

# Optimizing Interrupt Handling Performance for Memory Failures in Large Scale Data Centers

Harish Dattatraya Dixit  
Facebook, Inc.  
hdd@fb.com

Matt Beadon  
Facebook, Inc.  
mbeadon@fb.com

Fan (Fred) Lin  
Facebook, Inc.  
fanlin@fb.com

Zhengyu Yang  
Facebook, Inc.  
zhengyuyang@fb.com

Bill Holland  
Facebook, Inc.  
wholland@fb.com

Sriram Sankar  
Facebook Inc.  
sriramsankar@fb.com

## ABSTRACT

Intermittent hardware failures are generally non-catastrophic and typical large-scale service infrastructures are designed to tolerate them while still serving user traffic. However, intermittent errors cause performance aberrations if they are not handled appropriately. System error reporting mechanisms send hardware interrupts to the Central Processing Unit (CPU) for handling the hardware errors. This disrupts the CPU's normal operation, which impacts the performance of the server.

In this paper, we describe common intermittent hardware errors observed on server systems in a large-scale data center environment. We discuss two methodologies of handling interrupts in server systems - System Management Interrupt (SMI) and Corrected Machine Check Interrupt (CMCI). We characterize the performance of these methods in live environments as compared to prior studies that used error injection to simulate error behavior. Our experience shows that error injection methods are not reflective of production behavior. We also present a hybrid approach for handling error interrupts that achieves better performance, while preserving monitoring granularity, in large scale data center environments.

## CCS CONCEPTS

• **Hardware** → **Transient errors and upsets**; • **General and reference** → **Performance**; **Reliability**; *Experimentation*; • **Computer systems organization** → **Reliability**.

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ICPE '20, April 20–24, 2020, Edmonton, AB, Canada

© 2020 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-6991-6/20/04.

## KEYWORDS

memory errors; interrupts; system performance; hardware reliability; transient errors

## ACM Reference Format:

Harish Dattatraya Dixit, Fan (Fred) Lin, Bill Holland, Matt Beadon, Zhengyu Yang, and Sriram Sankar. 2020. Optimizing Interrupt Handling Performance for Memory Failures in Large Scale Data Centers. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20)*, April 20–24, 2020, Edmonton, AB, Canada. ACM, New York, NY, USA, 11 pages.

## 1 INTRODUCTION

When hardware errors happen in large data center environments, they are either manually repaired or automatically remediated. This is important to keep a large fleet of servers running at high availability for serving the software applications. There are various reasons why hardware errors happen during the life of a component, including material degradation (e.g. the mechanical components of a spinning hard disk drive); over-usage beyond the device's endurance (e.g. NAND flash devices); environmental impacts (e.g. corrosion due to humidity); and manufacturing defects. Large scale environments have minimal human intervention to detect and remediate such hardware failures. They achieve this by deploying autonomous management systems that can handle different types of hardware failures [21, 26, 29].

These autonomous systems are good at detecting permanent hardware failures in a deterministic manner. As a result, the management system takes the server offline, the faulty component is repaired, and then the server is brought back into service. However, when intermittent errors happen, it would be cost-prohibitive and disruptive to repair every component. The system is designed to tolerate or automatically remediate such errors. In addition, a common class of these errors are also correctable in hardware, and hence the application is not expected to see any impact on the correctness of the operations. There have been substantial studies on

program resilience against correctable and uncorrectable intermittent errors [15, 25, 28]. However, in real production environments, we see intermittent errors cause performance aberrations which are unpredictable. The major cause of this behavior is the underlying interrupt handling mechanism implemented in server architectures. Therefore, it is important to consider this while designing both server hardware and the software autonomous management systems.

Hardware errors in server systems are typically reported through interrupts. For memory error handling there are two types that are important. *System Management Interrupt (SMI)* is a high-priority interrupt that puts the system into the System Management Mode (SMM). SMI is commonly used for reporting correctable memory errors, along with the location of the errors [2]. When the system enters SMM to handle the interrupt, operations on all CPU cores are suspended, which leads to wasted cycles. *Corrected Machine Check Interrupt (CMCI)* is another type of interrupt which reports a user-specified number of correctable errors, and the signal is treated as a normal software interrupt, halting the operations on only the local core [4]. While this enables better performance, CMCI also has few constraints, for instance it does not reveal the location of the errors. In this paper, we present both approaches, and characterize their behaviors in production. We also present how we minimized the performance impact of the error reporting mechanism using SMI and CMCI jointly. In our hybrid approach, we minimized the polling frequency of the errors without losing the accuracy of the error count, while retaining the ability to look up error locations.

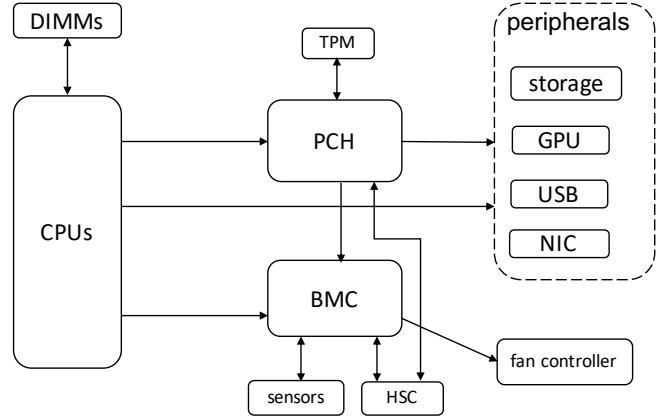
Prior studies [16, 19] in this area have used error injection methods to create varying number of interrupts to observe system behavior. These approaches provide a way to understand interrupt behavior in the absence of any production traffic and without monitoring at scale. However, the simulated behavior is not reflective of real environments. In production, interrupts exhibit non-deterministic behavior. Albeit, the performance impact is perceivable through cascading tail latency at the service level. These impacts are seen even at low interrupt counts due to the penalty associated with interrupt handling and logging the required error details. This paper presents an industry experience that looks at optimizing the interrupt performance in real production environments.

The rest of the paper is structured as the following: Section 2 describes the common intermittent hardware errors and the reporting mechanisms used in server systems. Section 3 covers the autonomous management system for error remediation in our production environment. Section 4 illustrates SMI and its performance impact when reporting correctable errors, and Section 5 demonstrates how we use

CMCI jointly with SMI to minimize the performance impact while enabling necessary monitoring. Finally, Section 6 concludes the paper.

## 2 IMPACT OF INTERMITTENT ERRORS

A typical server architecture consists of CPUs and Dual In-line Memory Modules (DIMMs) for executing operations, hard disk drives (HDD) or solid state drives (SSD) for data retention, and a network interface card (NIC) for external communications, as show in Figure 1. Special function devices for performing parallel processing like GPUs are connected to the CPU using the Peripheral Component Interconnect (PCI) Bus. To monitor the health of the machine, we use a Baseboard Management Controller (BMC), with multiple sensors for voltages, fan speed, Hot Swap Controller (HSC) etc. Each of these blocks are prone to hard faults, where they don't appear online, or intermittent faults which can cause a temporary performance drop or unavailability.



**Figure 1: High-level architecture of a server.**

In order to understand why intermittent errors cause performance impact, it is important to understand the common types of intermittent errors in server systems and the error reporting mechanisms used. Unlike permanent failures which can be reproduced in any operating state and workload, intermittent failures occur only under certain criteria. Intermittent errors also require continuous reporting for the hardware remediation system to decide when the hardware has degraded to a point that it needs be repaired. This section discusses intermittent errors, their impacts on server performance, and the reporting mechanisms for the errors.

### 2.1 Common Intermittent Errors

In this section, we describe primarily memory errors, and briefly cover other intermittent errors that happen in CPU,

PCI, NIC to provide examples of areas where our methodology could be useful.

**2.1.1 Memory Error.** A memory device can encounter bit faults due to many factors including hardware degradation, manufacturing defects, electrical noise, and cosmic rays. Conventionally, memory errors are categorized by the cause and by correction mechanisms [8].

- **Hard vs. Soft Errors**

Hard memory errors are inherent defects in the chip or the memory array. Retries or rewrites will not eliminate the error, as the hardware circuitry is permanently affected, and the hard error would continue to repeat. Soft memory errors are errors due to an electrical noise or a glitch in the system. In the case of hard errors, the remediation is to replace the component. Soft errors do not necessarily repeat and are usually fixed by retries.

- **Correctable vs. Uncorrectable Errors**

Implemented with the use of Hamming codes, Error Correcting Code (ECC) [18, 20] is a mechanism for correcting memory errors. When an ECC protected memory encounters correctable errors, the memory controller can detect and correct the errors up to design points. While correctable errors have no impact on the correctness of the program processing, they are reported through system interrupts, which could lead to noticeable latencies in a short period. Uncorrectable errors cannot be corrected by the ECC. These errors usually cause kernel panics and crash the machine. The errors induce noticeable unavailability for a production service through frequent reboots. For the purposes of this paper, we will focus our discussion around the reporting of correctable errors. We will discuss the reporting mechanisms and the performance impact in Section 2.2 and Section 4

**2.1.2 Other Intermittent Errors.**

- **CPU Intermittent Errors**

A Machine Check Exception (MCE) is raised by the CPU when it encounters an uncorrectable hardware issue either within itself or a subsystem connected to it [7]. An MCE is a subset of CATAstrophic ERRors (CATERR), and its root cause can usually be determined by examining the register values from the CPU crash dump. An MCE usually results in system hangs or reboots, which are highly disruptive to a service. Another intermittent error, thermal throttling is typically asserted when the CPU on a host is operating outside the thermal or voltage spec. Once this error is asserted, the CPU typically gets throttled to a lower operating frequency, which induces latency spikes on memory.

- **PCIe Error**

An error can occur on the PCIe link connecting the CPU, or through the Platform Controller Hub (PCH) to an attached PCIe device due to electrical noise, a loose connection, or a defective (or poorly tuned) PCIe receiver or transmitter [23]. PCIe link correctable data errors generate messages to the OS log and to the System Event Log (SEL). With each detected PCIe bus error, the affected transaction will be retried one or more times. In most cases, the retries are successful, and a correctable error is logged. In rare cases, where the retries are not successful, the event will become an uncorrectable error, and the link may go down, affecting performance and availability.

- **Data Link CRC Error**

Cyclic Redundancy Check (CRC) [27] errors indicate a corruption in the received data. CRC errors can occur due to a faulty link between the sender and receiver, or due to a defect in either the sender or receiver. Widely used in both storage and network links for data transfers, a CRC error usually indicates that the link is bad. To recover from this class of errors, data packets are re-transmitted when the CRC corruption is detected. The rate of errors is directly proportional to the number of retransmits that we may see in the system. This reduces the overall transfer bandwidth between the sender and the receiver as the percentage of retransmits can grow to occupy a large part of the overall data transfer rate, thereby impacting performance.

While we presented these different intermittent errors, for purposes of this paper, we will provide a deeper discussion of memory correctable errors and the related interrupt handling mechanisms.

## **2.2 Error Reporting for Memory Errors**

There are two major ways of reporting memory errors in server systems - EDAC and *mcelog*.

- **EDAC**

EDAC stands for "Error Detection and Correction" [3, 9]. The EDAC driver consists of Linux kernel modules, which make use of the error detection facilities of the memory controller hardware. EDAC has a number of features for detection and correction. For monitoring memory errors, EDAC internally uses the CMCI reporting mechanism. EDAC enables monitoring of memory errors at the granularity of DIMMs and caches in the CPU. EDAC is a feature that is supported on AMD (ghes-EDAC) and Intel (native EDAC) CPUs. However, in order to perform accurate accounting of memory errors correctly through EDAC, error correction and detection support needs to be enabled in system

firmware, i.e. BIOS (Basic Input/Output System). The EDAC driver also provides a configuration mechanism for enabling different types of detection and correction mechanisms. The logging features are enabled for both memory correctable and memory uncorrectable errors.

- *mcelog*

*mcelog* is another mechanism through the OS that is used for monitoring memory and CPU errors [24]. The *mcelog* runs as a daemon on a linux machine and aggregates memory errors through polling or interrupts. *mcelog* also provides support for page offlining (for eligible pages). *mcelog* enables reporting of memory errors at a DIMM level. If the DIMM is not found, it falls back to the memory channel or the CPU socket level granularity of error reporting. The reporting mechanism relies on the memory controller and machine check registers in the CPU. The daemon accounts for correctable errors in these blocks through the kernel.

While EDAC and *mcelog* both report memory errors, we use EDAC in our infrastructure due to its simpler error reporting features. *mcelog* in contrast provides a large number of config capabilities; however, in a scalable infrastructure, consuming all those capabilities effectively is difficult as the configurations (e.g. page offlining) introduce variability within similar machines. Since EDAC relies on the memory controller metrics, it is easier to scale and be backward compatible.

### 3 EXPERIMENT INFRASTRUCTURE

#### 3.1 Autonomous Remediation Flow

Autonomous systems are deployed for detecting and remediating hardware errors to keep the service infrastructure at high availability [21, 26, 29]. Figure 2 shows a hardware remediation system discussed in [26]. A tool named *MachineChecker* periodically runs a set of checks to detect hardware failures. When a hardware error is detected, an alert is raised by the daemon monitoring *MachineChecker* and sent to a centralized alert management system called *Alert Manager*. *Facebook Auto-Remediation (FBAR)* then responds to the alert with customizable remediations. The system is designed with the flexibility for different service owners to customize the corresponding actions given the failure signals, so servers dedicated for different services can go through the desired remediations at different rate limits, to satisfy the specific service requirements. When the auto-remediation in FBAR cannot bring the server back online, the hardware failure would then be passed to *Cyborg*, a tool that is designed for low-level software fixes, e.g. reimaging and firmware upgrade. *Cyborg* can also create a repair ticket for field engineers if a physical repair or manual debugging is needed.

The data gathered from these multiple systems enables us to identify performance issues or anomalies in hardware errors across the fleet of servers in our data centers.

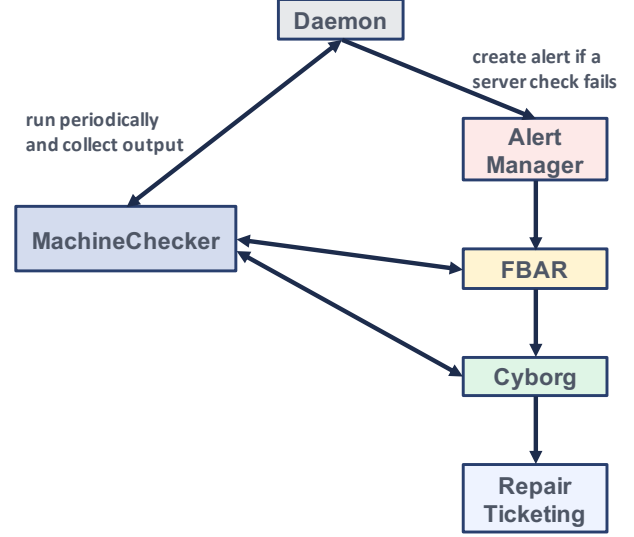


Figure 2: The hardware failure detection and remediation flow.

#### 3.2 System Setup

With the autonomous hardware remediation system, we have been able to collect hardware failure data in a fleet of servers across multiple large-scale datacenters. The system runs across multiple hardware generations and configurations, collecting hardware failure data from the System Event Log (SEL) and kernel log messages.

While we have significant monitoring for the actual production events, we also validate that these errors are in fact occurring in these systems. For reproducing the conditions that trigger the hardware failures, we deployed multiple benchmarks in the remediation flow, including CoreMark [1], stream-scaling [6], MPrime [11], stressapptest [10], SPEC benchmarks like perlbench, bzip2 [12], and iperf3 [13]. In Section 4 and 5 we present the performance impact and how we minimize it for memory correctable errors detected by stressapptest.

We also deployed a fine-grained *stall detector*, a tool that detects and measures the total time a CPU spent in interrupt handling. The stall detector was designed to measure stalls for every core while servicing the interrupts. It logs the stalls on the machine, to enable measurement and comparison of the performance impact induced by SMI and CMCI.

## 4 PERFORMANCE IMPACT OF SMI INTERRUPTS

### 4.1 SMI

System Management Interrupt (SMI) is a high-priority interrupt which puts the system into System Management Mode (SMM). As shown in Figure 3, SMI is handled by system firmware. When an SMI is invoked due to any of the sources (e.g. hardware errors, thermal or power events), the control within the firmware-first model is then transferred to the firmware. Within the firmware, the tasks are split between the logging handler and the interrupt handler. When both the routines are finished, the control is transferred back to the OS to resume the operations that were halted.

SMIs are the highest priority interrupts that are available on the server. These interrupts are non-maskable, and are not visible to the user or the kernel applications, and hence cannot be deferred. Any other interrupt would be kept pending until SMI exits SMM. The configurations for SMI can be altered within the firmware. In SMM, the machine has the highest privileged access, and thus provides a detailed overview and debug information for errors on the machine.

The performance impact of a System Management Interrupt is high because all the cores are suspended in SMM. The interrupt provides control to an error logging handler, and returns to normal operating mode after writing to an error log. Pending requests experience latencies while the cores are suspended in SMM. In addition, since the amount of time spent in SMM is non-deterministic as it depends on logging and interrupt handler routines, this introduces an unpredictable system behavior for time-bound applications.

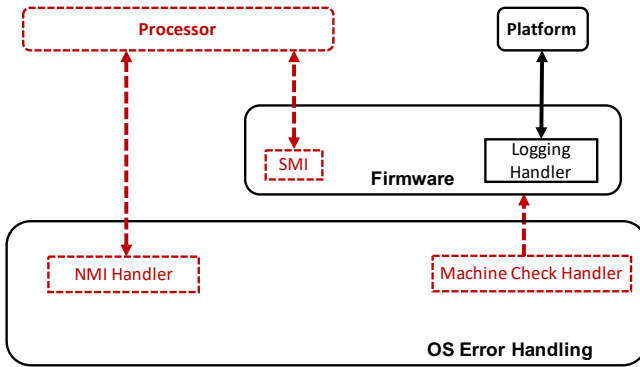


Figure 3: Interrupt handling architecture with SMI (dashed lines show interrupt path).

### 4.2 Performance Impact in Synthetic Data vs. Production

In a synthetic environment, it is possible to generate errors at a particular rate and measure the CPU time spent in system stall. A number of error injection scenarios that change SMI rate from 1 per second to 100 per second and corresponding test mechanisms can be used to show the different effects of SMI. Prior studies are valuable to indicate and quantify the experimental measurements for SMI with different rates.

However, production environments are not the same as synthetic environments in the error occurrence rate. In contrast, in a production environment, the errors are sporadic in nature. In a synthetic environment, it is possible to control the environmental factors, as well as the experimental setup, to introduce errors at the desired rates. In contrast, in a production environment usually consists of hundreds of thousands of machines, having randomness with respect to the workload running on the machine, the age of the machine. The wear and tear varies with respect to components based on environmental and thermal factors. As a result, production environments exhibit a larger randomness for error generation in comparison to synthetic and controlled environments.

In addition, the time spent in the System Management Mode (SMM) is dependent on the details for the logging mechanism. As a result, the number of system stalls as well as the duration of the system stalls due to SMIs in a production environment, are not the same as that measured for benchmarks with error injection. This has drastic impact on the performance of a workload executing on a machine, as the workload is “unprepared” for a stall, and an SMI in the middle of a critical workload can create cascaded failures in a production fleet. An example of this is provided in the next section.

**Observation 1:** System Management Interrupts (SMI) cause the machines to stall for hundreds of milli-seconds based on the logging handler implementation. This is measurable performance impact to report corrected errors.

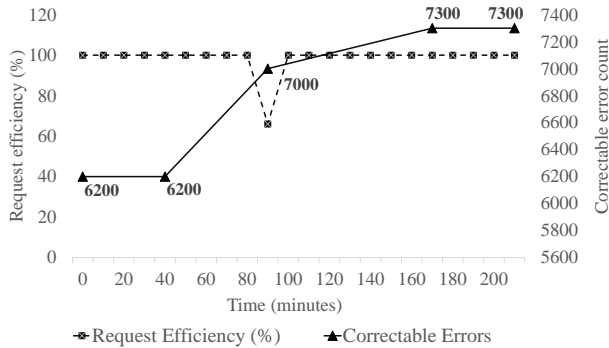
A typical internet service has a front-end web infrastructure, with intermediate caching service, and back-end databases. In an example caching service within Facebook infrastructure, each CPU core responds to several hundred thousand requests per second. In the few machines which had spikes in correctable errors, the CPU went into SMM mode (which stalls all cores of the CPU) and the caching service dropped thousands of requests per second. This caused timeouts for one-third of the requests for the service on that machine. These timeouts were observed on machines encountering System Management Interrupts (SMI), due to logging of correctable errors into the System Event Log (SEL).

Since large scale services have higher level aggregators and balancers, these issues are managed seamlessly. However, in a fixed capacity service architecture, this behavior will pressurize the rest of the machines which will get overloaded with requests. This might result in service capability being dynamically affected. Hence, due to unpredictable nature of SMIs, the system can experience cascaded service impact within front-end application performance.

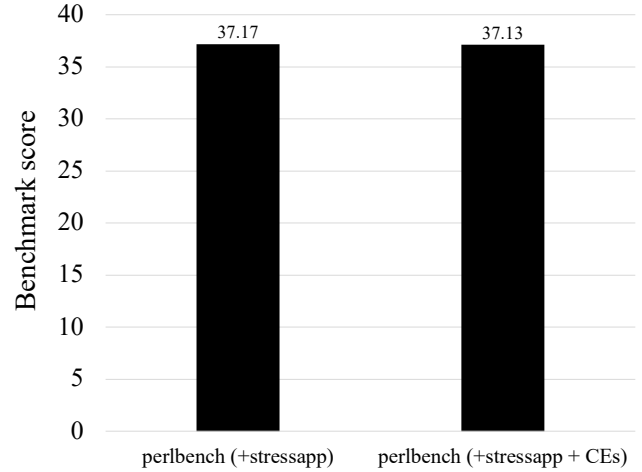
Figure 4 demonstrates a correlation of the correctable errors with service level caching request efficiency. The percentage of requests successfully executed by the service, within the deadline, is termed as request efficiency. When the caching service hosts encounter correctable errors, at every N correctable errors, an SMI is triggered, and all the cores encounter a system stall. This results in deadline-driven caching requests timing out due to cores not being available for performing workload computation while servicing the SMI. In this example, the number of errors generated resulted in one SMI event, which in turn caused the request efficiency to drop.

Similar to prior work that uses benchmarks while injecting errors [19], we deployed benchmarks to detect the performance anomaly for sporadic memory errors. The benchmarks didn't detect deviations in their scores for two key reasons, the benchmarks are long running, so sporadic patterns will not impact scores. In addition, the errors are not generated at a fixed frequency through an external test setup or error injection, and thus the SMIs are not intentionally disrupting the service.

Figure 5 shows the perlbench benchmark scores for 2 different cases. To obtain the first score, the benchmark is run with stressapptest running in the background on a machine which has no correctable errors (has a good DIMM). To obtain the second score, the benchmark is run with stressapptest in the background with thousands of correctable errors on the host (has a faulty DIMM). Both scores are close to each other



**Figure 4: Caching request efficiency vs. correctable error count.**



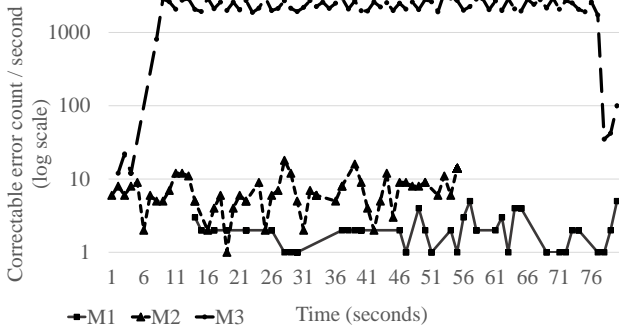
**Figure 5: Benchmark score variation with respect to correctable errors.**

(within error bounds) to indicate that the benchmark did not observe any difference with or without correctable errors on the machine. As observed, the benchmarks cannot capture the system slowness observed when actual production behavior is replicated with sporadic memory errors.

Figure 6 shows the number of correctable errors across three machines over time with varied error rates. These errors are *not* injected on the machine. The graph is generated by randomly selecting three machines with faulty memory known to have correctable errors, and plotting the errors generated under the same workload over time. This also shows that memories can inherently have faults, which generate different rate of errors. This is quite common in a large production environment. From the figure, we can observe that the errors do not have a consistent occurrence rate. In general, memory errors are dependent on the memory address access pattern and the variable memory utilization of the application over time.

With the stall detector, we observed that the correctable errors on the machine directly induced system stalls, and this in turn caused a spike in the request timeouts in a caching service. Since the stall detector is always monitoring for stalls, any single SMI event is captured and logged, to correlate with application performance impact.

Optimizing error handlers within SMI for reducing the stall time for all the cores is another mechanism to reduce performance impact. We provide recommendations on optimizing error handlers in Section 4.3



**Figure 6: Variable occurrence of correctable errors on three servers.**

### 4.3 Optimizing error handlers within SMI

When an SMI is triggered, the control is solely limited to the error handler within SMI. It is possible to optimize the error handler to reduce the time spent in SMM by exploring:

- Improving the retry strategies of Intelligent Platform Management Interface (IPMI) commands [5] used in SMM memory error handler to log memory error data into System Event Log (SEL) within the side band agent (ex. BMC).
- Adjust the IPMI commands' processing flow within the side band agent to expedite the acknowledgement of these IPMI commands.
- Minimize the amount of data logged to the SEL for memory errors by trimming the logged data to only what is required to identify the Field Replaceable Unit (FRU).

Optimizing these proprietary mechanisms [14] can reduce performance impact, however, stalls will occur on all cores due to SMM mode. Hence we explore error reporting through CMCI interrupt handling.

To summarize the findings, the performance impact due to SMIs in a production environment is fine-grained, non-deterministic and causes cascaded impact in a production service. In order to minimize the performance impact due to System Management Interrupts, we modify the default for logging correctable errors from SMI to CMCI.

**Observation 2:** Benchmarks like *perlbench* within SPEC are useful to quantify system performance. For variable events, we need to augment the benchmarks with fine-grained detectors to capture performance deviations.

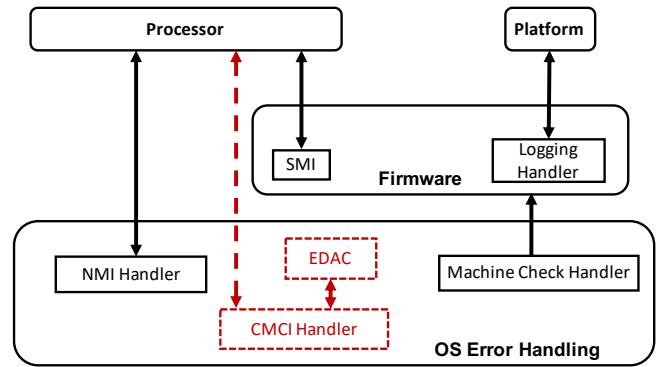
## 5 MINIMIZING PERFORMANCE IMPACT USING CMCI INTERRUPTS

### 5.1 CMCI

As shown in Figure 7, there is an alternate way of reporting memory errors. EDAC through the use of Corrected Machine Check Interrupts (CMCI) provides an accounting and aggregation of memory errors. This reporting is outside of the SMI, firmware, and Non-Maskable Interrupt (NMI) handlers. As a result, it is a simpler, lower cost per core reporting mechanism. Earlier mechanisms relied on a periodic polling mechanism with a fixed threshold. After the fixed threshold was crossed, on a subsequent poll, the errors were reported. However, CMCI provides a low cost per core interrupt, which is triggered based on the threshold configs controlled through Model Specific Registers. So instead of continuously polling for the errors at a software level, the CMCI interrupt is now triggered only when the threshold is crossed. This ensures that the errors are reported accurately per bank, and also per core. EDAC utilizes the CMCI mechanism to report the errors by reading through registers in each CPU core. EDAC aggregates all the errors logged by all of the CPU cores by cycling through the CMCI counters for each core, in order to report the total number of errors logged for the system, during the EDAC specified polling interval. The aggregation of the errors to provide a per system count is performed by EDAC running on just one core.

In Figure 8, we see the growth of stalls using different error reporting mechanisms in a production environment on an example host as the number of correctable errors increases. Reporting memory correctable errors through SMIs accumulates stalls at a faster rate than reporting the same errors through CMCI.

**Observation 3:** SMI interrupts are several times more computationally expensive than CMCI interrupts for correctable memory error reporting in a production environment.



**Figure 7: Interrupt handling architecture with CMCI (dashed lines show interrupt path).**

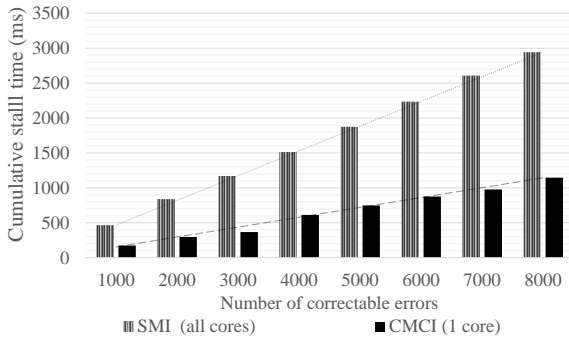


Figure 8: Comparison between SMI and CMCI stalls.

The EDAC driver offers a few configuration options for error reporting and error management:

- enabling PCI error reporting
- enabling uncorrectable error reporting
- enabling correctable error reporting
- polling frequency for errors
- enabling kernel panic on Uncorrectable Errors (UCE).

In our production infrastructure, we enable CMCI by default, and leverage the CMCI and EDAC infrastructure for reporting correctable errors. To demonstrate the performance impact observed through the EDAC based reporting mechanisms, we select a group of machines with measurable correctable error rate for obtaining the results below. We narrow our configuration changes on the performance characteristics to two main knobs in EDAC configs; enabling reporting of correctable errors through EDAC and polling frequency. We tune the polling frequency and provide observations on the stalls and the impact of these stalls on application workloads.

## 5.2 Impact of EDAC polling interval on individual stall time

Figure 9 shows the maximum time a core spends in EDAC aggregation logging with different configurations of polling interval. The graph captures the maximum stall time observed (in ms) per core. With increased polling interval for EDAC, it is more likely that multiple cores will have logged correctable errors. As a result, the logging mechanism of the EDAC driver needs to scan through more cores, aggregating and logging more messages.

**Observation 4:** We see that with increased polling interval, the amount of time spent in individual aggregate logging by the EDAC driver increases.

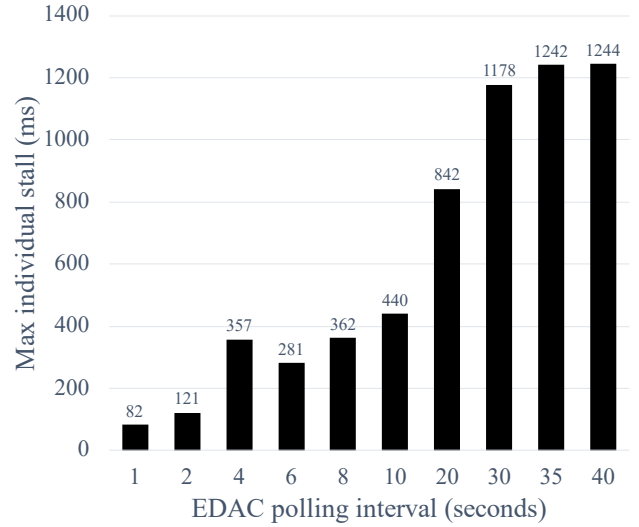


Figure 9: Max individual stall vs. polling frequency.

## 5.3 Impact of EDAC polling interval on total stall time

In Figure 10, we capture the total stall time a core spends (on average) across multiple EDAC aggregation logging with different configurations of polling interval. The graph captures the total stall time observed (in ms) per core on average. With increased polling interval for EDAC, more cores will have detected correctable errors. Instead of reporting a small bucket of correctable errors, the longer polling interval allows it to log the errors from multiple cores at once. This prevents frequent transitions between executing “workloads”

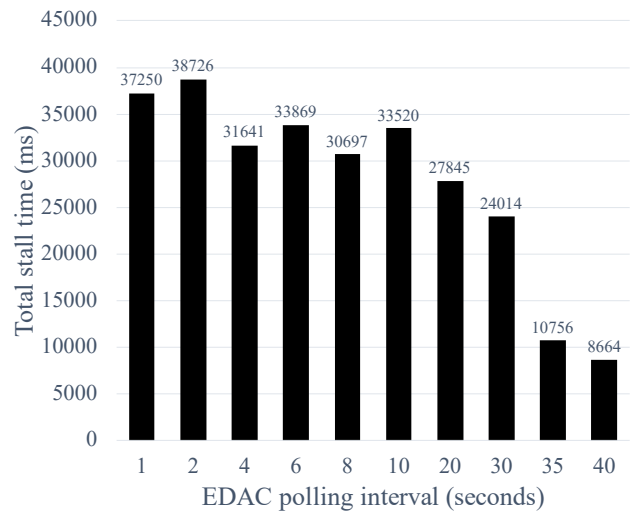


Figure 10: Total stall time vs. polling frequency.

to executing “logging”. The frequent context switches on one core have a larger penalty for smaller polling intervals.

**Observation 5:** We see that with an increased polling interval for EDAC, frequent context switches are reduced. Hence the total time a machine spends in stalls will be reduced.

#### 5.4 Impact of EDAC polling interval on error visibility

Figure 11 demonstrates the number of errors that are lost per poll with respect to different configurations of EDAC polling interval. The error counting registers have a finite bit-width allocated for tracking error count within a memory controller. Within the CMCI architecture, the counters trigger the CMCI interrupt after the threshold is crossed. If the EDAC driver doesn’t log the error counters frequently enough, the error counters can overflow. We then read an incorrect value for the correctable errors which are reported by EDAC to be the true value. This creates a condition where errors will be lost in the overall EDAC accounting scheme.

In a large-scale infrastructure, we set criteria for acceptable error rate for correctable errors beyond which we flag a DIMM as faulty. Error rate in 10s of correctable errors per second (10 CE/s) is more likely an indication that an entire row is faulty in memory, instead of just a single cell. If the errors are under-counted with a longer polling interval, this can lead to faulty DIMMs left in the fleet. In order to prevent that, we enforce a hard constraint based on our thresholds, and avoid under-counting the errors which will put our detection mechanisms at risk. For a different threshold, this will result in a different value of polling interval.

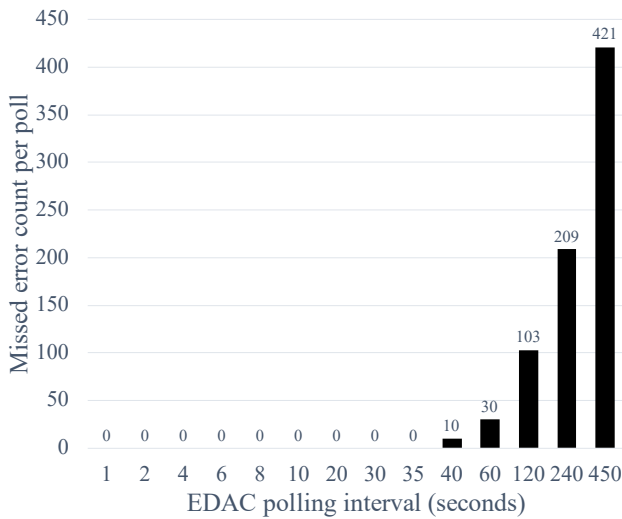


Figure 11: Missed error counts vs. polling frequency.

**Observation 6:** With increased polling interval for EDAC, we run the risk of overflow in error aggregation.

#### 5.5 Optimization of polling interval

Our goal is to reduce overall stall time on a machine using EDAC logging, and we have the following options.

- Disable EDAC logging: We can completely disable the logging of correctable errors by EDAC through one of the configs mentioned above. This eliminates the stall time due to EDAC logging on a machine. However, this in turn also means that we lose the ability to report errors and identify which DIMMs are faulty or have crossed a threshold of 10 Correctable Errors (CE) per second. So we don’t pursue this option as visibility into memory health is an imperative in a large infrastructure.
- Fine-tune polling interval: From observations 4, 5, and 6 above, we want to obtain an optimization point where we have the minimum total stall on a system, and yet retain the ability to diagnose our faulty memories correctly at our threshold.

Combining our 3 observations, we notice that having a polling interval between 33-38 seconds, provides an optimum point for minimizing total system stalls, while still retaining error visibility, as seen in Figure 12.

The downside to this method is that *one* of the cores will, at the worst case experience a stall in the range of 1200 ms. This is only true if all the cores have logged an error. Given that the arrival rate of memory errors is not constant, the probability of this reaching a 1200 ms value is small. Even with the worst-case scenario, we are still better than a large stall on all the cores through (SMI) or many consecutive stalls which amount to a larger aggregate stall time.

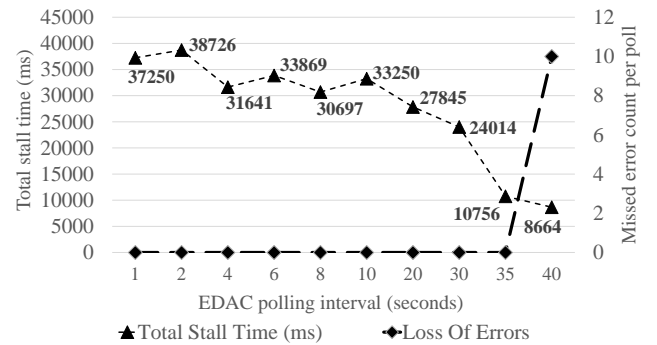


Figure 12: Total stall time vs. missed error count.

## 5.6 Optimizing Interrupt Handling Using Hybrid CMCI and SMI approach

Memory error reporting through SMI enables us to collect debug information related to row, cell and physical address of the failure. This information is useful for performing actions like Post-Package Repair (PPR) [17, 22]. Switching from SMI to CMCI means that PPR is no longer usable. CMCI through EDAC does not provide the same level of detail as that of SMI for memory correctable errors.

In order to minimize stalls and still use the PPR feature, we use a hybrid approach where we enable the SMI reporting in debug mode for a machine. When a machine is reported to have a high error rate for memory errors by EDAC, it is taken offline and put in a special debug mode to determine what action is necessary. In this debug mode, the interrupt is switched from CMCI to SMI and the threshold is reduced to trigger SMIs at the first error, instead of 1 SMI every Nth error. The machine is then subjected to a memory stress workload to obtain the physical address and the necessary detailed information needed for PPR. If the error is successfully recreated, a PPR action is initiated so that the memory address can be remapped.

This complex remediation provides us with the limited performance impact of CMCI reporting through EDAC when the machines are in production, but when the machine is reported for high memory errors, the additional information from SMI is used for performing PPR.

## 6 CONCLUSIONS

This paper presents important observations that highlight the performance impact of interrupt handling using large scale production data. Memory errors are a common class of intermittent errors and this phenomenon is less understood, as evident by the number of server manufacturers that use SMI as default mechanism [2]. In addition, we also present a methodology that explores the tradeoff between performance impact, granularity of error information and diagnostic capability. We believe that this experience will benefit several system designers to explore interrupt handling mechanisms that are better tuned to internet scale services.

### Acknowledgement

The authors would like to thank Domas Mituzas, Gautham Vunnam, Chris Davis, Junjie Wu, and other infrastructure engineers for their inputs in the implementation of the tooling and valuable technical suggestions.

## REFERENCES

- [1] 2002. <https://www.eembc.org/coremark>
- [2] 2003. Intel Platform Innovation Framework for EFI System Management Mode Core Interface Specifications (SMM CIS). <https://www.intel.com/content/dam/www/public/us/en/documents/reference-guides/efi-smm-cis-v09.pdf>
- [3] 2008. <http://bluesmoke.sourceforge.net/>
- [4] 2011. Platform-Level Error Handling Strategies for Intel Systems. <https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/platform-level-error-strategies-paper.pdf>
- [5] 2013. Intelligent Platform Management Interface Specification Second Generation. <https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ipmi-second-gen-interface-spec-v2-rev1-1.pdf>
- [6] 2016. <https://github.com/gregs1104/stream-scaling>
- [7] 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf>
- [8] 2016. Managing Correctable Memory Errors on Cisco UCS Servers. <https://www.cisco.com/c/dam/en/us/products/collateral/servers-unified-computing/ucs-manager/whitepaper-c11-736116.pdf>
- [9] 2017. Error Detection And Correction (EDAC) Devices. <https://www.kernel.org/doc/html/v4.14/driver-api/edac.html>
- [10] 2018. <https://github.com/stressapptest/stressapptest>
- [11] 2019. <https://www.mersenne.org/download>
- [12] 2019. <https://www.spec.org>
- [13] 2019. <https://iperf.fr>
- [14] 2019. <https://2019ocpglobalsummit.sched.com/event/Jil9/performance-oriented-firmware-implementation-for-error-handling-in-large-scale-data-center>
- [15] Cristian Constantinescu. 2008. Intermittent Faults and Effects on Reliability of Integrated Circuits. In *IEEE Annual Reliability and Maintainability Symposium*.
- [16] Brian Delgado and Karen L. Karavanic. 2013. Performance Implications of System Management Mode. In *IEEE International Symposium on Workload Characterization (IISWC)*.
- [17] Samsung Electronics. 2014. DDR4 SDRAM Specification. [https://www.samsung.com/semiconductor/global.semi/file/resource/2017/11/DDR4\\_Device\\_Operations\\_Rev11\\_Oct\\_14-0.pdf](https://www.samsung.com/semiconductor/global.semi/file/resource/2017/11/DDR4_Device_Operations_Rev11_Oct_14-0.pdf)
- [18] Neal Glover and Trent Dudley. 1990. *Practical Error Correction Design For Engineers* (second ed.). Cirrus Logic.
- [19] Mark Gottscho, Mohammed Shoaib, Sriram Govindan, Bikash Sharma, Di Wang, and Puneet Gupta. 2016. Measuring the Impact of Memory Errors on Application Performance. In *IEEE Computer Architecture Letters (CAL)*.
- [20] R. W. Hamming. 1950. Error Detecting and Error Correcting Codes. In *Bell System Technical Journal*, Vol. 29.2.
- [21] M. Isard. 2007. Autopilot: Automatic Data Center Management. In *ACM SIGOPS Operating System Review*.
- [22] JEDEC. 2012. JESD79-4 DDR4 SDRAM.
- [23] Mike Johnson and Ravi Budruk. 2015. *PCI Express Technology* (second ed.). MindShare Press.
- [24] Andi Kleen. 2010. mcelog: memory error handling in user space. *Linux Kongress*
- [25] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. 2010. A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility. In *USENIX Annual Technical Conference*.
- [26] Fan (Fred) Lin, Matt Beadon, Harish Dattatraya Dixit, Gautham Vunnam, Amol Desai, and Sriram Sankar. 2018. Hardware Remediation At Scale. In *IEEE/IFIP International Conference on Dependable Systems and*

*Networks Workshops.*

- [27] W. W. Peterson and D. T. Brown. 1961. Cyclic Codes for Error Detection. *Proceedings of the IRE.*
- [28] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. 2015. Characterizing the Impact of Intermittent Hardware Faults on Programs. In *IEEE Transactions on Reliability.*
- [29] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. 2015. Large-scale Cluster Management At Google with Borg. In *European Conference on Computer Systems (EuroSys).*