INIT can wake up the CPU again.

# Cooperation with Sun

Sun offers a "Fault management" in Solaris. While porting Solaris to Xen, they figured out, MCA/MCE support in Xen misses requirements for high availability. A talk between Sun and AMD resulted in a rough concept how MCA/MCE should work in Xen.

Details like how to notify the Dom0, which format will Dom0 use (native or abstracted) and how Dom0 will tell Xen what to do in the error case still needs to be worked out.

mcelog for Linux is only partially useful. Sun is interested in running Solaris as Dom0 rather than Linux. Andy Kleen from Novell/SuSE has been asked if he wants to relicense mcelog to BSD or MIT license. He would agree on GPL/CDDL dual licensing it, but has to ask Novell and Intel for permission. Sun might be interested in using it for the case of running Solaris on a machine, Solaris' own "fault management" does not support all MCA/MCE features supported by the machine. Nothing heard from Andy Kleen on this topic.

In February 2008 feedback from Sun were received that they started to use the Xen machine check patch for testing purposes. The results look promising.

There is also good feedback from Sun:

```
The greatest rewards here are in syndrome/row/column/bank analysis of the error stream.
Where something like a bad pin produces tonnes of CEs they are always on the same bit
and your chance of a UE is that of a random radiation type CE colliding within the set
of ECC checkwords being undermined by that pin - not very high. On the other hand if
we're seeing repeated distinct syndromes from the same chip-select (or chip-select in a
pair) then there is a good chance they could collide "soon" - our data is that this
combination predicts a UE within hours to a few days. If you have row/column/bank
decoding you can also perform further analysis of the error source and assess the
chances of a collision that would produce a UE.
```

# The concept

The Dom0 is the privileged domain and the DomU are managed from there. Xen collects the machine check error data and the Dom0 gets them in first place for analysis and error handling.

The Dom0 gets all correctable errors for statistical error analysis. The statistical evaluation is be used to arrange for error handling before non-critical error turns into critical errors. For example, if one certain memory page creates more and more correctable errors it may be supposed to produce an uncorrectable error soon. The Dom0 can replace this page with a healthy one using the online-spare RAM feature (hw feature) or relocating the data (sw feature) or something else depending on the decision.

The Dom0 can do some operations on a DomU at any time. It can live migrate the DomU to a different physical Xen host or save/resume the DomU for example - whatever is most appropriate in the situation.

The Dom0 gets immediately informed about uncorrectable errors. The Dom0 can handle this. It is possible that a valid copy of the data in error is maintained in software. In that case, the uncorrectable error can be turned into an correctable error by turning off the page in error.

The Dom0 can notify the DomU about the error and let the DomU handle the error. If the DomU is capable to handle uncorrectable errors it can kill the impacted process inside the DomU. It is always better to kill a solitaire game than the whole guest.

If the DomU is not capable to handle uncorrectable errors, the Dom0 can kill the DomU, tell Xen to no longer use the memory page in error and restart the DomU.

The DomU will never see correctable errors. If a DomU tries to install an event handler, Xen rejects this with an error.

Both Dom0 and DomU can also use software-only self-healing techniques such as Memory Page Retirement (MPR).

# The demands

In cooperation with SUN, the following demands have been worked out:

Must-have:

```
1. Dom0 and DomU register machine check trap callback handlers
   (already done via "set_trap_table" hypercall)
2. Dom0 registers machine check event callback handler
   (doable via EVTCHNOP_bind_virq)
3. Dom0 and DomU fetches machine check data (requires new hypercall)
4. Dom0 wants Xen to notify a DomU (requires new hypercall)
5. Dom0 gets DomU ID from physical address
6. Dom0 wants Xen to kill DomU (already done for "xm destroy")
```

Nice-to-have:

```
7. Dom0 wants Xen to deactivate a physical CPU (in corrupted state).
   This is better done as separate task as physical CPU hotplugging.
   CPU hotplug hypercall(s) should probably be Xen sysctl
   (special sort of hypercalls for Dom0 only).
8. Page migration proposed from Xen NUMA work, where Dom0 can tell
```

```
   Xen to move a DomU (or Dom0 itself) away from a malicious page
   producing correctable errors.
 9. offlining physical page: Xen frees and never re-uses a certain
   physical page.
10. Testfacility: Allow Dom0 to write values into machine check MSRs
   and tell Xen to trigger a machine check
```

The items are considered as usecases mostly from the Dom0's point of view.

# The guests

The Dom0 installs an event handler which behaves like an interrupt when a correctable error occurred. The Dom0 and DomU use the set_trap_table hypercall to install the machine check fault handler. It behaves like an exception when an uncorrectable error occurred.

Guests not installing a handler are considered as not being able to handle the error and/or not having machine check support implemented.

It is possible to have PV drivers also in HVM guests. Those HVM guests featuring a PV driver are considered to be able to handle the error.

# Rules for Dom0

The machine check architecture has some drawbacks in the design. Those causes conflicts between Xen and Dom0. In order to avoid them, the Dom0 has to keep some rules in mind:

```
* Never touch the machine check registers
* Always use the hypercalls
```

When guests tries to interfere with using MSRs, the CPU throws an exception Xen catches and disallows access to them by injecting the exception into the guest.

However, some northbridge MSR registers moved to the PCI config space in AMD K8. Since then it is possible that the Dom0 can interfere with Xen machine check settings by writing into the registers via PCI mmio. There is no #PF and no #GP Xen can catch and prevent the Dom0 to reconfigure the registers.

# The data structures

The data structure is designed to contain any value of any MSRs. To achieve this flexibility, there is one big container used with three substructures:

```
#define MC_TYPE_GLOBAL          0   /* struct mcinfo_global */
#define MC_TYPE_BANK            1   /* struct mcinfo_bank */
#define MC_TYPE_EXTENDED        2   /* struct mcinfo_extended */

struct mcinfo_common {
  uint16_t type;      /* structure type */
```

```
  uint16_t size;      /* size of this struct in bytes */
};
```

The substructures must use this common structure as a kind of header. *type* is one of the structures types global, bank or extended. *size* is the size of the substructure in bytes.

The first substructure:

```
#define MC_FLAG_CORRECTABLE     (1 << 0)
#define MC_FLAG_UNCORRECTABLE   (1 << 1)

/* contains global x86 mc information */
struct mcinfo_global {
  struct mcinfo_common common;

  /* running domain at the time in error (most likely the impacted one) */
  uint16_t mc_domid;
  uint32_t mc_socketid; /* physical socket of the physical core */
  uint16_t mc_coreid; /* physical impacted core */
  uint16_t mc_core_threadid; /* core thread of physical core */
  uint16_t mc_vcpuid; /* virtual cpu scheduled for mc_domid */
  uint64_t mc_gstatus; /* global status */
  uint32_t mc_flags;
};
```

This structure describes common error characteristics. There is usually one entry of this structure type in the container.

*mc_domid* is the id of the scheduled domain at error detection. On an uncorrectable error, this is very likely the impacted guest, on an correctable error this may be the impacted guest.
*mc_socketid* is the physical CPU socket of the CPU core in error.
*mc_coreid* is the physical CPU core in error.
*mc_core_threadid* is the physical CPU core thread in error. On AMD this is always 1.
*mc_vcpuid* is the virtual cpu scheduled for mc_domid
*mc_gstatus* is the value of the global status MSR
*mc_flags* is either MC_FLAG_CORRECTABLE or MC_FLAG_UNCORRECTABLE. Its purpose is mostly for verification.

The second substructure:

```
/* contains bank local x86 mc information */
struct mcinfo_bank {
  struct mcinfo_common common;

  uint16_t mc_bank; /* bank nr */
  uint16_t mc_domid; /* Usecase 5: domain referenced by mc_addr on dom0
                      * and if mc_addr is valid. Never valid on DomU. */
  uint64_t mc_status; /* bank status */
  uint64_t mc_addr;   /* bank address, only valid
                       * if addr bit is set in mc_status */
  uint64_t mc_misc;
};
```

This structure contains error data specific for one bank. If there is error data for more than one bank, you find multiple entries of this structure type in the container.

*mc_bank* is the bank number of the error class. 0 is data cache, 1 is instruction cache, etc.
*mc_domid* is the domain id the memory address in error belongs to. This is information for dom0 only and therefore only valid when data is fetched from dom0 and if mc_addr is valid.
*mc_status* is the status field of the bank. It contains information if mc_addr and mc_misc contain valid information.
*mc_addr* is the memory address in error. It is only valid if the address valid bit is set in mc_status.
*mc_misc* is additional error data. It contains cpu and cpu model specific error codes. It is only valid if misc valid bit is set in mc_status.

The third data structure:

```
struct mcinfo_msr {
  uint64_t reg;   /* MSR */
  uint64_t value; /* MSR value */
};

/* contains mc information from other
 * or additional mc MSRs */
struct mcinfo_extended {
  struct mcinfo_common common;

  /* You can fill up to five registers.
   * If you need more, then use this structure
   * multiple times. */

  uint32_t mc_msrs; /* Number of msr with valid values. */
  struct mcinfo_msr mc_msr[5];
};
```

This structure contains values of additional MSRs which do not fit into the struct mcinfo_bank and are new in certain CPU families / models. For example, in Family10h, there are three new MISC MSRs and Intel has one MSR for each register additionally.

This structure can contain information about five MSRs at maximum. There are multiple entries of this structure type in the container, in case the limitation is insufficient.

This way, it is possible to have support for any machine check MSR with a common interface.

mc_msrs is the number of MSRs filled with valid data in the mc_msr array. mc_msr is the array containing the MSR register and the MSR value.

If the mc_msrs is less than five it is very likely there are no more entries of this structure type, but it is wrong to assume this.

# The container

The data structures are all stored in one container. With the hypercall you only fetch the whole container, never a structures itself. The container is this:

```
#define MCINFO_HYPERCALLSIZE    1024
#define MCINFO_MAXSIZE          768
```

```
struct mc_info {
  /* Number of mcinfo_* entries in mi_data */
  uint32_t mi_nentries;

  uint8_t mi_data[MCINFO_MAXSIZE - sizeof(uint32_t)];
};
typedef struct mc_info mc_info_t;
```

There are some helpers for simple access to the structures within the container:

```
x86_mcinfo_nentries()
x86_mcinfo_first()
x86_mcinfo_next()
x86_mcinfo_lookup()
```

x86_mcinfo_nentries() returns the number of structures in the container.
x86_mcinfo_first() returns a pointer to the first structure.
x86_mcinfo_next() returns a pointer to the next structure.
x86_mcinfo_lookup() returns the first entry of a certain structure type in the container.

Here is an example code snippet on how to iterate over all structures in the container:

```
struct mc_info *mi;  /* the container */
int i;
struct mcinfo_global *mg;
struct mcinfo_bank *mb;
struct mcinfo_extended *ms;
struct mcinfo_common *mic;

...

mic = x86_mcinfo_first(mi);
for (i = 0; i < x86_mcinfo_nentries(mi); i++) {
        switch (mic->type) {
        case MC_TYPE_GLOBAL:
                printf("xen_mca: found mcinfo_global\n");
                mg = (struct mcinfo_global *)mic;
                ...
                break;
        case MC_TYPE_BANK:
                printf("xen_mca: found mcinfo_bank\n");
                mb = (struct mcinfo_bank *)mic;
                ...
                break;
        case MC_TYPE_SPECIAL:
                printf("xen_mca: found mcinfo_special\n");
                ms = (struct mcinfo_special *)mic;
                ...
                break;
        default:
                printf("xen_mca: unknown type %x\n", mic->type);
                break;
        }
        mic = x86_mcinfo_next(mic);
}
```

Note, there is no guarantee the container is terminated with zeros. It is also possible the structures even use the last byte in the container.

# The implementation

## The polling handler

At Xen boot, the CPU machine check capabilities are checked via presence of CR4.MCA and cpuid family. The polling handler is installed on the BSP. The polling handler runs every 30 seconds. It sends an IPI to all physical CPUs so that all CPUs run the handler at the same time. In case of an error, the polling interval is adjusted at runtime. The adjustment is done on the BSP, so the adjustment has an effect on all physical CPUs. If one error is detected, the next polling happens in 15 seconds. If an error is detected again, next polling happens in 7 seconds and so on. Minimum polling interval is 2 seconds otherwise two much cpu cycles get burned and Dom0 must have its chance to evaluate the error telemetry and to take steps. If no error is found, next polling happens in doubled seconds but not in longer than in 30 seconds.

On K8 NPT and on Family10h, the threshold counter is used to determine multiple correctable errors between two polls. Since it is only possible to read out the last error from the MSRs, the error data between the polls is lost. Therefore, polling frequency is increased more aggressively by the factor of number in errors found.

Starting poll interval is 30 seconds:
1 error found: next polling happens in 15 seconds
2 errors found: next polling happens in 7 seconds
3 errors found: next polling happens in 3 seconds
etc.

Minimum polling interval is at 2 seconds. The LAPIC interrupt handler is not used for two reasons.

The first reason is that it is not possible to get an interrupt on each error, which lets miss even more information for the Dom0 to analyze.
The second reason is, it only observes and counts ECC errors only.
The third reason is, the Dom0 may program the LAPIC. So using an interrupt would just interfere between Xen and Dom0.

When no errors are found, polling interval is doubled but not longer than in 30 seconds.

If an error occurred, error data is collected. If the Dom0 installed an machine check event handler, it gets notified otherwise Xen dumps the error data on the console similar as Linux does.

## The exception handler

At Xen boot, the CPU machine check capabilities are checked via presence of CR4.MCE and cpuid family. For each physical CPU an exception handler is installed. The exception handler is invoked by the CPU when an uncorrectable error occurred. The handler runs as interrupt gate.

The handler collects the error data and clears the MCIP bit in the global status MSR. Clearing the MCIP bit is necessary, in case another machine check exception happens. The CPU assumes the machine check handling code itself is corrupted and shuts down itself, otherwise.

The handler performs performs a quick check who is impacted by checking which domain was scheduled. If the idle domain was scheduled then this means the hypervisor itself was running.

If the hypervisor itself is impacted, it dumps error data on the console and calls panic.

If the Dom0 installed an machine check fault handler, Xen invokes Dom0's trap handler in a deferred way. If the Dom0 installed no machine check fault handler, Xen checks if the DomU has one installed. If the DomU installed an machine check fault handler, Xen invokes the DomU trap handler in a deferred way.

"Deferred" means, a softirq is scheduled right after the exception handler finished which wakes up the guest vcpu and runs the guests machine check fault handler.

If the DomU installed no machine check fault handler, then this means no one feels responsible to handle the error.

In this case, Xen checks if the Dom0 is the impacted guest determined by the quick check. If the Dom0 is impacted, then Xen dumps the error data on the console and panics. If a DomU is impacted, Xen kills that DomU.

# Collecting error data

The error data is collected in a global array that is used as storage. The polling and exception handler act as producer and the fetch hypercall as consumer. Several helper functions ensure smp-safe and lockless access to the storage.

The polling and the exception handler iterate over the all machine check banks and read all MSRs containing error data. The K8 exception handler and the polling handler invoke a callback if it is valid.

On Family10h the K8 exception handler and polling handler is reused with the callback used for reading out the values from the introduced MSRs. This way a lot of code is shared and makes it easy to add support for future CPUs.

# The hypercalls

Only two hypercalls are required to fulfill at least the must have requirements: The fetch and the notify hypercall. The hypercalls use a second consumer/producer storage for the guest notification mechanism. This is the notification storage. The notify hypercall acts as producer and the fetch hypercall coming from a DomU acts as a consumer.

This is the data structure passed to the hypercall:

```
struct xen_mc {
  uint32_t cmd;
  uint32_t interface_version; /* XEN_MCA_INTERFACE_VERSION */
  union {
      struct xen_mc_fetch        mc_fetch;
      struct xen_mc_notifydomain mc_notifydomain;
      uint8_t pad[MCINFO_HYPERCALLSIZE];
  } u;
```

```
};
typedef struct xen_mc xen_mc_t;
DEFINE_XEN_GUEST_HANDLE(xen_mc_t);
```

## The fetch hypercall

```
/* Usecase 3
 * Fetch machine check data from hypervisor.
 * Note, this hypercall is special, because both Dom0 and DomU must use this.
 */
#define XEN_MC_fetch            1
struct xen_mc_fetch {
  /* IN/OUT variables. */
  uint32_t flags;

/* IN: XEN_MC_CORRECTABLE, XEN_MC_TRAP */
/* OUT: XEN_MC_OK, XEN_MC_NODATA, XEN_MC_NOMATCH */

  /* OUT variables. */
  uint32_t fetch_idx;  /* only useful for Dom0 for the notify hypercall */
  struct mc_info mc_info; /* the container */
};
typedef struct xen_mc_fetch xen_mc_fetch_t;
DEFINE_XEN_GUEST_HANDLE(xen_mc_fetch_t);
```

The fetch hypercall performs some sanity checks first. If a DomU tries to fetch a correctable error (specified in the *flags* field), it returns -EPERM failure code to the guest. Fetching correctable errors are only allowed for the Dom0.

If the hypercall comes from a Dom0, the error data from the global storage is taken. The fetch_idx contains the index of the global storage array.

If the hypercall comes from a DomU and if the Dom0 installed a machine check fault handler, then it is assumed the DomU has been notified before from the Dom0. The oldest available error data in the notification storage for this DomU is taken.

If the hypercall comes from a DomU and if the Dom0 installed no machine check fault handler, then only Xen could invoke the DomUs machine check fault handler. The error data from the global storage is taken. The fetch_idx is meaningless for the DomU.

If it turns out, the taken error data is invalid, then the hypercall returns with an error to the guest. The *flags* field is flagged with XEN_MC_NODATA.

The error data is considered as invalid when sanity checks figured out, the fetch hypercall was delayed too long from the guest side after notification. During the long delay it may have happened that other errors occurred in the meantime and filled up the global storage. With the XEN_MC_NODATA flag, the guest is notified as being too slow in machine check handling.

If the error data is valid, it gets copied into the guest and returns successfully.

## The notify hypercall

```
/* Usecase 4
 * This tells the hypervisor to notify a DomU about the machine check error
 */
#define XEN_MC_notifydomain     2
struct xen_mc_notifydomain {
  /* IN variables. */
  uint16_t mc_domid;    /* The unprivileged domain to notify. */
  uint16_t mc_vcpuid;   /* The vcpu in mc_domid to notify.
                         * Usually echo'd value from the fetch hypercall. */
  uint32_t fetch_idx;   /* echo'd value from the fetch hypercall. */

  /* IN/OUT variables. */
  uint32_t flags;

/* IN: XEN_MC_CORRECTABLE, XEN_MC_TRAP */
/* OUT: XEN_MC_OK, XEN_MC_CANNOTHANDLE, XEN_MC_NOTDELIVERED, XEN_MC_NOMATCH */
};
typedef struct xen_mc_notifydomain xen_mc_notifydomain_t;
DEFINE_XEN_GUEST_HANDLE(xen_mc_notifydomain_t);
```

The notify hypercall first checks that is it called from the Dom0 only. If called from a DomU, it returns -EPERM. Then the data filled out by the Dom0 is evaluated:

*mc_domid* contains the domain id to notify. *mc_vcpuid* contains the vcpu id in the domain to notify. *fetch_idx* contains the index of the global storage array. The Dom0 is considered to just echo the value from the corresponding fetch hypercall. *flags* is XEN_MC_CORRECTABLE if Dom0 called this hypercall from the event handler and XEN_MC_TRAP if from the fault handler.

If the domain id is the Dom0 itself or the idle domain which represents Xen itself, the hypercall returns -EACCESS. The Dom0 cannot notify itself or the hypervisor.

If the vcpu id is outside of the maximum of supported vcpu in a guest, the hypercall returns -EACCESS.

The hypercall takes the error data from the global array storage from the given fetch_idx and verifies the error telemetry is for the specified domU. If this check fails, the flags field gets the XEN_MC_NOMATCH value and returns to the Dom0.

If the DomU installed a machine check fault handler, it is notified. On failure, the flags field gets the XEN_MC_NOTDELIVERED value and returns to the Dom0.

If the DomU has no machine check fault handler installed, the flags field gets the XEN_MC_CANNOTHANDLE value and returns to the Dom0.

Then the error data is copied into the guest and added to the notification storage (consumer is acting).

The hypercall returns successfully to the Dom0.

## CPU, VCPU affinity and DomU notification

This section assumes, the Dom0 and DomU installed a machine check fault handler.

At the time of an uncorrectable error happens, the physical CPU runs the exception handler and therefore doesn't schedule the VCPU in the guest and the scheduled process in the guest is not running.

No other physical CPU may schedule the impacted VCPU as the guests process may continue running and works with corrupted data and/or cause another uncorrectable error on the other physical CPU.

To ensure the guests VCPU is not scheduled until Xen's machine check exception handler finished, the VCPU must not rebounded on an other physical CPU nor to an other DomU.

Then the DomUs impacted process is still "halted" because the impacted physical CPU runs the Dom0 machine check fault handler.

Then on DomU's notification, it is necessary to trap the guests machine check fault handler of the impacted VCPU since this is the only way to interrupt the guests impacted process outside of the guest. If any other VCPU gets trapped, then the impacted process continues running and works with corrupted data. The DomUs machine check fault handler running on the impacted VCPU does not let run the impacted process until it finished.

# MCA/MCE support for HVM guests

Although MCA/MCE support must be reported to HVM guests to be available for HVM guests to let Windows XP boot, it has been decided to **not** multiplex MCA/MCE MSRs for HVM guests except for the threshold registers.

The problem with this is, even if the guest set up MCE properly (by setting CR4.MCE and possibly writing some MSRs) you cannot conclude that it is aware of the fact that it is running in a virtualized environment and that guest physical address relations do not map to machine physical address relations (i.e. a set of pages contiguous in guest physical address space is almost guaranteed to be discontiguous in machine physical address space). Hence if it is more than a single byte/cache line/page that is affected, any such assumptions made in the guest will be wrong.

So just immediately kill the domain as such - e.g similar to "domain_crash_synchronous()". Do not let the guest have any chance to "do something wrong" in the process - it is already broken, and letting it run any further will almost certainly not help matters.

So do not let the affected guest run one more instruction if we can. Sysadmins will learn to consult dom0 diagnostics to see if they explain any sudden guest deaths - no need, as you say, to splurge any raw error data to them.

## General Protection Fault (GPF)

Injecting a GP fault into the HVM guest as an alternative for MCA/MCE multiplexing has been proposed, but:

It is not sure if GP fault is the right thing for non-"MCA PV driver" domains. It cannot guarantee that a GP fault is actually going to "kill" the guest.

GP faults and uncorrectable errors really overlap that much. In a GP fault the extent of the damage is known - you tried to read from an address not in your address space, you lacked permissions for an operation etc. In an uncorrected error situation it is difficult to understand the bounds of the problem in that way - unless the hardware assists with data poisoning etc such errors may well be unconstrained and affect a wider area than just the bracket of code that caught a GP fault.

You can often ring-fence critical code sequences by inserting error barrier instructions before and after it. Those operations are usually very expensive (drain the pipeline or similar) and are suitable only in special places.

When running natively it is usually the "owner" of affected data that sees it bad in memory, e.g. from a read it made. In those cases we have the owner on cpu and can kill/signal it synchronously. There are times when the kernel may be shifting some data on behalf of the application owner (e.g., copyin/copyout, shift network data, page remapping, etc) in which case we still have a handle on the real owner (e.g., PV driver, ring buffer actions). If the access is from a scrub then we should not panic - just wait and see if the owner does indeed use the bad data at which time we take appropriate action.

With the virtualisation layer there is the additional case of the HV or dom0 performing operations on behalf of a guest, i.e. the HV may make the access that traps but its own state is not affected.

CPU errors get still trickier. For example what do we do when we're told that while running guest A we displaced modified data from l2cache that had uncorrectable ECC? We have a physical address only, and no idea of who the data belongs to (guest A, a recently scheduled guest, or the HV?). Where cachelines are tagged with some form of context or guest ID you have a chance, provided that is reported in the error state.

- Windows kernel drivers are allowed to use the kernel exception handling, and ARE allowed to "allow" GP faults if they wish to do so.
- When Linux and *BSD see a GPF while they are in userspace, then they kill the process with a SIGSEGV. If they are in kernelspace, they panic.
- Solaris has some wrappers that can be applied, maybe at some expense to performance, to make protected accesses that will catch and survive various types of error including hardware errors, wild pointers etc.

Further note, if the guest is in user-mode when the GPF happens, then almost all OSs will just kill the application - and there is absolutely no reason to believe that the application running is necessarily where the actual memory problem is - it may be caused by memory scrubbing for example.

Whatever we do to the guest, it should be a "certain death", unless the kernel has told us "I can handle MCEs". This may not be the prettiest solution, but then on the other hand, a "Windows blue-screen" or Linux "oops" saying GP fault happened at some random place in the guest is not exactly helping the SysAdmin understand the problem either.

# Error handling

# Separation of correctable and uncorrectable errors

This is about how the hypervisor detects correctable and uncorrectable errors. #MC correctable errors are not wanted as they block the hypervisor unnecessarily from normal operation. Therefore Xen polls for correctable errors. If the polling interval is too small, a solid error would again keep the hypervisor busy producing (mostly/all duplicate) error telemetry and the diagnosis code in dom0 would burn cpu cycles, too. Xen polls for correctable errors in a 30s internval.

The way errors are observed by the hypervisor, from #MC or from a poll, is propagated to the domains is unimportant from this point of view - e.g., if we decide to take error telemetry discovered via a poll in the hypervisor and propagate it to the domain pretending it is indistinguishable from a machine check that will not hurt or limit the domain processing.

The separation of CE and UE queues stops CEs from flooding the more important UE events (you can always drop CEs if there is no more space, but you can never drop UEs).

Originally it was planned to have some common memory shared between hypervisor and domain into which the hypervisor produces error telemetry and the domain consumes that telemetry.

But it turned out, that there are two problems with this:
a) A dangerous and locking mechanism is necessary to deal properly with multiple machine check errors
b) XenSource does not like this way, because the required size of shared memory is too large. He wants to use hypercalls.

Keir from XenSource suggested to extend the platform or sysctl hypercall. However, these are absolute Dom0 specific. To fullfill machine check requirements, they must at least allow one hypercall from any domain: **Any** domain must be able to fetch error data of uncorrectable errors by design/concept, only Dom0 is able to fetch error data of correctable errors and only Dom0 is able to notify a DomU.

On a multi-socket CPU machine, it is possible that multiple reports of correctable and uncorrectable errors happen at the same time. The Dom0 fetches them and only knows from the notification type the error is correctable. Xen knows from a flag given by the domain, if it wants to fetch a correctable or uncorrectable error and delivers the error data in the order they occurred.

# Threshold register handling

The RevF CPU introduced a new thresholding model specific register (msr). This is basically a hardware counter register for mce errors. Andy Kleen from Novel/SuSE thinks, this register is totally useless because counting can be done in software. So Linux is very likely going to NOT make use of this register.

But the world is not Linux. Customers (also) use Windows, Solaris and even OS/2. Some of them also use *BSD-Unix. For short, we must assume it will be used, because it is there.

A paravirtualized DomU is not allowed to use rdmsr and wrmsr. So it will just receive an exception pass through from Xen. A Dom0 is allowed to use rdmsr and wrmsr. Xen will use it in position for Dom0.

For HVM guests we must report MCA being available or some windows guests will not boot. However, this does not imply that it has to receive MCEs. It is better to always report zero errors on read access to the thresholding msr and ignore write access to it. Actually on read access, we report the guest a value with zero

errors and the lock bit set. This means, the threshold register got locked by the BIOS and is not available for the OS in use.

The patch went into xen-unstable changeset 15143.

---

cegger