Measuring the Impact of Memory Errors on Application Performance

Mark Gottscho[1], Mohammed Shoaib[2], Sriram Govindan[3], Bikash Sharma[3], Di Wang[2], and Puneet Gupta[1]

mgottscho@ucla.edu, {shoaib, srgovin, bsharma, wangdi}@microsoft.com, puneet@ee.ucla.edu

[1]Electrical Engineering Department, University of California, Los Angeles (UCLA) [2]Microsoft Research [3]Microsoft

Abstract—Memory reliability is a key factor in the design of warehouse-scale computers. Prior work has focused on the performance overheads of memory fault-tolerance schemes when errors do not occur at all, and when detected but uncorrectable errors occur, which result in machine downtime and loss of availability. We focus on a common third scenario, namely, situations when hard but correctable faults exist in memory; these may cause an "avalanche" of errors to occur on affected hardware. We expose how the hardware/software mechanisms for managing and reporting memory errors can cause severe performance degradation in systems suffering from hardware faults. We inject faults in DRAM on a real cloud server and quantify the single-machine performance degradation for both batch and interactive workloads. We observe that for SPEC CPU2006 benchmarks, memory errors can slow down average execution time by up to 2.5×. For an interactive web-search workload, average query latency degrades by up to 2.3× for a light traffic load, and up to an extreme 3746× under peak load. Our analyses of the memory error-reporting stack reveals architecture, firmware, and software opportunities to improve performance consistency by mitigating the worst-case behavior on faulty hardware.

Index Terms—Main memory, DRAM, error-handling, servers, reliability, availability, RAS, cloud, performance consistency, hardware/software interface

1 Introduction

In datacenters, particularly clouds, failures of servers and their components are a common occurrence and can impact performance and availability for users [35]. Faults can manifest as correctable errors (CEs) that can degrade performance, detected but uncorrectable errors (DUEs) that can cause machine crashes, and undetected errors that often cause silent data corruption (SDC). At best, errors are a nuisance by increasing maintenance costs; at worst, they cause cascading failures in software micro-services, leading to major end-user service outages [25].

Hard faults in main memory DRAM are one of the biggest culprits behind server failures in the field [22], [32], [34], [19], [26], while main memory comprises a significant fraction of datacenter capital and operational expenses [24], [29]. Unfortunately, memory reliability is expected to decrease in future technologies [26] as a result of increasing manufacturing process variability in nanometer nodes [18], [36], [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 [26]. 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 [26], 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. 4). Our measurements show that batch-type SPEC CPU2006 benchmarks suffer an average 2.5× degradation in execution time, while an interactive web-search application can experience up to 100× 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 reliability in memory systems [28]. Many compelling reliability-aware techniques for energy savings in caches and memory have been proposed [6], [7], [8], [23], [21], [30], [11], [15], [14], but none of them have focused on large-scale systems. A recent thread of research studied memory failures in large-scale field studies that characterized broader trends [32], [34], [19], [33], [12], [26], [10], but none of them has addressed the performance impact of memory errors. A recent work has proposed designing datacenter DRAM with heterogeneous reliability for cost reduction [24], but they did not consider the performance implications from increased error rates. A recent work on Software-Defined Error-Correcting Codes [16] proposed a novel set of techniques to opportunistically recover from memory DUEs and resume correct execution, but it did not consider the negative performance effects of memory CEs. 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 maintenance costs and reduce performance consistency. In this paper, we address these gaps in prior work by describing the mechanisms by which performance degrades and empirically demonstrate that the performance degradation on a real cloud server can be severe.

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 [32], [34], [19], [33], [12], [26], [10] 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 E5-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], [5].

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.2016.2599513

typical response in current cloud servers is to cause a kernel panic, which crashes and restarts the entire machine. (Higher-end machines, which are typically not deployed in public clouds, support poisoning, a technique that facilitates recovery attempts from uncorrectable errors instead of crashing.) For a CMCI, the kernel simply logs the DRAM error with a timestamp, which can then be read by user-level software through the system event log. On our system, a CMCI raised by the memory uncore is broadcast to all cores on the socket, but handled by just one thread (which may be statically assigned, as suggested by Intel [2], or as we believe in the case of our platform, dynamically load-balanced within a socket). With CMCI or MCE-based software error reporting, the OS kernel might know the physical address of the error, but generally not the precise location that the error occurred in the DRAM organization. This is because the mapping is complex and dependent on hardware and platform configurations. Thus, using purely software-mode interrupts on supported processors, the OS might use page retirement, although the datacenter operators would generally not know which memory module is failing on a machine that reports many errors. Note that in our platform used for the experiments, the kernel *does not* know the physical address nor the DRAM location of a memory error when using CMCI-based reporting.

Firmware-based error reporting, on the other hand, can determine both the physical address and the precise location of the memory error in DRAM by performing the required platform-specific calculations. This makes it our preferred boot-time option. If the firmware interrupt mode is selected in Intel machines with the Enhanced Machine Check Architecture (EMCA) [5], the ECC hardware raises a maximum-priority System Management Interrupt (SMI). In our platform, an SMI is broadcast to all logical processors across both sockets. All processors that receive SMIs immediately enter the special System Management Mode (SMM) in firmware. SMM raises each processor to the highest-possible machine privilege level. The job of SMM in response to a memory error is to read all relevant registers in the memory controller, compute additional information about the error, and report the error. Because SMM is not re-entrant, whenever a memory error occurs in firmware-first mode, the entire system becomes unresponsive. On our platform, when an SMI is being handled, all operating system and user threads are stalled: no forward progress on any system or application task can be made. Before exiting, SMM constructs an entry with detailed error information in the Advanced Configuration and Power Interface (ACPI) [1] tables and then forwards the error to the OS by raising the appropriate MCE or CMCI interrupt.

4 Measuring the Impact of Memory Errors

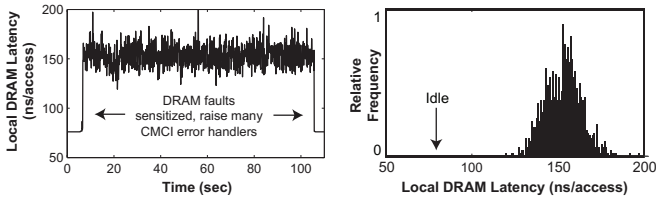
We experimentally characterized the impact of correctable memory errors to verify our claim that system-level error management and reporting is a primary cause of degraded application performance. Our hardware fault-injection methodology is described first, before we discuss the empirical results.

4.1 Experimental Methods

We measured the performance impact of memory CEs 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 well understood and do not explain the denial-of-service effect [26] 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 complete hardware/firmware/software error reporting stack.

DRAM fault injection was physically performed using proprietary hardware and software tools and was controlled by OS and user-level software. The tools have the flexibility to flip specific bit cells with any desired pattern. Before starting the application under test, our framework uses the Windows kernel debugger [3] to perform virtual-to-physical address translation for a special region in the private virtual memory region. The tools are then used to inject a soft single-bit fault into DRAM hardware at a controlled time and location using the translated physical address. Demand and patrol scrubbing were disabled to prevent unintended removal of the injected fault. Note that in production systems, scrubbing is typically enabled; this might contribute additional performance overheads beyond those measured in this paper in the presence of real hard faults.

Using a lightly-modified version of X-Mem, our open-source and extensible memory characterization and micro-benchmarking tool [17], [4], we sensitized a single faulty DRAM location using one 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



(a) Trace (b) Histogram

Fig. 1. CMCI handlers can cause significant memory interference, degrading performance even if the application thread is not directly interrupted. A non-interrupted thread sees an increase of DRAM latency that has additive Gaussian noise. Here, 1000 CMCIs per second worsens average memory latency by about 2X by competing for memory-level parallelism and bandwidth in the L3 cache and DRAM.

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 SMIs), the results only depend on the interrupt performance and the number of error-interrupts per second (as we show in Sec. 4.2, there was no measurable performance degradation caused by the *hardware* RAS technique in the presence of errors, such as SECDED vs. ChipKill). These facts make our experiments tractable to perform for different applications while yielding correct results for different DRAM fault models.

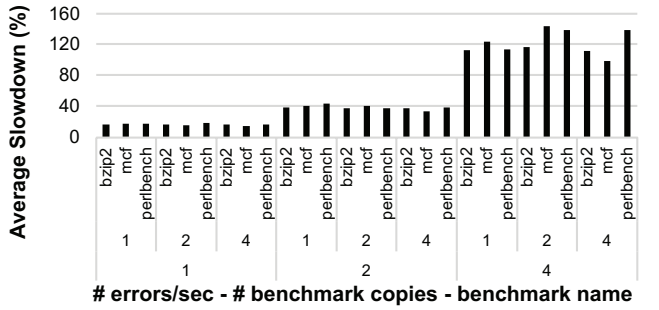
We deliberately swept a wide range of error-interrupt rates in our experiments to capture different scenarios. A rate in the range of 100-1000 error-causing interrupts per second, while seemingly extreme, might actually be common in production datacenters. This is due to a power-law distribution, where a few machines see many errors in a month [26]. However, existing data from the field does not provide sufficient time-resolution information to determine how bursty errors actually are compared to the relevant timescales of an application. For instance, a server that had one million reported errors in a month [26] 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 used by field studies. This is because existing errors that are pending 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 errors reported in the OS, and the performance impacts that we outline in the next subsection were identical. Therefore, all of our reported results use our fault injection framework as a valid substitute for real faulty DIMMs.

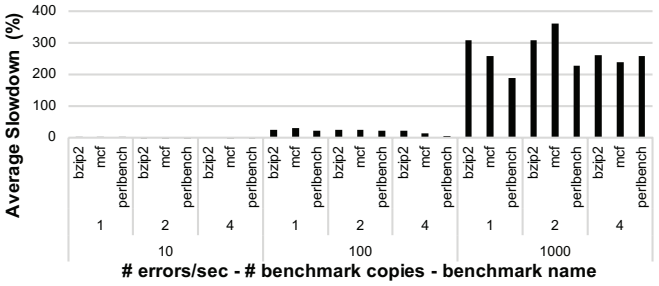
4.2 Empirical Results using Fault Injection

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. When interrupts were disabled, we measured no degradation in performance incurred by sensitizing 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. (Note that forms of memory scrubbing may also cause additional performance degradation in presence of errors, but these were not evaluated due to experimental limitations in our fault injection framework.) We then characterized the latency of the error-reporting interrupts before examining their interference with batch (throughput-oriented) and interactive (latency-oriented) applications.

Error-handling interrupt latencies. We measured interrupt latencies by accessing the faulty DRAM location – located on the same socket as the sensitizing thread – as fast as possible, causing interrupts that flood the whole socket and constantly pre-empt the sensitizing



(a) SMI Reporting



(b) CMCI Reporting

Fig. 2. Empirical results for three different SPEC CPU2006 "batch" applications, show significant performance degradation in the presence of errors. Workloads were run to completion, and run-to-run variation was negligible.

thread. The fault-sensitizing rate was thus limited by the interrupts, allowing us to measure the handler latency directly based on the completed number of memory accesses per second. We found that the SMI latency (133 ms) is up to 171× worse than the CMCI latency (775 μs). This is because the SMI invokes SMM, which must read all the relevant registers to reconstruct the error information, populate the ACPI tables, and then raise a CMCI to the OS before exiting and allowing threads to resume. The implementation of SMM impacts performance even more than the indicated latency by blocking all threads from executing. It also executes slowly due to the way it uses memory [2]. In contrast, CMCIs operate like a conventional interrupt, only pre-empting a single logical core and running in the OS context. Both of these error-reporting interrupts exhibit latencies that are high enough to cause significant application interference.

We studied the memory interference caused by CMCIs further. We measured the native DRAM latency over time using the standard version of our X-Mem tool [17], [4], while our fault injection framework sensitized a DRAM fault to raise 1000 CMCI/sec. The trace and histogram of memory latency is shown in Fig. 1. The flood of CMCI handlers compete for bandwidth in the shared L3 cache and DRAM, adding Gaussian noise to the overall memory latency, which doubled on average. This could interfere with the performance of a victim application, even if it is isolated in a virtualized environment.

Impact of error handlers on batch applications. Given the high interrupt-handling latencies that we measured, we characterized how much performance degradation memory errors can actually cause on real applications. First, we considered the performance of batch applications using three benchmarks from the SPEC CPU2006 suite: `bzip2`, `mcf`, and `perlbench`. The results are shown in Fig. 2(a) when the server is configured to report memory errors using the SMI interrupt, and Fig. 2(b) for the CMCI interrupt. For each benchmark, we varied the number of error-interrupts per second (outer axis labels) and the number of identical copies for each benchmark that were run simultaneously on different cores (inner axis labels).

Performance penalties were significant for both types of interrupt. When using SMIs, the average slowdown was roughly 16% for all three benchmarks with just a single error-interrupt per second. With two SMIs per second, the penalty rose to approximately 36%, and with four SMIs per second, the average penalty was almost 115%. For the SMI cases, the applications behave as though they were duty-cycled at a rate of one minus the average utilization consumed by error handlers. For CMCI reporting, performance degradation is negligible for ten error-interrupts per second. At 1000 error-interrupts/sec with CMCIs, our batch applications perform approximately 200% to 350% worse.

As the error rate increases, performance varies considerably within and across the three applications for both types of interrupts. We believe this is caused by two factors. *(i)* As we noted earlier, high error rates cause increased memory interference; this affects each application differently. For example, in the SMI case, `bzip2` has very consistent performance no matter how many copies run, while `mcf` shows more variation because it is a memory-heavy workload. 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. *(ii)* In the case of CMCIs, each processor core may not receive a fair share of interrupts. Without knowing the interrupt load-balancing policy taken by the kernel, a thread running on one core might receive, for example, only 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.

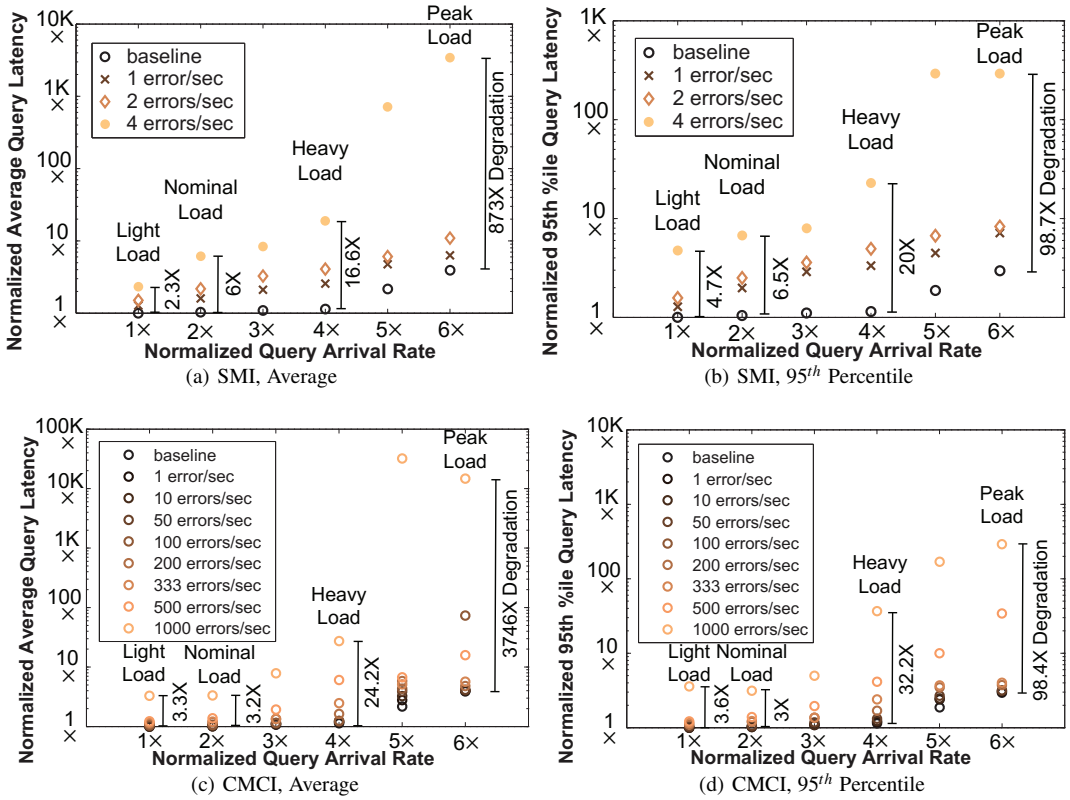
Impact of error handlers on an interactive web-search application. Finally, we consider an interactive web-search workload, which was developed internally. It emulates the index-searching component of a major production search engine using real web-search traces. Fig. 3 depicts the normalized average and 95^{th} percentile query latency as a function of the normalized query arrival rate and the number of DRAM error-interrupts per second. We find that even for light loads, error-interrupt rates of four SMIs per second can cause 2.3× higher average query latency. Under peak search traffic, the introduction of just a few memory errors to the system causes the average query latency to increase by up to a staggering 873× (Fig. 3(a)). We see similar trends in the tail latency (Fig. 3(b)). In contrast, CMCI-based error reporting does not result in significant performance penalties for up to 50 error-interrupts per second. CMCIs can still deny service completely, however, just like SMIs: up to 3746× degradation can occur in average search latency under peak load with 1000 error-interrupts per second (Figs. 3(c) and 3(d)). Note that the relative degradation in tail latency is usually greater than the average for the lightly loaded cases, but less than the average for the heavily-loaded cases. This implies that error-interrupts can *completely* deny service in the worst case, which will cause the average latency to exceed the tail latency.

The interactive application is much more sensitive to errors than the batch applications because its performance metric (tail latency) is dependent on the worst-case interference of interrupt handlers with user event handlers, i.e., the timing of events. Conversely, the batch applications' performance metric (overall execution time) is linearly dependent on the number of interrupts and their total handling duration, but not the arrival times of the errors. These results highlight the need for further empirical evaluation and modeling by the community in order to develop compelling solutions to the problem of memory errors.

Possible solutions. There are a number of ways to mitigate this issue. *(i)* Dramatically speed up interrupt handlers through firmware/software optimization. *(ii)* Change SMM architecture to allow just a single processor to execute in firmware mode concurrently with the other processors in kernel or user mode, improving performance of SMI-based error handling. However, this may have implications for firmware complexity as well as platform security. *(iii)* Expose the complete and static physical-to-DRAM organization mapping at boot time from firmware to OS through a new ACPI structure that is cached in the kernel, improving utility of CMCI-based error handling. This would add complexity to both firmware and system software. *(iv)* Leverage aggressive page retirement to remove the source of faults that are sensitized and degrading performance. This could potentially cost capacity and memory-level parallelism. *(v)* Finally, revert to the older error polling-style method, which bounds the time spent handling errors, but potentially reduces the fidelity of error logs for use by the OS, datacenter operators, and field study researchers. It may also reduce the common-case performance of all machines in a WSC, which may affect the tail latency of interactive applications. *(vi)* Leverage application-level load-balancing in the datacenter to mitigate the denied service of machines with memory errors. This might have limited use, however, for applications that are composed of many microservices, like those at Google [20]. We leave these solutions, as well as studying the impact of memory errors on datacenter-level performance consistency and availability, to future work.

5 Conclusion

Memory reliability will continue to be an important consideration in the design of WSCs for the foreseeable future. As researchers explore new directions in reliability-aware design, variation-tolerant systems, and approximate computing, we advocate for increased awareness of the more subtle effects of reliability on performance, energy, and cost. Our empirical measurements confirmed our hypothesis: a primary cause of performance degradation witnessed on machines with faulty memory is not the underlying hardware itself; rather, it is the extreme software overheads that are required to manage and report the error occurrences in the system. Compelling directions for future work include the design of improved error-handling architectures, characterizing the impact of memory errors on performance of distributed



(a) SMI, Average (b) SMI, 95th Percentile

(c) CMCI, Average (d) CMCI, 95th Percentile

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.

and microservice-based applications across multiple machines, and accounting for the performance degradation in machine servicing and datacenter total cost-of-ownership models. Our ongoing work aims to analytically model the observed performance degradation using queuing theory in pursuit of these goals.

Acknowledgments

This work was conducted jointly between Microsoft Corporation and the NanoCAD Lab of 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

[1] "Advanced Configuration and Power Interface (ACPI) Specification," www.uefi.org, accessed: 2015-06.
[2] "Intel 64 and IA-32 Architectures Software Developer Manuals," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, accessed: 2015-05-01.
[3] "Microsoft WinDbg," https://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx, accessed: 2015-06.
[4] "X-Mem Source Code," https://nanocad-lab.github.io/X-Mem/, accessed: 2016-07.
[5] "MCA Enhancements in Future Intel Xeon Processors," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/enhanced-mca-logging-xeon-paper.pdf, 2013.
[6] J. Abella *et al.*, "Low Vccmin Fault-Tolerant Cache with Highly Predictable Performance," in *MICRO*, 2009.
[7] A. R. Alameldeen *et al.*, "Adaptive Cache Design to Enable Reliable Low-Voltage Operation," *TC*, vol. 60, no. 1, 2011.
[8] A. Ansari *et al.*, "Archipelago: A Polymorphic Cache Design for Enabling Robust Near-Threshold Operation," in *HPCA*, 2011.
[9] M. Armbrust *et al.*, "A View of Cloud Computing," *CACM*, vol. 53, no. 4, 2010.
[10] E. Baseman *et al.*, "Improving DRAM Fault Characterization Through Machine Learning," in *DSN-W*, 2016.
[11] L. Chen *et al.*, "E3CC: A Memory Error Protection Scheme with Novel Address Mapping for Subranked and Low-Power Memories," *TACO*, vol. 10, no. 4, 2013.
[12] N. DeBardeleben *et al.*, "Extra Bits on SRAM and DRAM Errors - More Data from the Field," in *SELSE*, 2014.
[13] B. Delgado and K. L. Karavanic, "Performance Implications of System Management Mode," in *IISWC*, 2013.
[14] M. Gottscho *et al.*, "DPCS: Dynamic Power/Capacity Scaling for SRAM Caches in the Nanoscale Era," *TACO*, vol. 12, no. 3, 2015.
[15] ——, "ViPZonE: Hardware Power Variability-Aware Memory Management for Energy Savings," *TC*, vol. 64, no. 5, 2015.
[16] ——, "Software-Defined Error-Correcting Codes," in *DSN-W*, 2016.
[17] ——, "X-Mem: A Cross-Platform and Extensible Memory Characterization Tool for the Cloud," in *ISPASS*, 2016.
[18] P. Gupta *et al.*, "Underdesigned and Opportunistic Computing in Presence of Hardware Variability," *TCAD*, vol. 32, no. 1, 2013.
[19] A. A. Hwang *et al.*, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," *SIGARCH Comp. Arch. News*, vol. 40, no. 1, 2012.
[20] S. Kanev *et al.*, "Profiling a Warehouse-Scale Computer," in *ISCA*, 2015.
[21] S. Li *et al.*, "System Implications of Memory Reliability in Exascale Computing," in *SC*, 2011.
[22] X. Li *et al.*, "A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility," in *USENIX ATC*, 2010.
[23] S. Liu *et al.*, "Flikker: Saving DRAM Refresh-Power Through Critical Data Partitioning," *SIGARCH Comp. Arch. News*, vol. 39, no. 1, 2011.
[24] Y. Luo *et al.*, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *DSN*, 2014.
[25] B. Maurer, "Fail at Scale," *CACM*, vol. 58, no. 11, Nov. 2015.
[26] J. Meza *et al.*, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *DSN*, 2015.
[27] S. Mittal, "A Survey of Architectural Techniques for Managing Process Variation," *ACM Comp. Surv.*, vol. 48, no. 4, 2016.
[28] S. Mittal and J. S. Vetter, "A Survey of Techniques for Modeling and Improving Reliability of Computing Systems," *TPDS*, vol. 27, no. 4, 2016.
[29] P. Nikolaou *et al.*, "Modeling the Implications of DRAM Failures and Protection Techniques on Datacenter TCO," in *MICRO*, 2015.
[30] M. K. Qureshi and Z. Chishti, "Operating SECDED-based Caches at Ultra-Low Voltage with FLAIR," in *DSN*, 2013.
[31] A. Rahimi *et al.*, "Variability Mitigation in Nanometer CMOS Integrated Systems: A Survey of Techniques From Circuits to Software," *Proc. IEEE*, vol. 104, no. 7, 2016.
[32] B. Schroeder *et al.*, "DRAM Errors in the Wild: A Large-Scale Field Study," in *SIGMETRICS*, 2009.
[33] V. Sridharan *et al.*, "Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults," in *SC*, 2013.
[34] V. Sridharan and D. Liberty, "A Study of DRAM Failures in the Field," in *SC*, 2012.
[35] K. V. Vishwanath and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability," in *SoCC*, 2010.
[36] L. Wanner *et al.*, "NSF Expedition on Variability-Aware Software: Recent Results and Contributions," *De Gruyter it*, vol. 57, no. 3, 2015.