SAND2021-9208C

Understanding the Effects of DRAM Correctable Error Logging at Scale

Kurt B. Ferreira, Scott Levy & Victor Kuhns
Center for Computing Research
Sandia National Laboratories
{kbferre,sllevy,vgkuhns}@sandia.gov

Nathan DeBardeleben & Sean Blanchard
Ultrascale Systems Research Center
Los Alamos National Laboratory
{ndebard,seanb}@lanl.gov

Abstract—**Fault tolerance poses a major challenge for future large-scale systems. Current research on fault tolerance has been principally focused on mitigating the impact of uncorrectable errors: errors that corrupt the state of the machine and require a restart from a known good state. However, correctable errors occur much more frequently than uncorrectable errors and may be even more common on future systems. Although an application can safely continue to execute when correctable errors occur, recovery from a correctable error requires the error to be corrected and, in most cases, information about its occurrence to be logged. The potential performance impact of these recovery activities has not been extensively studied in HPC.**

In this paper, we use simulation to examine the relationship between recovery from correctable errors and application performance for several important extreme-scale workloads. Our paper contains what is, to the best of our knowledge, the first detailed analysis of the impact of correctable errors on application performance. Our study shows that correctable errors can have significant impact on application performance for future systems. We also find that although the focus on correctable errors is focused on reducing failure rates, reducing the time required to log individual errors may have a greater impact on overheads at scale. Finally, this study outlines the error frequency and durations targets to keep correctable overheads similar to that of today's systems. This paper provides critical analysis and insight into the overheads of correctable errors and provides practical advice to systems administrators and hardware designers in an effort to fine-tune performance to application and system characteristics.

I. Introduction

Maintaining the performance of high-performance computing (HPC) applications as failures become more and more frequent is a major challenge that needs to be addressed for next-generation extreme-scale systems. Recent studies have demonstrated that hardware failures are expected to become ever more common [1]. Increasing the scale of HPC systems requires the aggregation of more individual components. More components means more frequent failures. Additionally, advances in memory device design may exacerbate this trend. Reductions in device-feature sizes and near-threshold supply voltages may mean that next-generation DRAM devices are less reliable than devices that are currently deployed in large-scale HPC systems [1]. Understanding the implications of these trends requires that we have detailed knowledge of how DRAM errors affect current leadership-class systems.

Recent research has largely focused on uncorrectable and fatal errors: errors that require applications to be restarted from a known good state.[1] However, the impacts of the most common type of memory errors in large-scale systems, correctable errors (CE), have largely been overlooked. An analysis of failures on recent leadership-class systems shows that the correctable error rates are 20 times higher than uncorrectable errors [2]. Correctable errors are corrected in hardware and are essentially invisible to the affected application, except for the logging of the error. While applications can continue to make progress despite the presence of correctable errors (i.e. restarting the application is unnecessary), the time required to correct and log these errors has the potential to impact application performance by delaying application computation.

In this paper, we present a detailed analysis of the relationship between the cost of correcting and logging memory CEs and application performance on large-scale systems. Specifically, to better understand the potential performance impact of CEs, we answer the following key questions:

- What is the expected performance impact of CEs on applications running on current and projected future extreme-scale systems?
- How frequently can CEs occur without significantly degrading application performance?
- What is the application performance impact of CEs that are isolated to a single process?
- How can system designers address CEs to utilize future DRAM hardware while improving application performance on next-generation systems?

We address these questions by using a validated simulation framework to study the impact of correcting and logging correctable errors on the performance of important HPC workloads. Simulation enables us to perform experiments that would be difficult or impossible on real hardware. Based on the data and analysis presented in this paper, we make the following contributions:

- we confirm the costs associated with the logging of correctable errors (§IV-A);
- we show how CEs from just one single node can significantly impact application performance (§IV-B)

[1]Linux uses several mitigation strategies, e.g., page offlining, when the error affects a page whose contents can be safely reconstructed or discarded. However, even given mitigation, uncorrectable and fatal errors frequently necessitate a node reboot and an application restart.

- we demonstrate the impacts of the correction and logging correctable errors on current systems is modest and project how these costs may impact applications on future exascale class systems, outlining reliability targets to keep overheads minimal (§§IV-C,IV-D); and,
- we examine the relationship between the correctable error rate and the time required to correct and log the errors, showing that if per-event overheads for CEs can be kept low, very high CE rates can occur without significant application impacts (§IV-E).

We note that our analysis only accounts for the CPU impact of the correctable logging overheads and not the less significant impacts of a potential decrease in memory bandwidth and other side effects. Therefore, our analysis represents a lower bound on the overall possible application impacts. For applications that are sensitive to memory bandwidth, correctable overheads may be more significant.

Overall, this paper provides critical insight into the overheads of correctable errors and provides practical advice to users and systems administrators in an effort to fine-tune performance to application and system characteristics.

II. Background

A. CPU Memory Error Detection and Correction

Error detection and handling is critical to pinpointing failing components and taking corrective action in a timely fashion. Error handling is typically a cooperative activity between the platform hardware, firmware (UEFI or BIOS), and the host operating system. Errors are processed in software by the OS or in firmware [3]. Firmware error processing allows the firmware to collect detailed information about where the error occurred that is not available in the processor. For example, the processor provides the physical address where the error occurred. However, it may not be possible for the processor to use the physical address to identify the precise DRAM device that failed. Identifying the DRAM devices enables the firmware to take corrective action to avoid the error in the future[4]. When firmware decoding is enabled, the processor signals the occurrence of an error to the firmware with a System Management Interrupt (SMI). Based on information obtained from the processor and other hardware components, the firmware creates a detailed description of the error and notifies the OS of its occurrence. Alternatively, when software processing is enabled, the processor generates a Machine Check Exception that is handled in software by the OS. Although the affected workload can continue to execute, the time spent correcting and collecting information about correctable errors can perturb application progress.

Memory errors are typically classified into three categories: Correctable, Uncorrectable, and Fatal. Correctable errors (CE) are errors that can be corrected or mitigated in hardware such that the platform's state is the same as it would have been in if no error had occurred. An example of a CE is a single bit (or single symbol in the case of chipkill) error. Detected, uncorrectable errors (DUE) are those errors that were detected by hardware, but could not be corrected. Multi-bit/multi-symbol errors are an example of DUEs. The system may continue to execute but the application may need to restart in order to recover lost state.[2] Fatal errors corrupt the state of the processor such that continued correct operation can no longer be guaranteed. Recovering from a fatal error typically requires the processor to be halted and the node rebooted.

B. GPU Memory Error Detection and Correction

A large chunk of the systems in the TOP500 list utilize an accelerator like the Nvidia GPUs [7].[3] The Tesla V100 GPUs, which are used in Summit and Sierra, protect critical memory structures (Streaming Multiprocessor (SM) register files, L1 cache, L2 cache, and main memory) with a Single-Error Correcting Double-Error Detecting (SECDED) Error-Correcting Code (ECC). Single-bit errors (SBE) are corrected by the hardware. Applications are notified of double-bit errors (DBE) so that they may gracefully exit and the GPU halts [8].

Limited information is available about the occurrence of errors detected by the SECDED ECC. Counts of the number of SBEs can be obtained via the Nvidia Management Library (NVML). We are not aware of existing empirical data regarding the time required to handle a correctable error in GPU memory. Moreover, because GPUs lack facilities for error injection like those found on CPUs, *cf.* III-A, it is not currently feasible to collect this data. As a result, we do not examine the impact of correctable memory error logging on GPUs in this paper.[4]

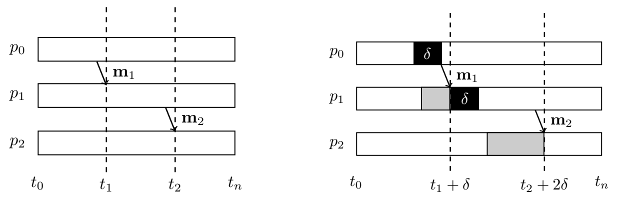
C. Operating System Noise

Delays introduced in application processes by CE-related correction and logging activities are analogous to operating system noise (or *jitter*), which may affect the performance of large-scale applications [9], [10]. Figure 1 illustrates this phenomenon. Figure 1a shows a fixed interval ($t_0 - t_n$) of an application running on three processes (p_0, p_1, and p_2) in the absence of CEs. These three processes exchange two messages, $\mathbf{m}_1$ and $\mathbf{m}_2$. For the purposes of this figure, we assume that messages represent strict dependencies: any delay in message arrival requires the recipient to stall until the message is received. Figure 1b illustrates the potential impact of CE correction and logging. If p_0 encounters a CE just before it would have otherwise sent $\mathbf{m}_1$, then p_1 must wait (the waiting period is shown in grey) until the message arrives. If p_1 subsequently encounters a CE before sending $\mathbf{m}_2$, then p_2 must wait. Part of the time that p_2 spends waiting is due to a delay that originated at p_0, which it does not communicate directly with. In other words, delays incurred handling CEs

[2]Many techniques for avoiding restarts have been explored, *see e.g.*, [5], [6], but to the best of our knowledge none of these approaches have been widely deployed.

[3]For several systems on the list, Intel Xeon Phi processors are identified as accelerators. However, the Intel Xeon Phi processors that Trinity (Los Alamos National Laboratory) and Cori (Lawrence Berkeley National Laboratory) are not identified as accelerators).

[4]Given the very limited information (e.g., neither memory addresses nor timestamps are recorded for SBEs) that is logged, we expect that the impact of SBE logging on application performance will be much more modest than we observe for correctable memory errors on CPUs.



(a) without CE activity (b) with local CE activity delays

Fig. 1: To illustrate how delays introduced by local correctable error (CE) activities may propagate along application communication dependencies, this figure shows a fixed interval ($t_0 - t_n$) of execution for three processes: p_1, p_2, and p_3. These processes exchange two messages, m_1 and m_2. For illustration purposes, the messages in this figures are assumed to create strict dependencies (i.e., any delay in the arrival of a message will cause the receiver to stall). The black regions marked with a white δ denote the execution of CE mitigation activities. The grey regions denote periods in which the execution of a process is stalled due to an unsatisfied communication dependency.

on one process may propagate to other processes along the application's communication dependencies.

III. Experimental Approach

A. APEI Error Injection

To develop a detailed understanding of the cost of recovering from correctable errors, we collected empirical data using ACPI Platform Error Interfaces (APEI) Error Injection, *see* Section IV-A. APEI is part of the Advanced Configuration and Power Interface (ACPI). ACPI is a standard that defines how operating systems interact with the hardware components that comprise the systems on which they run. APEI defines an interface by which the operating system can be notified of errors. The APEI also provides a mechanism for injecting hardware errors: the error injection table (EINJ).

EINJ provides a platform-independent interface that enables the operating system to inject hardware errors without requiring platform-specific software support. The primary objective of this mechanism is to validate operating system Reliability, Availability, and Serviceability (RAS) features. The ACPI specification defines several error types for EINJ, including correctable and uncorrectable errors from DRAM memory, the processor, and the PCI interface. EINJ is not supported on every platform; it requires support from the host processor, OS, and firmware/BIOS. Moreover, platforms that support EINJ may not support all possible error types. For example, our test platform (*see* Section IV-A) supports only the *Memory Correctable* and *Memory Uncorrectable* error types.

Using the EINJ table to inject errors on Linux-based systems is accomplished by writing to virtual files in the sysfs.[5] Using this interface, the user can specify the error type and target memory address, and trigger the injection of the error.

[5]All of the relevant virtual files are in the `/sys/kernel/debug/apei/einj` directory. Additional information about Linux EINJ support, including a simple example of injecting an error, is available in the Linux kernel documentation, *see* [11].

B. Memory Failure Logging

DRAM on modern HPC systems is protected by ECCs. When the memory controller detects a memory error, it attempts to use the ECC to correct the error. If it is able to correct the error, the error is recorded as a CE. If it is unable to correct the error, the error is recorded as a DUE. On x86-based processors, correctable errors are recorded in registers provided by the x86 Machine Check Architecture (MCA) [12], [4]. These registers are polled periodically and their contents are used to record detailed information about the occurrence of CEs in the console log. This information includes the physical address where the error occurred and ECC syndrome data that describes the cause of the error. Decoding the information recorded for each error allows us to identify the physical location of each logged error, but the decoding process takes time, perturbing application performance.

For CEs, APEI supports two types of processor notification: software-based (or OS-based) and firmware-based. In the case of software-based notification, a Corrected Machine Check Interrupt (CMCI) [4], [3] is generated which records the DRAM error and the time of its occurrence. However, it may be difficult to determine the precise DRAM location of an error because of complexities related to memory organization [3]. As a result, mitigating errors with memory page retirement [13] may not always be possible with CMCI-based reporting. With firmware-based notification, the information recorded when the error occurs includes the physical address and the specific DRAM device where the error occurred. Firmware-based notification relies on the Enhanced Machine Check Architecture (EMCA) [4] and is independent of the underlying OS running on the node. While this method allows for precise identification of the source of the error, it is also comparatively more expensive. It requires the system to enter System Management Mode (SMM) which halts all forward progress on *all* cores of the processor while the memory configuration information is assembled and passed to system software [3]. Modern processors like the Intel Skylake processor typically require firmware-based notification to enable advanced RAS features. Therefore, the performance impacts of this error model are important to the HPC community.

To measure the system impacts of the memory decoding and logging overheads, we use the `selfish` [9] system noise measurement microbenchmark. `selfish` tracks the periods of time (*detours*) when the CPU is taken from the application to perform system tasks. It detects detours by continuously reading the processor timestamp counter (TSC) [14]. When the counter interval exceeds a user-defined threshold,[6] the time and duration of this detour is recorded. To ensure that the we accurately captured the impact of injecting errors with EINJ, we used `taskset()` to bind each of the `selfish` threads and the kernel to specific cores and collected selfish traces

[6]For the data presented in this section, we used 150 nanoseconds

on all 48 cores. By running selfish while we inject errors using EINJ we can measure the time required to recover from a correctable error.

C. Simulating Correctable Overheads

In general, communication in Message Passing Interface (MPI) programs cannot be determined offline because message matches cannot be established statically [15]. This makes analytically modeling application performance challenging even if all parameters of the application are known. We therefore use a validated discrete-event simulation framework to evaluate the impact of local correctable error mitigation activities on application performance.

Our simulation-based approach models CE mitigation activities as CPU detours: periods of time during which application progress is blocked by CE handling. We measure these detours using a well known microbenchmark while injecting correctable errors on the system (*see* Section IV-A). This approach allows a level of fidelity and control not always possible in implementation-based approaches. It also allows us to examine simulated systems much larger than those generally available.

Our simulation framework is based on the freely available LogGOPSim [16] and the tool chain described by Levy et al. [17]. LogGOPSim uses the LogGOPS model, an extension of the well-known LogP model [18], to account for the temporal cost of communication events. An application's communication events are generated from traces of the application's execution. These traces contain the sequence of MPI operations invoked by each application process. LogGOPSim uses these traces to reproduce all communication dependencies, including indirect dependencies between processes which do not communicate directly.

LogGOPSim can also extrapolate traces; a trace collected by running the application with p processes can be extrapolated to simulate performance of the application running with $k \cdot p$ processes. The extrapolation produces exact communication patterns for MPI collective operations and approximates point-to-point communications [16]. The validation of LogGOPSim and its trace extrapolation features have been documented previously [16], [9], [17], along with the simulator's ability to accurately predict application performance in the presence of performance perturbations [19], [17], [9].

D. Simulation Setup and Repeatability

We model correctable errors using an extension of the OS noise injection functionality provided by LogGOPSim. Our extension programmatically injects detours that represent correctable errors. The timing of each simulated correctable error is determined statistically using random numbers drawn from an exponential distribution. The mean of the distribution (i.e., the mean time between correctable errors) is based on data regarding the frequency of correctable errors in existing publications, *see e.g.*, [20], [21], [22], [23], [24]. The duration of the detour is determined by the amount of time required to recover from a correctable error. For the experiments in this paper, we rely on empirical results described in this paper (*see* Section IV-A) and data published elsewhere [3] for these values. Application processes are delayed appropriately when a simulated correctable error occurs. Delays that occur on one application process have the potential to propagate along communication dependencies and introduce delays in other processes, *cf.* Figure 1.

To generate the data presented in this paper, we collected execution traces for 128 process (125 process for LULESH, 64 for LAMMPS-crack) single-threaded runs on Mutrino of each of the workloads described in Table I. Mutrino is a Cray XC40 that is a development system for Trinity. Like Trinity it is composed of two partitions: one consisting of compute nodes built around Intel Haswell processors and one consisting of compute nodes built around Intel Knights Landing processors. All of the traces collected for this paper were collected on the Haswell partition.[7] We used these traces to simulate the execution of each workload in the presence of correctable errors. For all of the data presented in this paper, we configured LogGOPSim to extrapolate these traces to simulate one MPI process per node for the systems in TABLE II. We also configured it to use the network parameters collected on a Cray XC40 system, *see* [25]. The accuracy of LogGOPSim simulations has been verified [16] and, for certain applications, has been shown to be within 6% of the actual execution time [26].

We examine the performance of seven HPC workloads. These workloads, described in Table I, include three important DOE production applications (LAMMPS, CTH, and SPARC), an important HPC benchmark (HPCG), a proxy application (LULESH) from the Department of Energy's Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx), a scientific code used to study the behavior of subatomic particles (MILC), and a mini-application (miniFE) from Sandia's Mantevo suite. This diverse set of workloads captures a wide range of computational methods and application behaviors. It additionally captures a significant cross-section of the scalable, high-performance applications that are run on current extreme-scale systems as well as workloads that represent the computational patterns that are expected to be run on future systems.

E. Simulated Correctable Error Rates

In this section, we discuss the system parameters we used in our simulation-based evaluation of the overheads of CEs in the remainder of the paper. This evaluation will examine overheads for a number of key systems as well as an evaluation for a *strawman* exascale-class system. For these evaluations we need an understanding of the correctable errors rates observed in literature as well as a method for projecting these rates onto future exascale class systems with the understanding that the reliability of future DRAM is likely to decrease due to feature size reductions, power concerns, and DRAM ECC

[7]The traces of the open-source workloads, i.e., all of the workloads except CTH and SPARC, are publicly available online at: (*redacted for double-blind review*).

Application	Description
LAMMPS	A classical molecular dynamics simulator from Sandia National Laboratories [27], [28]. The data presented in this paper are from experiments that use the Lennard-Jones (LAMMPS-lj), SNAP (LAMMPS-snap), and Crack (LAMMPS-crack) potentials.
LULESH	A proxy application that approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh [29].
HPCG	A benchmark that generates and solves a synthetic 3D sparse linear system using a local symmetric Gauss-Seidel preconditioned conjugate gradient method [30].
CTH	A shock physics code [31], [32] developed at Sandia National Laboratories. We used an input file that describes the detonation of a conical explosive charge (CTH-st) to collect the data presented in this paper.
MILC	Numerical simulation for the study of quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics [33].
miniFE	A proxy application that captures the key behaviors of unstructured implicit finite element codes [34].
SPARC	SPARC [35] is a next-generation compressible computational fluid dynamics (CFD) code developed by Sandia National Laboratories. We used the "Generic Reentry Vehicle" (GRV) input problem to collect the data presented in this paper.

TABLE I: Descriptions of the workloads used in evaluation.

System	CEs / node / year	GiB / node	CEs / GiB / year	*MTBCE_{node}* (seconds)	Nodes	Simulated Nodes
Google [36]	22,696	1–4	11,384	1368	-	-
Facebook [2]	5,964	2–24	460 (median 108)	5292	-	-
Cielo [24]	26.35	32	0.82	1.2×10^6	8,894	8,192
Trinity [37] (w/ CE_{Cielo})	89.6	128	0.82	311,400	19,420	16,384
Summit [38] (w/ CE_{Cielo})	425.6	608	0.82	62,280	4,608	4,096
Exascale (w/ CE_{Cielo})	574	700	0.82	55,440	16,384	16,384
Exascale (w/ $CE_{Cielo\times10}$)	5,740	700	8.2	5,544	16,384	16,384
Exascale (w/ $CE_{Cielo\times20}$)	11,480	700	16.4	3,024	16,384	16,384
Exascale (w/ $CE_{Cielo\times100}$)	57,400	700	82	554.4	16,384	16,384
Exascale (w/ $CE_{median(Facebook)}$)	75,600	700	108	432	16,384	16,384

TABLE II: Measured and hypothesized correctable error parameters used in this work. $MTBCE_{node}$ is defined as the mean time in hours between correctable errors on a node. The number of simulated processes for LULESH is the nearest power-of-2 multiple of 125 to the number of simulated nodes used for the other workloads, e.g., 16,000 instead of 16,384.

technology concerns. Therefore, for an exascale-class system we are trying to evaluate how much correctable error rates can increase while still keeping overheads to an acceptable level.

Table II outlines the correctable errors parameters used in the remainder of this work. For each system, we define the mean time between correctable errors per node (denoted $MTBCE_{node}$). $MTBCE_{node}$ is calculated from values in the table. The first three rows of the table, are measured correctable error rates from recent data centers (Google [36], Facebook [2], and the Cielo supercomputer located at Los Alamos National Labs [24]). One interesting result from these three studies is the significant increase of reliability for newer generation memory systems. For example, the Google study measured an average of $11,384$ CEs per gigabyte of DRAM per year, while the Facebook and Cielo studies measured an average 460 and 333 CEs/GiB/year, respectively. It is unclear how much of this increase in reliability is due to process changes and how much is due to technology changes. For the Facebook and Google studies, much of the DRAM was protected by SECDED ECC (single-bit error correction, double-bit error detection error correcting code), while the Cielo system used chipkill-correct ECC, which allows single DRAM device correction, double-device error detection.

The next two rows of the table outline the CE parameters for the HPC systems Summit and Trinity. As the CE rates for Trinity and Summit are not publicly available, we use the per gigabyte rates measured from the Cielo study [24]. We believe this is a reasonable assumption as all three systems utilize chipkill-correct memory protection.

The remainder of the table defines parameters for a number of hypothetical exascale class systems evaluated in this work. For each of these exascale systems we assume $16,384$ nodes and 700 GiB of DRAM memory. What varies for each of these systems is the per gigabyte CE rate. As one goal of this work is to provide guidance on how much the CE rate can increase without impacting performance, we start as a baseline the rate measured on Cielo (the most reliable in available literature). As memory error rates are expected to decrease for exascale systems (x4 chipkill is unlikely for exascale due to power concerns and decreased feature sizes may increase error rates), we also examine increased error rates of those measured on Cielo: $10\times$, $20\times$, and $100\times$ Cielo (denoted $CE_{Cielo\times10}$, $CE_{Cielo\times20}$, $CE_{Cielo\times100}$), and the median of the rate measured by Meza *et al.*[2] (denoted $CE_{median(Facebook)}$) which corresponds to a rate of about 120X of that measured on Cielo.

IV. Results

In this section we examine the potential impact of DRAM correctable errors on workload performance. The data presented in this section were collected using the experimental methodology described in Section III.

A. DRAM Correctable Costs

In this section, we evaluate the time required to handle DRAM CEs on a modern HPC system. Specifically, we consider the time required for: hardware error correction (i.e., how long it takes to correct an error using ECC), and the time required to decode and log the error. Decoding and logging can be performed in software by the OS, or in firmware.

All of the data presented in this section was measured on Blake, a 48-node Linux cluster with an Intel OmniPath interconnect network. Each compute node consists of 4 sockets. Each socket is occupied by a 24-core, 2.1GHz Intel Skylake processor (a total of 96 cores/node). Each node also has 192GB of DDR4 DRAM. It is running Red Hat Enterprise Linux Server release 7.4 with a Linux kernel, version 3.10.693.

Using Blake, we collected empirical data on the time required to handle CEs using APEI EINJ as described in Sections III-A and III-B. To establish a baseline, we measured the background noise signature of Blake. The results of this experiment are shown in Fig. 2a. As described in Section III-B, the data in this figure were collected using `selfish`. Each bar in this figure represents a detour identified by `selfish`: its height corresponds to the duration of the detour and its position on the x-axis corresponds to the time when the detour occurred. We also measured the noise signature of the system during "dry-run" error injections. In this case, we wrote to the APEI EINJ virtual files in the Linux sysfs to configure the error to be injected but do not actually trigger injection of the error. We repeat this process every 10 seconds over the duration of the experiment. The results of this experiment are shown in Fig. 2b. Comparing Figures 2a and 2b reveals that configuring error injection via the interface provided by APEI EINJ results in minimal disruption of the processor. In other words, our measurements are not significantly affected by the act of configuring error injection.

To measure the cost of decoding and logging correctable errors in software using CMCI, we configured the processor to generate Corrected Machine-Check Error Interrupts (CMCI) when CEs occur [14, Volume 3 §15.5]. We also set the CMCI Error Threshold to 1 (i.e., every correctable error generates a CMCI interrupt) and triggered an error injection every 10 seconds using the APEI EINJ interface. The results of this experiment are shown in Fig. 2c. The tallest bars (approximately $700\mu s$) occur every 10 seconds and represent the cost of decoding and logging the occurrence of the error in software. To measure the cost of decoding the error in firmware we configured the processor to enable Intel's Enhanced MCA Memory Logging [4]. We configured the correctable error threshold in firmware to 10 (i.e., only every 10^{th} correctable error is logged). The results of this experiment are shown in Fig. 2d. In this figure, the taller bars fall into two groups. The first group of detours occur every 10 seconds and are approximately 7 milliseconds in duration. These detours represent the delay introduced by a System Management Interrupt (SMI) that is generated each time a correctable error occurs [4, §1.2.4]. The second group of detours occur every 100 seconds and are approximately 500 milliseconds in duration. These detours represent the time required by the firmware to decode and log the occurrence of the correctable error.

Missing from this figure is the "All logging turned off" noise signatures, denoting the overheads of the hardware correction mechanisms. This data was collected and was similar to the "Native" and "Dry Run" figures

B. Single Process CEs

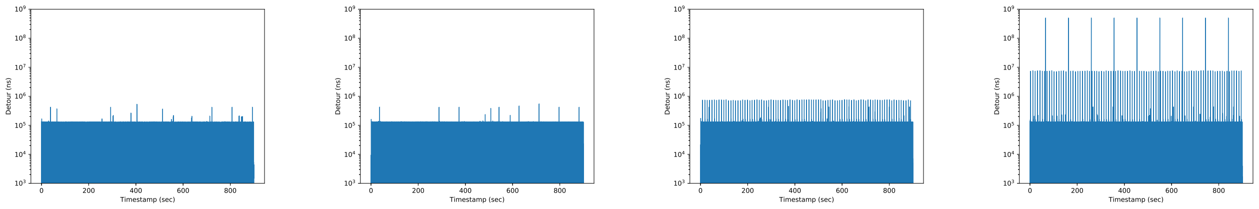
In this section, we examine the performance impact of a single process experiencing CEs. Specifically, we consider how frequently CEs can occur without significantly degrading application performance. Understanding when CEs degrade application perform may be useful to system administrators who are faced with deciding when to replace DIMMs that are experiencing CEs.[8] To understand how CEs on a single process affect application performance, we conducted a series of experiments in which we varied the $MTBCE_{node}$ and simulated how long our applications took to complete for three different logging overheads. The results of these experiments are shown in Fig. 3. Fig. 3a shows the results for correction-only (i.e., no information about the correctable errors is logged). The performance impact in this case is negligible; the simulated application slowdown is below 1% even though CEs are very frequent. In the case of software logging, the simulated application slowdown remains below 10% for an $MTBCE_{node}$ as low as 10 milliseconds. In the case of firmware logging, the simulated application slowdown remains below 10% for an $MTBCE_{node}$ as low as 1 second. Given that the overhead of logging each CE in this case is 133ms, the application struggles make meaningful progress when $MTBCE_{node}$ is much smaller than 1 second. For example, an $MTBCE_{node}$ of 200ms causes the application's runtime to be hundreds of percent slower than error-free execution.

C. Correctable Overheads for Current and Future Systems

In this section, we examine the performance impact of every application process experiencing correctable errors at the same rate on all processes. Specifically, we consider how the $MTBCE_{node}$ affect application performance on current, recent, and projected future systems. A description of each of the systems we considered is shown in Table II.

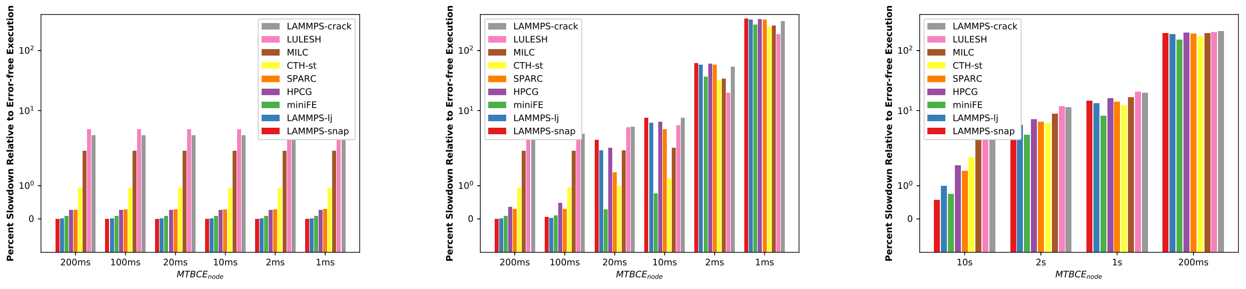
To understand the impact of CEs on current HPC systems, we conducted a set of experiments in which we simulated the performance impact of three values of CE logging overheads on nine workloads. We repeated these experiments for three current (or recent, in the case of Cielo) extreme-scale systems. The $MTBCE_{node}$ for Trinity and Summit are not public. Therefore, we use the CE rate derived from data collected over the lifetime of Cielo [24], [39]. We project the CE rate on Summit and Trinity by assuming that the rate is constant per byte of memory. Because all three systems use chipkill-correct DRAM

[8] This is particularly true given that a recent study from Levy et al. presented data showing no correlation between correctable and uncorrectable errors [24]. Based on this data, system administrators may be able to allow DIMMs to generate CEs without jeopardizing the operational status of the machine.



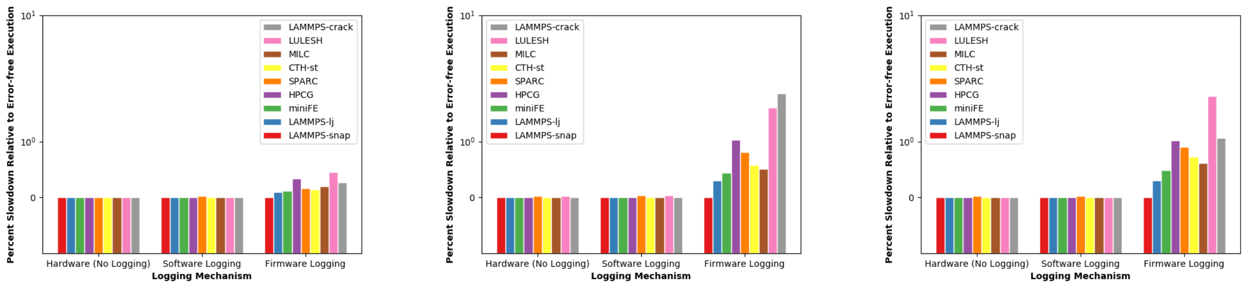
(a) Native OS Signature for Blake (b) “Dry Run” Injection OS Signature (c) Software Cost (OS decoding with CMCI) (d) Firmware Cost (Firmware decoding with EMCA, threshold set to 10)

Fig. 2: Native and “dry run” OS noise signature for Blake. The “dry run” option configures the EINJ interface at the requested frequency (in this case every ten seconds) but does not trigger the error. This attempts to measure the cost of the error injection utility and writing to the `sysfs` filesystem. As can be seen from the figure, the injection utility impacts no additional OS noise. Missing from this figure is the “All logging turned off” case, denoting the overheads of the hardware correction mechanisms. This data was collected and looked the same as the “Native” and “Dry Run” figures.



(a) Hardware Correction Impacts (b) Software-based Logging Impacts (c) Firmware-based Logging Impacts

Fig. 3: **Performance impacts of one process experiencing correctable errors as a function of the recovery overhead. Results are based on 16,384 (16,000 for LULESH) node simulations of each application. Data for three scenarios are shown: hardware only correction with no logging** ($150ns$ **per event), Software-based logging using the Corrected Machine Check Architecture (CMCA)** ($775\mu sec$ **per event), and the Firmware-based logging using the Enhanced Machine Check Architecture (EMCA)** ($133msecs$ **per event)** .



(a) Cielo (b) Trinity (c) Summit

Fig. 4: **Performance impacts of correctable errors for existing systems Cielo, Trinity, and Summit using data from Table II. Data for three scenarios are shown: hardware only correction with no logging** ($150ns$ **per event), Software-based logging using the Corrected Machine Check Architecture (CMCA)** ($775\mu sec$ **per event), and the Firmware-based logging using the Enhanced Machine Check Architecture (EMCA)** ($133msecs$ **per event). These results confirm that the correctable error rate on recent systems do not lead to significant application performance slowdowns.**

ECC, we believe this is a reasonable approximation. Figure 4 shows the results of these experiments. In these figures, the height of each bar represents the arithmetic mean of at least eight simulations (in some cases data from 16 simulations was used). Error bars are not shown in this figure because they would not be visible. The data in this figure show that the impact of CEs on these systems is negligible, significantly less than 10% in all cases. These results confirm that CEs on current systems are not a significant issue.

To understand the potential impact of CEs on future systems, we conducted as series of experiments in which we simulate workload performance for five projected exascale systems. We simulated the execution of nine workloads running on five projected future systems with three values of the CE logging overhead. A description of each of these systems is available in Table II, *see also* Section III-E. For each exascale system, we assume $16,384$ nodes, 700 GiB of DRAM memory. Each system uses a different $MTBCE_{node}$ projection, based on data collected on Cielo. Because the CE rate measured on Cielo may be optimistic for an exascale platform, these experiments allow us to explore how much more frequent CEs can without significantly degrading application performance.

The results of experiments are shown in Figure 5. In all five cases, the performance impact of Hardware correction-only (no logging) is negligible. Similarly, the performance impact of software logging is modest: significantly below 10% in all five cases. In contrast, the performance impact of firmware logging is, in some cases significant. The workload slowdown on the $\text{Exascale}_{\text{CE}_{\text{Cielo}\times 10}}$ system is nearly 100% for LAMMPS-crack and LULESH, and between 10 and 15% for miniFE, HPCG, SPARC, CTH and MILC. Applications running on the $\text{Exascale}_{\text{CE}_{\text{Cielo}\times 100}}$ $\text{Exascale}_{\text{CE}_{\text{median(Facebook)}}}$ systems experience even larger slowdowns: 100–1000% for SPARC, CTH, MILC, LULESH, and LAMMPS-crack. Interestingly, LAMMPS-lj and LAMMPS-snap never see overheads greater than a few percent in all five cases. We believe this variance in application behavior is due to the difference in collective frequency of each application, *see* [19]. The implication of this data is that to achieve good workload performance on future systems, we should limit the $MTBCE_{node}$ so that CEs are no more 10 to 20 times more frequent than they were observed on Cielo. In other words, we should ensure that $MTBCE_{node}$ remains above 5.544–$3,024$ seconds.

D. Exploring Software/OS Reporting Impacts

In the previous section we outlined $MTBCE_{node}$ rates which demonstrate significant impacts for the firmware decoding of correctable errors. In this section, we try to determine at what CE rates do we see overheads for the software/OS decoding of CE.

Figure 6 shows the simulated overheads for three different $MTBCE_{node}$ scenarios: 36 seconds, 3.6 seconds, and around 1 second. Also shown on the figures are the hardware-only and firmware overheads, which can be used for comparison. From this figure we observe that even in the case of *very* frequent CE (once every second, per node), the performance impacts are less then 10%. Therefore, the CE rate can increase by a factor of one million of the measured on Cielo before the software logging has significant performance impacts.

Finally in this section, we note that we could not find a reasonable $MTBCE_{node}$ such that we could observe significant performance impacts for the "no logging" case. Such a rate reaches the limits of our simulation infrastructure and would correspond to a scenario where hardware would be broken to such a degree that replacement is the proper action.

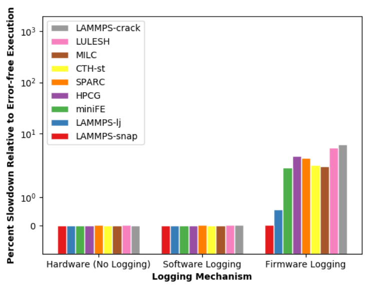
E. Exploring Correctable Reporting Duration Overheads

Lastly in the section, we investigate the impact of correctable error reporting durations on application performance at scale. To illustrate this point will will use two very different $MTBCE_{node}$, of 200 milliseconds and 720 seconds. For each of these rates we vary the per correctable event overheads from 150 nanoseconds to 133 milliseconds. Again we use the simulation framework described previously with a node count of $16,384$ nodes.

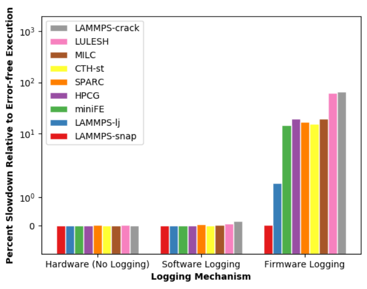
Figure 7 shows the results of these reporting overheads. First thing to note is that no data included for the $MTBCE_{node} = 5.55 \times 10^{-5}$ hours case. Due to the frequency CEs generated on each node, the application is essentially unable to make any reasonable forward progress. The next important point to note is, though there is four orders of magnitude difference in the CE rates, there is only two orders magnitude difference in overheads (in some case the overheads is only one order). Overall, this figure demonstrates two important points: (i) Keeping per-event CE overheads lower is key to keeping overheads low, and (ii) If per-CE event overheads are kept low, a much higher CE rate can be tolerated by the system without significant impacts. These two points are important to the design and construction of future exascale-class systems as less reliable (in terms of CE rate) DRAM hardware may be utilized with possible saving on power and/or cost.

V. Related Work

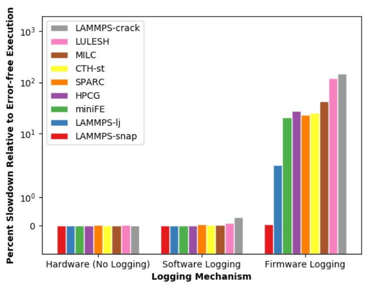
Efforts to characterize the frequency and type of correctable and uncorrectable failures on large-scale HPC and cloud systems have been ongoing for over a decade now. Schroeder and Gibson studied failures in supercomputer systems at LANL [40]. Schroeder *et al.* conducted a large-scale field study using Google's server fleet [36]. Li *et al.* studied memory errors on three different data sets, including a server farm of an Internet service provider [41]. In 2010, Li *et al.* published an expanded study of memory errors at an Internet server farm and other data center sources [20]. Hwang *et al.* published an expanded study on Google's server fleet, as well as two IBM Blue Gene clusters [21]. Sridharan and Liberty presented a study of DRAM failures in a high-performance computing system [42]. El-Sayed *et al.* studied temperature effects of DRAM in data center environments [43]. Similarly, Siddiqua *et al.* studied DRAM failures from client and server systems [44]. Sridharan *et al.* studied DRAM and SRAM faults, with a focus on positional and vendor effects [22]. Di Martino *et al.* studied failures in Blue Waters, an HPC system at the



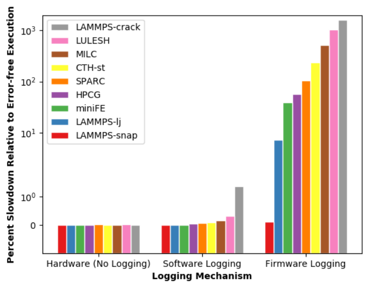
(a) $\text{Exascale}_{\text{CE}_{\text{Cielo}}}$



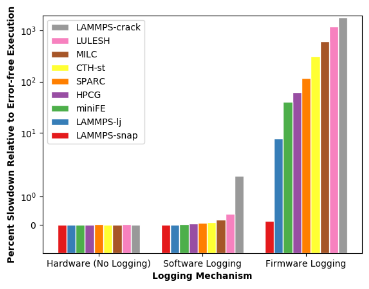
(b) $\text{Exascale}_{\text{CE}_{\text{Cielo}\times 10}}$



(c) $\text{Exascale}_{\text{CE}_{\text{Cielo}\times 20}}$

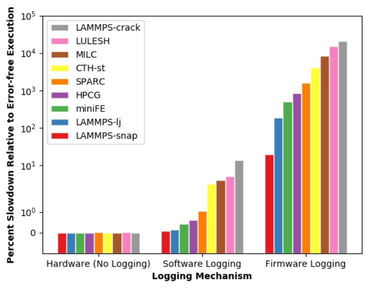


(d) $\text{Exascale}_{\text{CE}_{\text{Cielo}\times 100}}$

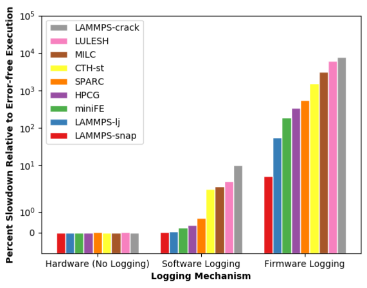


(e) $\text{Exascale}_{\text{CE}_{\text{median(Facebook)}}}$

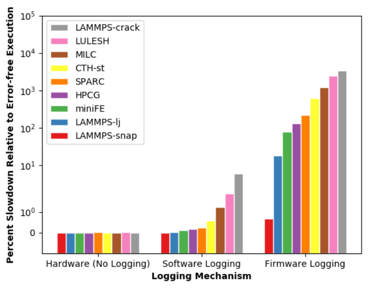
Fig. 5: **Performance impacts of correctable errors for hypothetical Exascale-class systems using the data from Table II. Data for five error rates are shown: the rate measured on Cielo [24], 10 times the rate measured on Cielo (denoted $\text{CE}_{\text{Cielo}\times 10}$), 20 times the rate measured on Cielo (denoted $\text{CE}_{\text{Cielo}\times 20}$), 100 times the rate measured on Cielo (denoted $\text{CE}_{\text{Cielo}\times 100}$), and the rate measured as the median from Meza et al. [2](about 120 times the CE rate measured on Cielo, denoted $\text{CE}_{\text{median(Facebook)}}$). Three scenarios are shown: hardware only correction with no logging ($150ns$ per event), Software-based logging using the Corrected Machine Check Architecture (CMCA) ($775\mu sec$ per event), and the Firmware-based logging using the Enhanced Machine Check Architecture (EMCA) ($133msecs$ per event). To keep correctable overheads low, the *MTBCE_{node}* for an exascale system should not drop below 5.544–$3,024$ seconds**



(a) $\mathit{MTBCE}_{node} = 1$ second

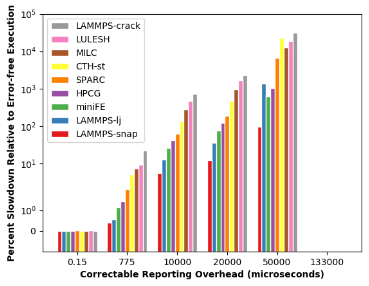


(b) $\mathit{MTBCE}_{node} = 3.6$ seconds

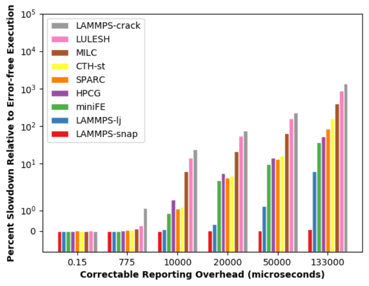


(c) $\mathit{MTBCE}_{node} = 36$ seconds

Fig. 6: **Performance impacts of correctable errors for a hypothetical Exascale-class systems using an extreme MTBCE rate to determine where Software-OS reporting is impacted. Again, three scenarios are shown: hardware only correction with no logging ($150ns$ per event), Software-based logging using the Corrected Machine Check Architecture (CMCA) ($775\mu sec$ per event), and the Firmware-based logging using the Enhanced Machine Check Architecture (EMCA) ($133msecs$ per event)** .



(a) $MTBCE_{node} = 0.2$ seconds



(b) $MTBCE_{node} = 720$ seconds

Fig. 7: **Performance impacts of correctable errors for a hypothetical Exascale-class systems with $MTBCE_{node} = 0.2$ seconds and $MTBCE_{node} = 720$ seconds. A number of different correctable reporting overheads are shown, from $150ns$ per event to $133msecs$. Results demonstrate that if per reporting event durations can be kept low, a higher CE rate (possibly using less power) will achieve acceptable overheads.**

University of Illinois, Urbana-Champaign [45]. Additionally, Bautista-Gomez *et al.* [23] presented a study of DRAM memory errors on a large-scale system in the explicit absence of an ECC in an effort understand the behavior of raw memory failures. Siddiqua *et al.* [39] presented a study demonstrating that the incidence of each DRAM correctable fault *mode* on Cielo was stable over time. Gupta *et al.* [46] studied five vastly different systems of varying sizes and hardware and software configurations to discover common failure trends that are across HPC systems. The data set covering the longest period of operation that they considered was collected on the Jaguar XT4 system from 2008-2011. Finally, Levy *et al.* [24] demonstrated that Cielo exhibited no discernible aging effects and that contrary to popular belief, correctable DRAM faults are not predictive of future uncorrectable DRAM faults.

Our study has origins in previous works that characterizes application behavior in the presence of OS noise [10], [9]. Collectively, this research shows that the pattern of OS noise events determines the impact on application performance and the benefits of coordination. Moreover, it shows that the duration of an OS noise event can significantly slowdown application performance. Additionally, Delgado and Karavanic [47] examined the impact of system management mode (SMM) interrupts on network and IO workloads. Finally, Macarenco *et al.* [48] examine the impact of SMM mode induced by using virtualization for small scale NAS parallel benchmark runs and the UnixBench benchmark.

Most closely related, Gottscho *et al.* [3] examine the single-machine performance impacts of correctable DRAM errors for web search and SPEC CPU2006 benchmarks. In this work, the authors demonstrate that an "avalanche" of these correctable errors can significantly impact benchmark performance. Finally, using a proprietary hardware and software tool, the authors also characterize the impacts of DRAM correctable errors for Windows Server 2012 at a hardware, firmware, and OS level.

Our work distinguishes itself from these existing studies in several important ways. First, to the best of our knowledge this is the first study to examine the HPC performance impacts of *correctable* DRAM errors. Much of the existing work is focused on either characterizing failures or examining the application performance implication of DUEs. In addition, unlike previous work, we attempt a principled analysis of the correctable error rate increases likely with future systems as well as the importance of reducing the duration of each memory error event. Finally, we provide prescriptive guidance on how possible increases in MTBCEs of future exascale systems can influence performance.

VI. Conclusions & Lessons Learned

Efficiently deploying and using the first Exascale system will require a detailed understanding of correctable error overheads. In this paper, we provide a detailed analysis of the impacts of correctable errors on a number of hypothetical Exascale-class systems. Our results demonstrate that: (i) If *Firmware First* CE reporting is used on future systems, the $MTBCE_{node}$ for an exascale system should not drop below $5,544$–$3,024$ seconds (depending on workload) to minimize impacts to less than 10%; (ii) If firmware first functionality is not needed on a proposed exascale system and OS-level reporting of CEs is used, it may be possible utilize significantly less reliable DRAM hardware, for example an $MTBCE_{node}$ greater than 432 seconds, or 120X the CE rate measured on Cielo; and (iii) For single node, bursty CE errors, in the software-based logging case a node can generate a CE every 10ms without significantly impacting application performance. For firmware-based logging, CE that occur more than once every second lead to significant slowdowns. Taken together, the results of this paper provides critical insight and practical advice to users and systems administrators on correctable error rates and overheads.

VII. Acknowledgments

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems, Peter Kogge, editor & study lead," 2008.

[2] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 415–426. [Online]. Available: https://doi.org/10.1109/DSN.2015.57

[3] M. Gottscho, M. Shoaib, S. Govindan, B. Sharma, D. Wang, and P. Gupta, "Measuring the impact of memory errors on application performance," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 51–55, Jan 2017.

[4] Intel, "MCA enhancements in Intel Xeon processors," https://software.intel.com/en-us/download/mca-enhancements-in-intel-xeon-processors, January 2018.

[5] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 4, pp. 410–416, 2009.

[6] K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using mpi-ulfm," in *Proceedings of the 21st European MPI Users' Group Meeting*. ACM, 2014, p. 51.

[7] TOP500, "Highlights — TOP500 supercomputer sites," https://www.top500.org/lists/2019/06/highs/, retrieved September 2019.

[8] Nvidia, "Dynamic page retirement: Reference guide," 2019. [Online]. Available: https://docs.nvidia.com/deploy/pdf/Dynamic_Page_Retirement.pdf

[9] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[10] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 19.

[11] Linux Kernel Organization, "APEI error injection," https://www.kernel.org/doc/Documentation/acpi/apei/einj.txt, retrieved April 2019.

[12] "AMD64 architecture programmer's manual volume 2: System programming, revision 3.23," http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf, 2013.

[13] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro, "Assessment of the effect of memory page retirement on system RAS against hardware faults," in *International Conference on Dependable Systems and Networks (DSN'06)*, June 2006, pp. 365–370.

[14] Intel, "Intel 64 and IA-32 architectures software developer's manual," http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, January 2019.

[15] G. Bronevetsky, "Communication-sensitive static dataflow for parallel message passing applications," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2009, pp. 1–12.

[16] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim - simulating large-scale applications in the LogGOPS model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, Jun. 2010, pp. 597–604.

[17] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, T. Hoefler, and P. Widener, "Using simulation to evaluate the performance of resilience strategies at scale," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2013 SC Companion:*. IEEE, 2013.

[18] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '93. New York, NY, USA: ACM, 1993, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/155332.155333

[19] K. B. Ferreira, S. Levy, P. Widener, D. Arnold, and T. Hoefler, "Understanding the effects of communication and coordination on checkpointing at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*. IEEE Press, 2014, pp. 883–894.

[20] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, Calif., USA: USENIX Association, 2010, pp. 6–20. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855846

[21] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," in *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150989

[22] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 22:1–22:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503257

[23] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of DRAM raw error rate on a supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 55:1–55:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014978

[24] S. Levy, K. B. Ferreira, N. DeBardeleben, T. Siddiqua, V. Sridharan, and E. Baseman, "Lessons learned from memory errors observed over the lifetime of Cielo," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 43:1–43:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3291656.3291714

[25] K. B. Ferreira, S. Levy, K. Pedretti, and R. E. Grant, "Characterizing mpi matching via trace-based simulation," *Parallel Computing*, vol. 77, pp. 57 – 83, 2018.

[26] S. Levy, B. Topp, K. B. Ferreira, D. Arnold, T. Hoefler, and P. Widener, "Using simulation to evaluate the performance of resilience strategies at scale," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Springer International Publishing, 2014, pp. 91–114.

[27] S. Plimpton, "Fast parallel algorithms for short-range molecular-dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[28] Sandia National Laboratories, "LAMMPS molecular dynamics simulator," http://lammps.sandia.gov, Apr. 10 2013.

[29] Lawrence Livermore National Laboratory, "Co-design at Lawrence Livermore National Lab : Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)," http://codesign.llnl.gov/lulesh.php.

[30] Indiana University, "HPCG benchmark," http://physics.indiana.edu/~sg/milc.html, retrieved September 2017.

[31] J. McGlaun, S. Thompson, and M. Elrick, "CTH: A three-dimensional shock wave physics code," *International Journal of Impact Engineering*, vol. 10, no. 1, pp. 351–360, 1990.

[32] J. E. S. Hertel, R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. PetneY, S. A. Silling, P. A. Taylor, and L. Yarrington, "CTH: A software family for multi-dimensional shock physics analysis," in *Proceedings of the 19th Intl. Symp. on Shock Waves*, Jul. 1993, pp. 377–382.

[33] Sandia National Laboratories and University of Tennessee Knoxville, "MIMD lattice computation (MILC) collaboration," http://www.hpcg-benchmark.org, 2017, retrieved September 2017.

[34] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and

R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratory, Tech. Rep. SAND2009-5574, 2009.

[35] M. Howard, A. Bradley, S. W. Bova, J. Overfelt, R. Wagnild, D. Dinzl, M. Hoemmen, and A. Klinvex, "Towards performance portability in a compressible CFD code," in *Proc. 23rd AIAA Computational Fluid Dynamics Conference*, 2017.

[36] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: a large-scale field study," *Commun. ACM*, vol. 54, no. 2, pp. 100–107, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1897816.1897844

[37] Los Alamos National Laboratory, "Trinity: Advanced technology system," https://www.lanl.gov/projects/trinity/index.php, Apr. 10 2018.

[38] Oak Ridge National Laboratory, "Introducing Summit," https://www.olcf.ornl.gov/summit/, Apr. 10 2018.

[39] T. Siddiqua, V. Sridharan, S. E. Raasch, N. DeBardeleben, K. B. Ferreira, S. Levy, E. Baseman, and Q. Guan, "Lifetime memory reliability data from the field," in *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct 2017, pp. 1–6.

[40] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in *Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, June 2006.

[41] X. Li, K. Shen, M. C. Huang, and L. Chu, "A memory soft error measurement on production systems," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ser. ATC'07. Berkeley, Calif., USA: USENIX Association, 2007, pp. 21:1–21:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364385.1364406

[42] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, Calif., USA: IEEE Computer Society Press, 2012, pp. 76:1–76:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389100

[43] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder, "Temperature management in data centers: why some (might) like it hot," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '12. New York, NY, USA: ACM, 2012, pp. 163–174. [Online]. Available: http://doi.acm.org/10.1145/2254756.2254778

[44] T. Siddiqua, A. Papathanasiou, A. Biswas, and S. Gurumurthi, "Analysis of memory errors from large-scale field data collection," in *Silicon Errors in Logic - System Effects (SELSE), 2013 IEEE Workshop on*, 2013.

[45] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of Blue Waters," in *International Conference on Dependable Systems and Networks*, 2014.

[46] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 44:1–44:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126937

[47] B. Delgado and K. L. Karavanic, "Performance implications of system management mode," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sept 2013, pp. 163–173.

[48] K. Macarenco, K. Frye, B. Hamlin, and K. L. Karavanic, "The effects of system management interrupts on multithreaded, hyper-threaded, and MPI applications," in *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, Aug 2016, pp. 338–345.