We quantify the extent of performance degradation caused by memory errors. Our contributions are as follows: that correctable memory errors can lead to wildly unpredictable and addressing this issue is important: a recent study at Facebook found that suffer from millions of correctable errors [26]. Understanding and consideration because of the rarity of memory errors. In modern performance on machines when errors actually occur. In past smaller-alternatively using less reliable DRAM chips to reduce provisioning costs [24]., we observed in past smaller-scale systems (QoS) from SMM on latency-sensitive user applications.

There is a large body of prior work that address reliability in systems [33]. Many compelling reliability-aware techniques and reduce performance degradation. Delgado et al. [13] were the first to experimentally expose the performance implications of Intel’s System Management Mode (SMM), which is often used for memory error reporting (and which we discuss in this work). They observed inconsistent Linux kernel performance and reduced quality-of-service (QoS) from SMM on latency-sensitive user applications. Researchers have generally considered the server and application performance overheads of DRAM fault tolerance schemes only in the case when no errors occur. Prior work, however, has not identified or measured the performance degradation caused by memory errors in systems with faulty hardware. This effect is important: datacenter operators have observed performance-degrading memory errors to occur routinely in the field, where their increase in quality-of-service has been observed in the past. Unfortunately, memory reliability is expected to decrease in future technologies [29] as a result of increasing manufacturing process variability in nanometer nodes [19], [25], [27], [31]. Meanwhile, researchers are actively exploring new approaches to designing memory for the datacenter, such as intentionally using less reliable DRAM chips to reduce provisioning costs [24], [29].

Thus far, the research community has not explored application performance on machines when errors actually occur. In past smaller-scale systems with older DRAM technology, this was not a major consideration because of the rarity of memory errors. In modern warehouse-scale computers (WSCs), however, the worst case for errors is no longer a rare case. While most servers have low error rates, in any given month, there are hundreds of servers in a datacenter that suffer from millions of correctable errors [25]. Understanding and addressing this issue is important: a recent study at Facebook found that correctable memory errors can lead to wildly unpredictable and degraded server performance [25], which is a primary challenge in cloud computing [9].

In this work, we experimentally characterize the performance of several applications on a real system that suffers from correctable memory errors. Our contributions are as follows:

- **We identify why and how memory errors can degrade application performance** (Sec. 3). When memory errors are corrected, they are reported to the system via hardware interrupts. These can cause high firmware and software performance overheads.
- **We quantify the extent of performance degradation caused by memory errors on a real Intel-based cloud server using a custom hardware fault-injection framework** (Sec. 5). Our measurements show that batch-type SPEC CPU2006 benchmarks suffer average 2.5x degradation in execution time, while an interactive web-search application can experience up to 100x degradation in quality-of-service when just 4 memory errors are corrected per second using “firmware-first” error reporting.

2 Related Work

There is a large body of prior work that address memory errors in systems [23]. Many compelling reliability-aware techniques and reduce performance overheads. Delgado et al. [13] were the first to experimentally expose the performance implications of Intel’s System Management Mode (SMM), which is often used for memory error reporting (and which we discuss in this work). They observed inconsistent Linux kernel performance and reduced quality-of-service (QoS) from SMM on latency-sensitive user applications.

Researchers have generally considered the server and application performance overheads of DRAM fault tolerance schemes only in the case when no errors occur. Prior work, however, has not identified or measured the performance degradation caused by memory errors in systems with faulty hardware. This effect is important: datacenter operators have observed performance-degrading memory errors to occur routinely in the field, where they increase in quality-of-service has been observed in the past. Unfortunately, memory reliability is expected to decrease in future technologies [29] as a result of increasing manufacturing process variability in nanometer nodes [19], [25], [27], [31]. Meanwhile, researchers are actively exploring new approaches to designing memory for the datacenter, such as intentionally using less reliable DRAM chips to reduce provisioning costs [24], [29].

Thus far, the research community has not explored application performance on machines when errors actually occur. In past smaller-scale systems with older DRAM technology, this was not a major consideration because of the rarity of memory errors. In modern warehouse-scale computers (WSCs), however, the worst case for errors is no longer a rare case. While most servers have low error rates, in any given month, there are hundreds of servers in a datacenter that suffer from millions of correctable errors [25]. Understanding and addressing this issue is important: a recent study at Facebook found that correctable memory errors can lead to wildly unpredictable and degraded server performance [25], which is a primary challenge in cloud computing [9].

In this work, we experimentally characterize the performance of several applications on a real system that suffers from correctable memory errors. Our contributions are as follows:

- **We identify why and how memory errors can degrade application performance** (Sec. 3). When memory errors are corrected, they are reported to the system via hardware interrupts. These can cause high firmware and software performance overheads.
- **We quantify the extent of performance degradation caused by memory errors on a real Intel-based cloud server using a custom hardware fault-injection framework** (Sec. 5). Our measurements show that batch-type SPEC CPU2006 benchmarks suffer average 2.5x degradation in execution time, while an interactive web-search application can experience up to 100x degradation in quality-of-service when just 4 memory errors are corrected per second using “firmware-first” error reporting.

3 DRAM Error Management and Reporting

Error reporting, or logging, in firmware and/or software is required for datacenter operators to detect failures and service them appropriately. They also enable the numerous past [33], [24], [19], [33]. Array Error Management and Reporting (AEMR) and future field studies of DRAM errors. Page retirement, which has been recently shown to significantly reduce DUE occurrences at Facebook [26], also relies on accurate and precise error logging in order to identify failing pages. We believe that a primary cause of performance degradation from memory errors is the firmware/software stack, not the hardware fault tolerance mechanisms. Thus, we discuss how DRAM errors are made visible to software. Because Intel-based systems are dominant in the cloud, we focus on their Machine Check Architecture (MCA) [2], specifically for Haswell-EP (Xeon ES-v3) processors. Other platforms may have similar mechanisms, but they are beyond the scope of this work.

When the ECC circuits in the memory controller detect or correct an error, they write information about the error into special registers. This includes information like the type/severity of the error (CE or DUE) and possibly the physical address. The hardware then raises an interrupt to either the OS or firmware, but not both simultaneously; this option is specified statically at the system boot time [2], [3].

If the software interrupt mode is selected, the ECC hardware raises either a Corrected Machine Check Interrupt (CMCI) for a CE, or Machine Check Exception (MCE) for a DUE. For an MCE, the

---

**Authors’ Copy dated August 8, 2016**

To appear in the IEEE Computer Architecture Letters (CAL)

DOI: 10.1109/LCA.2016.2595913
Measuring the Impact of Memory Errors on Application Performance

4 Measuring the Impact of Memory Errors

We experimentally characterized the impact of correctable memory errors to verify our claim that system-level error management and reporting is a primary cause of degraded application performance. Our hardware fault-injection methodology is described first, before we discuss the empirical results.

4.1 Experimental Methods

We measured the performance impact of memory CEAs on a real Intel Haswell-EP-based cloud server running Windows Server 2012 R2. We expect the behavior of Linux-based machines to be similar, to what we find in this work, although we were not able to adapt our experimental framework and all workloads of interest to function on both platforms. We did not evaluate the relative performance of the reliability, availability, and serviceability (RAS) features in non-faulty situations because they are not well understood and do not explain the denial-of-service effect seen on machines with errors in the field.

Instead, we measured the relative performance impact on the system when errors actually occurred, enabling us to quantify the impact of the error-reporting interface as well as application-level performance with error interrupts enabled and disabled in the BIOS. When interrupts were disabled, we measured no degradation in performance incurred by interrogating the memory faults – even at very high rates – for each of the available hardware RAS techniques (SECDED, SDDC/ChipKill, rank sparing, channel mirroring, etc.). Conversely, the performance degradation when interrupts were enabled depended only on the fault-sensitizing rate – regardless of the RAS scheme. This proved that the firmware/software overhead to report memory errors causes significant performance degradation, and warranted further analysis.

4.2 Empirical Results using Fault Injection

We first verified our hypothesis that the error reporting interface is a major culprit behind performance degradation on machines with memory errors. This was done by measuring raw memory performance as well as application-level performance with error interrupts enabled and disabled in the BIOS.

Typically, memory errors are handled by the operating system by trapping the error, logging it, and reporting it to the user. However, this can be expensive and can lead to significant performance degradation.

Instead, we measured interrupt latencies to ensure only one access could be outstanding at a time. This fault-sensitizing approach is independent of application access behavior. In general, because the performance penalty incurred by interrupts is not necessarily "paid" by the aggressor thread that sensitizes the fault (except in the case of SMMs), the results only depend on the interrupt priorities and how these are set up. The latency for different DRAM fault-sensitizing rates–regardless of the RAS scheme–is shown in Fig. 2. We believe the latter type of scenario is more likely to occur in reality: a hard fault may begin to manifest in a frequently-accessed mechanism, such as a stuck I/O pin in the DDR channel interface. Moreover, there may actually be more errors in practice than those indicated in the logs using fault-injection batch tests. This is because existing service from firmware/software may block the recording of others.

To validate the accuracy of our fault injection approach, we used a variety of faulty memory modules (DIMMs) with known fault patterns that spanned major DRAM manufacturers. The faulty DIMMs consisted of specimens that failed in a production datacenter setting and were characterized after the fact, and of specimens that had failed post-manufacturing screening tests and were graciously provided by each manufacturer for our research needs. We replicated several of the known failure patterns using our fault injection framework on known-good DIMMs. For both the injected and the ground-truth faulty memories, the system-level response was identical: the expected number of interrupts reported by the OS, and the performance impact due to a law of power-law distribution, where a few machines see many errors in a month. However, existing data from the field does not provide sufficient time-resolution information to determine how bursty errors actually are. The relevant timescales of an application for a server, for example, that had one million reported errors in a month might have had them uniformly over time (average 0.38 errors/sec) or as an avalanche during a single hour (average 277 errors/sec). We believe the latter type of scenario is more likely to occur in reality: a hard fault may begin to manifest in a frequently-accessed mechanism, such as a stuck I/O pin in the DDR channel interface. Moreover, there may actually be more errors in practice than those indicated in the logs using fault-injection batch tests. This is because existing service from firmware/software may block the recording of others.

Using a lightly-modified version of X-Mem, our open-source and extensible memory characterization and micro-benchmarking tool [10], we developed a single-threaded DRAM loop that injected a single memory access per thread at a user-controlled constant rate. To ensure every load to the faulty memory actually reached the DRAM (and not just the caches), we flushed the cache line after every access and used memory barriers to ensure only one access could be outstanding at a time. This fault-sensitizing approach is independent of application access behavior. In general, because the performance penalty incurred by interrupts is not necessarily "paid" by the aggressor thread that sensitizes the fault (except in the case of SMMs), the results only depend on the interrupt priorities and how these are set up. The latency for different DRAM fault-sensitizing rates–regardless of the RAS scheme–is shown in Fig. 2. We believe the latter type of scenario is more likely to occur in reality: a hard fault may begin to manifest in a frequently-accessed mechanism, such as a stuck I/O pin in the DDR channel interface. Moreover, there may actually be more errors in practice than those indicated in the logs using fault-injection batch tests. This is because existing service from firmware/software may block the recording of others.

To validate the accuracy of our fault injection approach, we used a variety of faulty memory modules (DIMMs) with known fault patterns that spanned major DRAM manufacturers. The faulty DIMMs consisted of specimens that failed in a production datacenter setting and were characterized after the fact, and of specimens that had failed post-manufacturing screening tests and were graciously provided by each manufacturer for our research needs. We replicated several of the known failure patterns using our fault injection framework on known-good DIMMs. For both the injected and the ground-truth faulty memories, the system-level response was identical: the expected number of interrupts reported by the OS, and the performance impact due to a law of power-law distribution, where a few machines see many errors in a month. However, existing data from the field does not provide sufficient time-resolution information to determine how bursty errors actually are. The relevant timescales of an application for a server, for example, that had one million reported errors in a month might have had them uniformly over time (average 0.38 errors/sec) or as an avalanche during a single hour (average 277 errors/sec). We believe the latter type of scenario is more likely to occur in reality: a hard fault may begin to manifest in a frequently-accessed mechanism, such as a stuck I/O pin in the DDR channel interface. Moreover, there may actually be more errors in practice than those indicated in the logs using fault-injection batch tests. This is because existing service from firmware/software may block the recording of others.

Using a lightly-modified version of X-Mem, our open-source and extensible memory characterization and micro-benchmarking tool [10], we developed a single-threaded DRAM loop that injected a single memory access per thread at a user-controlled constant rate. To ensure every load to the faulty memory actually reached the DRAM (and not just the caches), we flushed the cache line after every access and used memory barriers to ensure only one access could be outstanding at a time. This fault-sensitizing approach is independent of application access behavior. In general, because the performance penalty incurred by interrupts is not necessarily "paid" by the aggressor thread that sensitizes the fault (except in the case of SMMs), the results only depend on the interrupt priorities and how these are set up. The latency for different DRAM fault-sensitizing rates–regardless of the RAS scheme–is shown in Fig. 2. We believe the latter type of scenario is more likely to occur in reality: a hard fault may begin to manifest in a frequently-accessed mechanism, such as a stuck I/O pin in the DDR channel interface. Moreover, there may actually be more errors in practice than those indicated in the logs using fault-injection batch tests. This is because existing service from firmware/software may block the recording of others.

Using a lightly-modified version of X-Mem, our open-source and extensible memory characterization and micro-benchmarking tool [10], we developed a single-threaded DRAM loop that injected a single memory access per thread at a user-controlled constant rate. To ensure every load to the faulty memory actually reached the DRAM (and not just the caches), we flushed the cache line after every access and used memory barriers to ensure only one access could be outstanding at a time. This fault-sensitizing approach is independent of application access behavior. In general, because the performance penalty incurred by interrupts is not necessarily "paid" by the aggressor thread that sensitizes the fault (except in the case of SMMs), the results only depend on the interrupt priorities and how these are set up. The latency for different DRAM fault-sensitizing rates–regardless of the RAS scheme–is shown in Fig. 2. We believe the latter type of scenario is more likely to occur in reality: a hard fault may begin to manifest in a frequently-accessed mechanism, such as a stuck I/O pin in the DDR channel interface. Moreover, there may actually be more errors in practice than those indicated in the logs using fault-injection batch tests. This is because existing service from firmware/software may block the recording of others.
Impact of memory errors on application performance. Given the frequent task pre-emption and possible cache pollution and/or flushes that are caused by SMM, multiple copies of MCF are more likely to interfere in main memory, causing additional performance degradation. In the case of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, we consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.

Impact of error handlers on an interactive web-search application. Finally, consider an interactive web-search workload, which was developed internally. It emulates the index-searching component using real web-search traces. Because of CMCIs, each processor core may not receive a fair share of interrupts. We simulated the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, 80% of the interrupts that a thread running on another core receives. Regardless, we ran all workloads to completion, and found that run-to-run variation was negligible.
Measuring the Impact of Memory Errors on Application Performance
Gottscho et al.
IEEE CAL 2016

Fig. 3. An industrial web-search application running on a state-of-the-art cloud server experiences severe performance degradation in the presence of memory errors. SMI interrupts (133 ms) degrade performance much faster than CMCIs (775 µs) because of their higher handling latencies.

Acknowledgments
This work was conducted jointly between Microsoft Corporation and the NanoCAD Lab at the Electrical Engineering Department at the University of California, Los Angeles (UCLA). The authors thank Dr. Jie Liu of Microsoft Research, and Dr. Badriddine Khessib and Dr. Kushagra Vaid of Microsoft for supporting this work while Mr. Gottscho was an intern at Microsoft Research in 2015. Funding came partly from the NSF Variability Expedition Grant No. CCF-1029783.

References