Application Health Monitoring for Extreme-scale Resiliency using Cooperative Fault Management

Pratul K. Agarwal,[1,2,] Thomas Naughton,[1] Byung H. Park,[1] David E. Bernholdt,[1] Joshua J. Hursey,[1,3] Al Geist[1]*

[1]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA, [2]Biochemistry and Cellular and Molecular Biology, University of Tennessee, Knoxville, Tennessee, USA, [3]Present address: IBM Systems, IBM, Rochester, Minnesota, USA

Corresponding Author: pratul@agarwal-lab.org; +1-865-765-7750.

Keywords

Fault-tolerance, exascale resiliency, silent errors, heterogeneous systems, molecular dynamics, quantum chemistry calculations.

ABSTRACT

Resiliency is and will be a critical factor in determining scientific productivity on current and *exascale* supercomputers, and beyond. Applications oblivious to and incapable of handling transient soft and hard errors could waste supercomputing resources or, worse, yield misleading scientific insights. We introduce a novel application-driven silent error detection and recovery strategy based on application health monitoring. Our methodology uses application output that follows known patterns as indicators of an application's health, and knowledge that violation of these patterns could be indication of faults. Information from system monitors that report hardware and software health status is used to corroborate faults. Collectively, this information is used by a fault coordinator agent to take preventive and corrective measures by applying computational steering to an application between checkpoints. This cooperative fault management system uses the Fault Tolerance Backplane as a communication channel. The benefits of this framework are demonstrated with two real application case studies, molecular dynamics and quantum chemistry simulations, on scalable clusters with simulated memory and I/O corruptions. The developed approach is general and can be easily applied to other applications.

1. INTRODUCTION

Resiliency, in addition to concurrency and energy efficiency, is and will be a major challenge for future high-performance computing (HPC) architectures [1]. Preliminary estimates suggest mean time to interrupt (MTTI) could be from a few hours to a day as the concurrency on future systems increases rapidly [2, 3]. Without additional effort, applications tend to be susceptible to this reduction in MTTI, resulting in potentially substantial losses in productivity for end users. A wide range of research efforts are under way to enable fault prediction, fault detection, preparation for failure and recovery [4-17]. However, the more intrusive such strategies become in application code bases, the greater the resistance to their adoption, especially for the large body of existing applications. In our experience, application-level checkpoint/restart is the most commonly employed fault tolerance technique at present, and a recent survey of exascale application teams suggests that it will continue to be for at least the next generation of large-scale supercomputers [18], despite dire predictions for their MTTIs.

Historically, resilience concerns for application running on HPC systems have centered around "fail-stop" situations, in which faults leads to failures of hardware or software components, and often, interruption of user applications. However, increasingly, evidence is demonstrating the data corruption issues are also significant [19-23]. Data corruption, in the memories of host or accelerations, processor caches and registers, as well as storage systems based on spinning disk or non-volatile memory, may be detected and converted to a fail-stop error, but may also be undetected, or *silent.* Undetected data corruption is one of the most insidious forms of error in computational science because of the potential for erroneous results. Further, the hardware techniques used to detect (and sometimes correct) data corruption tend to be expensive in both additional hardware and power. Levy et al. [24] note that while advanced

error correcting codes (e.g., chipkill-correct) are useful, they require more power that may limit their viability in future systems where power is an increasing limitation in the design. The problem is further exacerbated by the fact that progress in simulation-based science, combined with the increasing capabilities of HPC systems, are resulting in rapidly increasing data generation and consumption, and therefore more opportunities for data corruption to impact simulations.

As a practical matter, undetected data corruptions can be very challenging for end-users to deal with. In cases where they lead to abnormal or early termination of the application, there are often few options for debugging or tracing the root cause of the failure. Where they are not fatal, there may be delayed detection of numerical inconsistencies and possibly, incorrect scientific results. Many HPC systems operate in a batch execution mode, which may make it harder for users to monitor their jobs for inconsistencies as they run. Anecdotally, the most common response to an unexplained job failure or numerical inconsistency is to simply re-run the job, which of course further reduces user productivity as well as the resource utilization efficiency of the system [25].

To address these issues, we hypothesize that application-aware, and, in fact application-driven, strategies for failure detection, preparedness, and recovery will provide reasonable control over productivity. In this paper, we describe an approach that utilizes application "health" monitoring, which can be paired with a system-level framework for fault information that can be used to corroborate deviations from expected behavior observed at the application level for better decision-making. The approach can be implemented mostly as a small, standalone framework which is customized to the target application with plugins.

The concept of application "health" is schematically illustrated in Figure 1. Many applications already compute (or could compute) quantities that are indicative of the health of the simulation. Deviations or unexpected behavior, as defined by the user, may be indicators of errors occurring in the system. Such occurrences can trigger user-defined actions to help diagnose and/or respond to the fault. Responses could include, for example, rolling back to a pre-fault checkpoint and resuming execution. We illustrate the potential for this approach with molecular dynamics (MD) and quantum chemistry simulations [26].

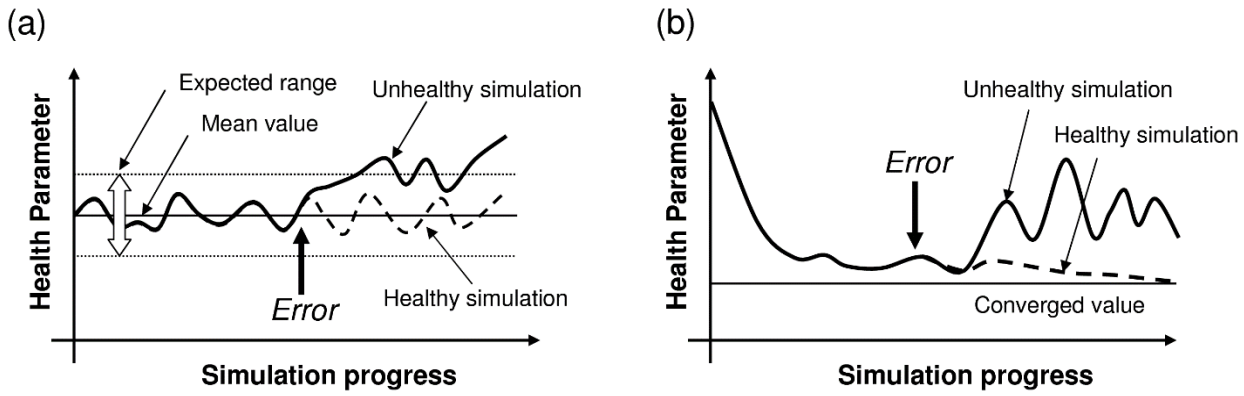

Figure 1: Fault detection mechanism based on application health monitoring based on two alternative scenarios. (a) A case where health parameters (such as simulation temperature) stay within a range. (b) A case where health parameters approach a converged value (such as minimized system energy).

We believe this concept of application health monitoring can be generalized to other applications and domains and therefore provide a novel mechanism for application error detection and recovery. The advantages of the proposed approach include minimal or no code modifications to the application; immediate information on the impact of a fault on the application; and mechanisms for application recovery. In particular, the unique contributions of the developed approach are

1. Development of a fault detection and recovery strategy based on application health monitoring;
2. Introduction of a framework to implement cooperative application fault management; and

3. Demonstration of the proposed infrastructure and its benefits with two production-level applications to test cases using early detection and failure recovery strategies.

2. METHODOLOGY AND FRAMEWORK

Our approach is based on monitoring application progress using a set of quantities that are expected to conform to a set of rules predefined by the user. If those rules are violated, warnings are triggered, and a cooperative fault management system is then consulted to corroborate the failure and locate its possible source. If the failure is confirmed, then the recovery process can be initiated, based on user-defined actions. In this section, we describe the detailed methodology and the developed framework.

Application health monitoring: Many applications monitor calculated quantities that are expected to follow predictable patterns in a scientific simulation. Such quantities typically are inspected by users as indicators of the *health* of the simulation. The use of a quantity to indicate application health depends on the following properties: (1) It should follow expected patterns. (2) No (or few) resources should be required to obtain this quantity while the application is running. (3) There should be no (or very little) delay in the simulation state to which that the quantity corresponds. (4) It should be possible to define rules that indicate that the quantity is not behaving as expected for normal application runs. For example, production-stage MD simulations are often run as constant-temperature ensembles with a specific target temperature, such as 300 kelvin (K), or as constant-energy ensembles. While the temperature (or energy) does fluctuate for numerical and algorithmic reasons, any significant deviation or drift from the target value indicates a bad simulation state, due to either a software problem (inappropriate

configuration of the simulation or a bug in the application) or a fault. In another type of simulation (such as a quantum chemistry calculation) in which energy is minimized through an iterative process at each step, any abnormal increase in the system energy or the number of iterations at each minimization step can be used to trigger a warning of an unhealthy simulation state.

Error detection and recovery: We propose that simulation health can be measured by monitoring suitable parameters such as simulation temperature (or, e.g., total energy or simulation volume). In Figure 1(a), the black dashed curve denotes a healthy trajectory, and an unhealthy state (solid black) can be detected when the temperature violates a set of ranges, which could be defined as a set of *rules*. Similarly, as shown in Figure 1(b), an unexpected large increase in system energy could trigger a warning for the type of energy minimization simulation commonly used for quantum chemistry calculations. Note that results from healthy runs are not needed, as long as the user has enough knowledge of the simulation domain to be able to define appropriate rules to identify suspicious changes in the system state. Rules may be based on instantaneous behavior or may utilize historical values (for example a moving average). In principle, an application could be modified to incorporate the monitoring and response mechanisms directly; however, if the mechanisms operate external to the application, there is much greater potential for reuse of the resilience infrastructure. While the specific quantities of interest, the rules that identify unhealthy simulations, and the desired responses are specific to an application, the overall process of monitoring quantities of interest against the rule set, corroborating possible faults, and carrying out the user-defined action in response is quite general.

External monitors (run concurrently with the application) that are aware of the history of the quantities of interest offer another major advantage over internal monitors: individual time-steps in applications (such as MD simulations) may show only small deviations in these quantities; however, over time, a large deviation (drift) may be a sign of failure. Therefore, we hypothesize that an external agent that can keep a history of desired health parameters could provide a vital strategy for failure detection and recovery.

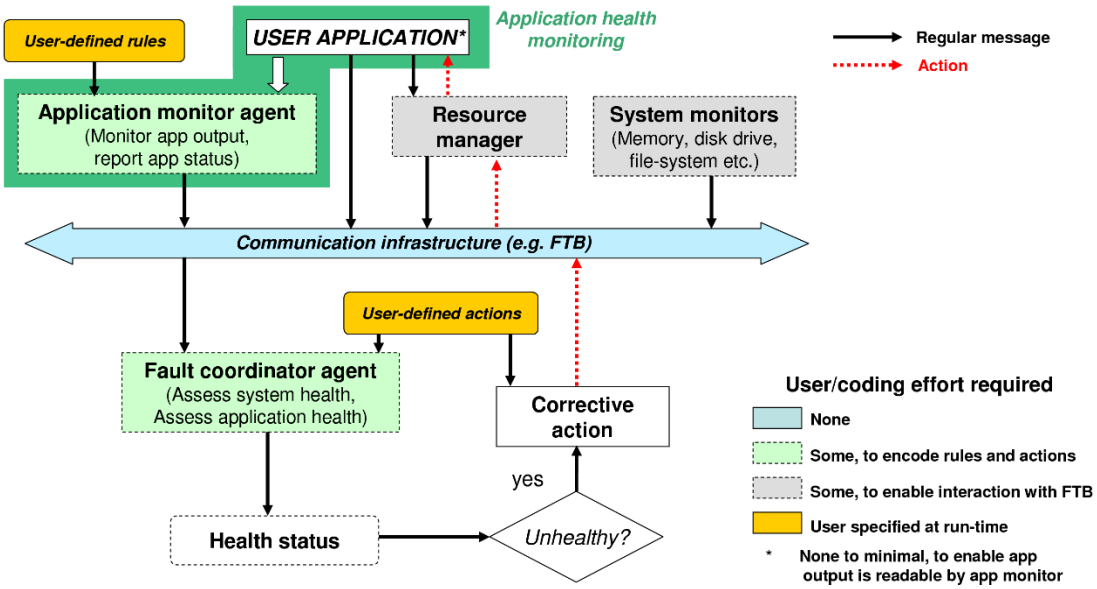

Figure 2: Cooperative fault management system.

Framework description: Our framework consists of the application monitoring agent, the fault coordinator agent, system monitors, the communication infrastructure, and other essential components. Figure 2 illustrates a prototype implementation of the proposed framework. An important component of the framework is the *application monitor agent*. The features of the application monitor agent include the ability to (1) read in a set of user-defined rules based on a health parameter, (2) monitor the simulation output (during the progress of the simulation) for the defined health parameter, (3) run an evaluation engine on the simulation output, (4) issue a warning or error message when the user-defined rules are violated, and (5) maintain a history of

the simulation health parameter to aid the recovery process by determining a healthy checkpoint before the error began impacting the simulation. Note that a regular expression describing how to extract the health parameter information from the simulation output is also an input to the health agent.

The application monitor agent is fundamentally independent of the application. The rules and the application output to be monitored (defined as input to the agent by the user) are application-specific; however, the occurrence and functioning of the agent are independent from the application. As discussed earlier, the application monitor agent could be closely integrated with (or even coded into) the applications. However, this approach would make an agent too specific to an application and limit its reusability. Further, the generic and decoupled nature of this agent allows close integration into the communication infrastructure, such as the Fault Tolerance Backplane (FTB) from the Coordinated Infrastructure for Fault Tolerant Systems (CIFTS) project [4]. CIFTS is based on a holistic approach to dealing with failures and provides a framework, through FTB, that enables any component using an agreed API, whether software or hardware, to share events to facilitate coordinated identification of and response to faults within the system. FTB can connect a diverse range of components, such as operating systems, job schedulers, mathematics libraries, parallel runtime libraries, file systems, and user applications.

In addition to monitoring the application, additional monitors may be deployed to track the hardware and system software environments. System monitors may take various forms, using information from the computer's reliability, availability and serviceability (RAS) subsystem, such as an Intelligent Platform Management Interface, system logs, or other information sources. Information about faults, errors, and other anomalous events can be used to assist in diagnosing

and responding to anomalies in the application health parameters. Warnings from both the application and system monitors are fed into the fault coordinator agent as they occur. The fault coordinator provides the cooperative aspect of the system by checking for correlations between anomalous events in the application and the system. Heuristically, if a system event is followed closely by an application event, a causal connection is assumed, allowing the user to be more confident in formulating responses to application anomalies. Without such corroboration, the application anomaly might be due to a programming error in the application itself, bad initial conditions, or bad configuration, or to system errors that occurred during execution but went undetected. System anomalies that occur without triggering application anomalies can also be handled, if desired.

The fault coordinator agent selects a corrective action for an application in response to the events observed by consulting rules registered by the application. These rules, provided by the user (or site policies), define preferable actions depending on the fault situation. The role of the system and application monitors is to garner all information that may provide clues to enable the fault coordinator to evaluate the application integrity and possible causes and take appropriate actions. The corrective actions could have implications for system-wide resources (for example, an application requesting immediate checkpointing could impact file-system and other applications); therefore, the process requires coordination and policy management. In the past, we have described efforts to enable such coordinated policy management [27].

Regarding the implementation details of the framework, the FTB from the CIFTS project was selected as the fault communication infrastructure in this study. The FTB is a prototype for exchanging fault events and provides a mechanism for transparent event exchange. Further

details of the FTB implementation used in the proposed management system are available in ref. [27].

Types of failures detected: The proposed framework will allow a mechanism for detection of *silent errors*, defined as errors that are not fatal to the application but lead to corruption of the application state and data. We do not claim that the proposed framework is an exhaustive mechanism for detection of all silent errors; rather, it detects only those errors that alter the health of a simulation to trigger a warning when the user-defined rules are violated. The types of errors that could lead to such silent corruptions in an application include (1) hardware/software memory corruption on nodes; (2) file-system or data corruption during I/O; and (3) in graphical processing unit (GPU) –enabled simulations, errors induced when the error-correcting code (ECC) memory is turned off to improve memory bandwidth and capacity on the device. The approach may also detect some kinds of software bugs which push the simulation into unhealthy states, and potentially certain hardware issues, such as the Pentium FDIV bug [28]. Note that our assumption is that errors that are fatal to an application, such as a node going down or failure of a communication link, would be detected by other mechanisms.

Overhead and scalability considerations: The resource requirements for the application monitor and the fault coordinator agents are modest. They can run on a front/login node or any other node with access to application output and the FTB. Therefore, the overall impact of the proposed infrastructure on HPC systems depends on the underlying scalability of the application code as well as the scalability of the failure communication framework (such as the FTB). Moreover, the framework is event driven, so communication, processing, and memory overheads

are generated only in the event of an unexpected behavior. MD and quantum chemistry codes have been demonstrated to scale to thousands and tens of thousands of cores on current HPC machines for quite some time [29, 30]. The scalability of the FTB has also been described [4]. Consequently, the framework should not lead to any significant overheads in terms of memory and communication requirements; therefore, the given approach can be targeted for *exascale* deployment.

Applicability to other applications: The motivation of the development of the health monitoring strategy and framework is to allow wider applicability to other applications and domains. As discussed earlier, the use of an external agent to monitor the health parameters outside a simulation will require no changes or minimal changes (if the output must be formatted to conform for easy monitoring) to the application code base. A brief survey of other applications indicates that it should be possible to define suitable health parameters in many cases. The essential requirement is that it be possible to define computationally observable quantities that behave predictably in a healthy simulation and that could be expected to be sensitive to the introduction of random error into the simulation data. For example, simulations in biophysics, chemistry, and materials science that use energy minimization and constant- energy/temperature simulations could use health parameters similar to those described for MD simulations and the quantum chemistry test cases discussed in the following section. Other areas, such as computational fluid dynamics, might make use of conservation laws (i.e., conservation of mass or momentum) or drift in statistical quantities characterizing steady-state flows. Note that it is not necessary that values be constant to be useful for monitoring. Quantities with predictable trends, such as monotonic increases or decreases, can also be used.

Techniques for detecting (and in some cases correcting) silent errors have been studied for a variety of iterative solvers [31-35], adaptive numerical integrators [36-38], and other widely used computational approaches [39-41]. Clearly, not all application outputs would be useful as health indicators; but our experience, informed by discussions with many colleagues, is that most computational scientists develop techniques to check their simulations to determine whether the results are believable. Many such checks could be automated and applied throughout a simulation using our approach. In cases where none of the reported information in the application output is suitable for use as a health parameter, memory access, I/O, or communication patterns could be used as indicators of simulation health, as well as timing, as in the AutomaDeD project [14].

3. RESULTS

In this section, we describe the use of the proposed application health monitoring strategy and cooperative fault management for two application test cases with simulated errors in the memory and I/O systems, respectively.

Application 1: Molecular dynamics with memory corruption

MD is a computational technique for simulating the behavior of a system of particles over time by integration of Newton's equations of motion. Atomistic MD simulations are computationally intensive. Each step involves billions of arithmetic operations that model the interactions among thousands of particles, which additionally must be repeated millions or more times. The typical implementation spreads the computation over cores and uses a communication layer (such as MPI) to aggregate the intermediate results from all cores, update the simulation state, and redistribute new computations. This process implies that the processes must be synchronized

after each time-step and the computations delegated to each processor core are essential for the entire simulation.

MD simulation runs last for hours (or longer) and small errors (associated with even one processor core) will likely alter the future trajectory of the simulation. Therefore, early detection of silent errors is crucial to minimize the impact on end-user productivity. For the purposes of this study, we chose the MD engine LAMMPS (Large-scale Atomic Molecular Massively Parallel Simulator) [42, 43] because of its performance characteristics and wide user base. LAMMPS contains routines applicable to performing simulations in biology, chemistry, and materials science. The massively parallel scalability of LAMMPS, even up to billion-atom systems on tens of thousands of CPU-cores, is well documented [29]. No code modifications of LAMMPS were required for this study. LAMMPS was an early target for porting to heterogeneous architectures and has been successfully ported on systems accelerated with GPUs and field-programmable gate arrays [44, 45]. The checkpointing (or application restarting) information consists of the atomic coordinates and velocities that are written to disk periodically, based on the number of time-steps. Typically, this restart information ranges from 1 to 200 megabytes (MB) depending on the number of particles in the system.

Without health monitoring: Currently, simulations that use no health monitoring strategy but encounter silent errors are difficult to diagnose. The fault (either hard or soft) may be fatal, leading to termination of the application. In the case of silent errors, there is no explicit mechanism to validate the accuracy of the simulation. One exception is the use of internal checks to ensure that the atomic forces do not exceed thresholds, which are typically set to very large values, to detect bad simulation states. Normally, in the post-production period, if the user

suspects unexpected behavior, the typical response is to re-run the simulation and see if the cause was silent error (in which case, the two outputs are different) or a bad simulation state (the two outputs are the same or similar).

With health monitoring: Simulation temperature (T) is commonly analyzed by end-users as an indicator of simulation health. In simulation ensembles with constant temperature (or constant energy), the quantity is routinely computed and printed in the output information. Therefore, our application monitoring agent is configured to monitor T in the simulation output and evaluate a number of user-defined rules. Table 1 and Figure 3 describe a set of rules to monitor the health of an application based on T. Note that in Figure 3, the black trajectory is the healthy trajectory, which is not known in advance. The definition of these rules does not require knowledge of the healthy trajectory. Further, a significant advantage of this approach is that, based on the defined rules to which these faults are traced back, the simulation can be rolled back to a healthy checkpoint for an application recovery process. As shown in Figure 3, careful selection of rules will allow tracing in the history where the deviant behavior started. The checkpoints (labeled as CP_X) before this point in history will be useful for application restart.

Table 1: Description of temperature-based rules for MD

	Description	Notes
Rule 1:	$T > T_{high}$ *or* $T < T_{low}$	Temperature exceeds hard limit of T_{high} or falls below T_{low}
Rule 2:	$abs(T_n - T_{(n-1)}) > \Delta_1$	Temperature difference since last step exceeds Δ_1
Rule 3:	$abs(T_n - T_{(n-d)}) > \Delta_2;$ $d < M$	Temperature difference in current step and any time during the last M steps exceeds Δ_2

The fault management system was tested using LAMMPS and the standard *rhodo* benchmark (32,000 atom system based on the protein rhodopsin in explicitly represented waters). Note this is a real benchmark that is distributed with the LAMMPS source code. The test was conducted under constant-energy conditions (also referred to as an NVE ensemble), and no temperature control was applied. The test, a widely used benchmark for biological simulations with LAMMPS, was run for 1,000 MD steps taking about 300 seconds to complete. The output from LAMMPS (log.lammps) was monitored for simulation temperature (*grep –i temp*). The stream of extracted temperatures was monitored by the application monitoring agent. A set of three representative rules (see Table 1) were read by the monitoring agent at the start.

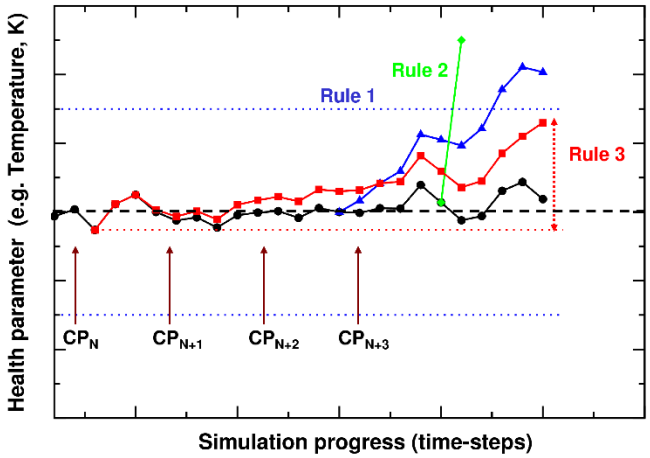

Figure 3: Sample rules for application health monitoring.

Fault injection: Our evaluation included a fault injection mechanism that introduced bit-flips at runtime into the dynamically allocated *heap* memory of a single MPI rank of a LAMMPS simulation. The injection technique employed an entirely user-space mechanism based on the *ptrace()* system call. On invocation, a random address was selected from the *heap* and an error was injected into the target address (data) by randomly flipping one or more bits. The following is an approximation of the sequence of steps:

1. Start the FTB fault injection event logger.

2. Start LAMMPS using *mpirun* on all available cores.
3. Sleep for N (=30) seconds for LAMMPS to start up.
4. Start fault injector on a single node, for a single process, looping until process exits or a threshold is exceeded. Every M (=10) seconds, the injector selects a random address from the *heap* and injects an error (bit-flip) into the target address (data).
5. Stop the FTB fault injection event logger.
6. Archive fault injection log, LAMMPS input and output data, and metadata for the experiment run.

The number of faults injected during a single LAMMPS run varied between 20 and 30. Approximately 50% of 67 test runs resulted in hard failures. However, our focus was on identification of potential silent data corruption, so we ignored these hard failures and concentrated on the cases in which LAMMPS exited normally (i.e., no backtrace or program-detected error in the LAMMPS output file). Further, 90% of the trajectories that ran to completion showed almost no numerical variation, possibly indicating the injected memory fault did not lead to corruption of data essential to the simulation, or introduced a small enough error that it was naturally corrected as the simulation progressed.

About 10% of the trajectories that eventually ran to completion (5% of overall runs) triggered a warning due to of violation of health rules. Figure 4 shows two alternate cases: (a) the fault injection was a single bit-flip and (b) two bit-flips. As a result of fault injection, the first case triggered a warning due to violation of Rule 2 (with Δ_1 = 4 K), and the second case showed violation of Rule 1 (with T_{high} = 304 K). The analysis and detailed statistics regarding the impact of the different numbers of bit-flips and the time required to observe a deviation in the healthy behavior will be the topic of future studies. This information will be vital for defining the

sensitivity of the rules required to capture a fault quickly without leading to an increased frequency of false positives.

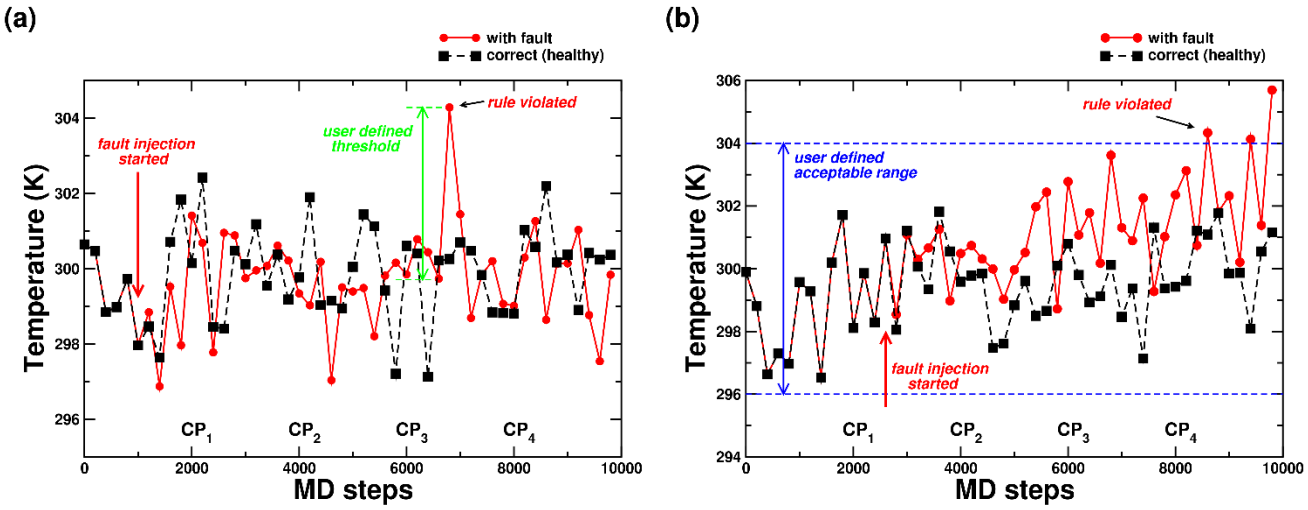

Figure 4: Detection of memory-related fault by MD health monitoring in two alternative scenarios. (a) Simulation with single bit fault injection. (b) Simulation with 2-bit fault injection.

Error corroboration and recovery: Once the application monitor agent issues a warning, the fault coordinator (see Figure 2) looks for corroboration from other independent sources. This test case emulated silent errors in the application that occurred due to corruption in the host (main) memory or the non-ECC memory of the GPU devices. Corroboration will come from a system monitor agent that publishes information regarding memory errors detected at the hardware level or other software layers. Many modern hardware platforms provide basic mechanisms to detect and report memory errors (e.g., bit flips). For example, on x86 platforms, the machine-check architecture (MCA) and machine-check exception (MCE) provide a basis for detecting and reporting errors (e.g., ECC errors, parity errors, cache errors) [46]. These exceptions are typically managed by the operating system and logged in the system logs. There are some user-space utilities that can be used to process logs of MCE events and decode the exception data, i.e.,

determine whether the event was a single- or double-bit error. For example, the *mcelog* utility [47] can be used to aid in reporting errors. It can also assist in management techniques like memory *page soft off-lining*, where a user-space daemon tracks errors and, if thresholds are exceeded, leverages a kernel mechanism to migrate data from the problem page to a new page [47]. FTB-enabled monitors that analyze the system logs (e.g., syslog) to extract these types of events can be developed. Recently, the use of accelerators GPUs has improved the throughput of simulations. Features of GPUs such as ECC memory provide improved accuracy at the expense of reduced memory bandwidth and capacity. Therefore, some applications may prefer to disable ECC during production runs. In such cases, our health monitoring agent could also provide a mechanism for detecting deviant behavior due to memory errors on the GPU. The hardware query/monitoring agents can provide a mechanism for validating the errors. For example, NVIDIA's *System Management Interface* program provides information about errors when ECC is turned on.

Once the event is corroborated, then the fault coordinator can initiate a recovery process, such as rolling back to a checkpoint determined as healthy. Note that the fault coordinator needs a list of actions generated by the user, such as rollback and restart (excluding or replacing failed resources), checkpoint and terminate, or terminate immediately (a graceful failure). In this test case, the checkpoint CP_3 for the single-bit flip case (immediately before the threshold was crossed) and CP_2 for the 2-bit flip case (the checkpoint before the 10-point moving average of T was greater than $(T_{high}-T_{avg})/2$) were automatically selected for restarts.

Implementation on HPC architectures: This test was conducted on a Linux cluster. The cluster provided us the ability to experiment and allowed the fault injection mechanism for memory

corruptions. However, the individual components of the prototype framework are also available for HPC architectures, such Oak Ridge National Laboratory's IBM AC922 supercomputer (Summit). Unfortunately, the batch submission process on these HPC systems does not allow the flexibility to inject faults for this test. Nonetheless, the overall cooperative fault management system described here is fully applicable to current HPC and future extreme-scale systems.

Application 2: Energy minimization with I/O errors

The motivation for this test case is to simulate faults in the I/O subsystem, and errors associated with file-system/storage are explored based on checkpointing and restarting a simulation. Quantum chemistry (electronic structure) calculations are routinely used to investigate the properties of various chemical systems based on quantum mechanical models. Common calculations consist of energy minimization, where the system energy follows predictable patterns of gradual decrease toward a converged value. We used the popular ab initio package GAMESS-US (The General Atomic and Molecular Electronic Structure System) [48, 49], which is available on many large-scale HPC machines. No code modifications to GAMESS were required for this study. Note that our strategy is equally valid for other ab initio quantum chemistry packages, as long as the simulation output containing the appropriate quantity (energy) is available to the health monitoring agent.

The quantum chemistry calculations write information (such as two electron integrals) to disk, which is read in later during the simulation. Further, these calculations typically run for much longer durations than the wall time permitted by batch queues; therefore, restart files are commonly used for long runs. Our test case consists of a GAMESS calculation, which is started periodically in the simulation from the information written to disk by the previous calculation. In this test case, we simulate an I/O fault by corrupting a single byte in the restart file.

Without health monitoring: No specific mechanism is currently available to check for silent errors directly. However, a number of internal integrity checks ensure that the simulation states obey the underlying equations (based on the defining physical laws). In a manner similar to MD runs, if a user suspects unexpected behavior, the typical course of action is to re-run the simulation and see if the behavior is caused by silent error (the two outputs are different) or a bad simulation state (the two outputs are same or similar).

With health monitoring: As shown in Figure 1 (b), we used deviations from the predictable pattern of gradual decrease in system energy as a health parameter. In addition, we used the number of self-consistent field (SCF) iterations at each step as another health parameter. Table 2 describes two rules used to investigate the health of these simulations. GAMESS simulations in this test case were used to minimize energy for a 195-atom system using STO-3G basis set. This test was performed on a Linux cluster with a total of 128 cores (16 nodes with 8 CPU-cores each) and took approximately 20 minutes to complete. To represent a long run involving multiple restarts, we restarted the simulation every five minimization steps, using the atomic coordinates and orbital guess from the previous run. In addition to the normal simulation, two additional runs were performed with fault injections.

Table 2: Rules used for monitoring quantum chemistry runs

	Description	Notes
Rule 1:	$\Delta E_N > f_e * \Delta E_{N-1}$ $f_e = 2$	The energy change from the previous step is greater by a factor *(f)* since the preceding steps. We use $f = 2$.
Rule 2:	$I_N > I_{<S>} + \delta$ $S = 5, \delta = 1$	The number of SCF iterations are greater that average of last *S* steps and a grace (δ) of 1, we use $S = 5$.

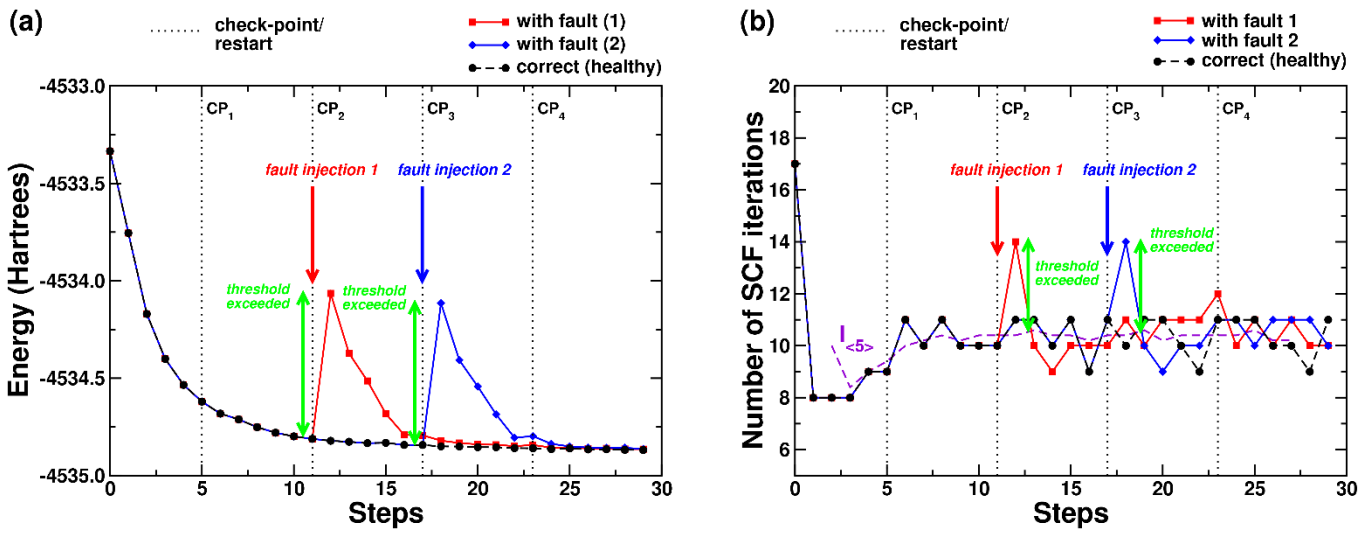

Figure 5: Detection of error in I/O system based on application health monitoring in quantum chemistry calculations. (a) Monitoring system energy convergence. (b) Monitoring the number of SCF iterations.

Fault injection: The fault injection mechanism involved modifying the restart files by changing one digit in one atomic coordinate (that is, 1-byte corruption in a ~5.1 MB file). See Table 3 for fault injection details. The results for this test case are shown in Figure 5, where red and blue curves (representing two different runs with fault injection) immediately show violations of Rules 1 and 2 and therefore the health monitoring agent triggers a warning.

Table 3: Difference in original and corrupted restart files

	Size (bytes)	Fault injection 1 (red)	Fault injection 2 (blue)
Original	5,148,693	22.**9**65622	22.**9**69007
Corrupted	5,148,693	22.**0**65622	22.**0**69007

Error corroboration and recovery: A system monitoring agent associated with the file system (or even system logs) could be used by the fault coordinator to corroborate the warning issued by the application monitor. These logs may also provide insights into I/O errors that are reported by storage devices or file-system errors. These file-system events are especially common in network

file systems in which communication links may be busy or partitioned because of failures. Once the occurrence of the rule violation is correlated in history with the file-system monitoring agent, a user-predefined direction to the fault coordinator could allow the recovery to be initiated by picking up a healthy checkpoint from before the fault. For this test case, the checkpoints/restart events are indicated by vertical dotted lines in Figure 5; the ones immediately before the issuance of a warning by the application monitor (CP_2 for the red simulation and CP_3 for the blue simulation) were used to restart the healthy application.

The relevance of this synthetic test case is that it demonstrates the ability of the proposed fault management system to catch silent errors. It could be argued that the hardware fault detection mechanism associated with hard disks and the software mechanisms in the file system will be able to catch a significant proportion of errors that lead to corruption. Moreover, the application will generate warning messages and quit if there is a significant amount of data corruption (file format violations). However, this synthetic test case provides an example of an additional mechanism in the multi-level hierarchy that can allow detection of errors associated with file systems and storage.

Implementation on HPC architectures: This test was also conducted on a Linux cluster, similar to the MD test case. However, the overall cooperative fault management system described for this test case is also fully applicable to current HPC and future extreme-scale systems.

Selection of check-point considerations: Checkpointing and selection of checkpoints are important for ensuring end-user productivity in case of failures and recovery [5, 50-51]. In the

proposed approach, violation of the health rules is based on the assumption that the silent errors are almost instantaneous and affect the simulations immediately so that the rule violation is trigged within a few steps of the simulations. If such an event occurs, a healthy or error-free checkpoint is selected from a state that is considered healthy, based on the defined rules. However, it is possible to envision cases in which the silent errors are small and do not cause immediate rule violations but rather a drift in the health parameter. Such cases would be much harder to trap with simple rules. With user experience, detailed knowledge of the health parameters will be useful to define tighter rules to trap slow drift (and not only sudden violations).

Another approach to ensuring that the selected check point is error-free is performing short, multiple simulation restarts with checkpoints going several states back. If the simulations are deterministic, a healthy state will be marked by the last checkpoint that provides values for health parameters that agree with those from older checkpoints. Such an approach would require dedicating somewhat more resources; but as it can be completely automated, it would provide increased end-user productivity with more control by ensuring that the checkpoint is error-free.

Cost of recovery considerations: One of the benefits of the concurrent monitoring of the application health parameters is that it involves minimal cost (time) during the recovery process. The selection criteria for the health parameters include the ability to access the information at a relatively low cost. In the two test cases, this was done by live monitoring (by `grep`) of text files. The cost of this step was trivial, as the output files are generated typically at rate of <1 MB/sec. As the decision agent is run simultaneously, the real cost in recovery is associated with issuing an application termination signal, waiting for the unhealthy application to terminate,

and restarting the application at a previously healthy checkpoint. The decision of which checkpoint is used for restart is made when the warning is triggered (rule violation); therefore, it does not require any additional analysis or resources. If the simulation is being performed within a runtime allocated by a queue manager, the recovery and restart steps are also performed within the allocated time and resources.

The cost of recovery for the two test cases was less than 1 minute for issuing the application termination signal, a grace period for cleanup, and relaunching the application. MD and electronic structure calculations require run for hours to several days. Therefore, this approach resulted in a significant improvement in productivity. Note that the recovery cost does not include the time already spent between the healthy state and the point when the health warning was triggered. Further, it does include the case when termination, cleanup, and application relaunching could be delayed because of other factors. It can be envisioned that as this approach is used, a good set of health rules will be developed by the community which will allow the most optimal wait between the unhealthy behavior and when the warning is triggered.

Performance modeling of recovery: Realistic performance modeling would require comparing the average time-to-solution for an ensemble of regular jobs submitted to a queue manager which fail as a result of fault injection, with the average time-to-solution based on jobs that use the proposed recovery approach triggered by rule violation. This comparison would only be meaningful when the collected data is based on a set of rules that allow an in-domain user to improve the overall productivity. In the current study, we propose only a limited set of rules. However, future research would explore selecting rules that improve user productivity and evaluating the performance of the framework and approach based on these rules.

4. RELATED WORK

Approaches for error detection based on basic textual logging (e.g., syslog) to numeric metric gathering (e.g., performance counters) have been studied extensively in prior work [6, 52-55]. There has been extensive research in the analysis of large-scale system monitoring data, specifically reliability, availability, and serviceability (RAS) and console logs [7, 8, 56-58]. The Chopstix [59] system employs a probabilistic approach to monitoring, whereby a *sketch* of monitoring events efficiently characterizes a state, which can be used for identification and diagnostic purposes. The FENCE framework analyzes data logs to explore potential markers for fault prediction [9]. Such system monitoring tools can provide input to a system-level fault information framework, making them complementary to our application-driven primary approach.

Monitoring, maintenance, and job logs were used in several studies to identify errors over the lifespan of a large-scale system, or successive generations of systems. Levy et al. studied the system logs to gain insights into memory errors [24]. They observed no significant correlation between correctable memory (DRAM) faults and subsequent uncorrectable faults that would serve as a predictive measure. It was also noted that no root cause was determined for the majority of "system down events"; this finding underscores the need for both improved monitoring and resilience mechanisms. The mean time between failures (MTBF) is a useful metric that applications can use to estimate failure-free work durations. Gupta et al. studied logs from five generations of systems at Oak Ridge National Laboratory spanning 2008–2015 and found a node-count normalized MTBF ranging from ~8 hours to ~23 hours [58]. The study found that most of the failures came from a relatively small set of failure types (causes) over the

five system generations, with machine check exceptions (MCEs)[1] being the top failure type in three of five systems (and the second and third most frequent failure type in the other two systems) [58].

Software-implemented fault tolerance (SIFT), which takes advantage of the redundancy available in distributed networks, was used to develop middleware providing fault tolerance capabilities [60]. SIFT provided a hierarchical error detection framework designed to adapt to changing application needs, including changes in throughput. SIFT test cases included space-borne applications, wireless telephone network controllers, and main memory database systems [61]. One of the limitations of SIFT was the high overhead introduced by redundancy, which might limit its benefit for user applications in an HPC environment.

The AutomaDeD project [14] is a statistical tool that uses timing models of the application to detect abnormalities in the runtime behavior of an application. It is similar to the health monitor presented in this paper in that neither require changes to the target application. However, although timing anomalies may be related to errors occurring within the system, they are at best secondary indicators. In fact, many other factors can also influence the execution times of different phases of an application, including the specifics of the simulation inputs and usage of the system (i.e., contention with other jobs for shared resources). The Trident tool [62] is a compiler-based approach that seeks to model the effects of soft errors on a program overall and on individual instructions. Trident uses a probabilistic model to predict silent data corruption effects, in contrast to fault injection–based methods. A recent survey found the majority of exascale computing projects use MPI [18]. Therefore, work to add resilience capabilities to MPI [15-17] will be useful to help support resilience in next-generation applications.

[1] The Machine Check Exception (MCE) is the hardware mechanism used to report various hardware errors, e.g., bus errors, memory errors, CPU errors.

Autonomic computing has emerged as an approach to deal with large, complex software systems characteristic of many areas of modern computing [63]. For example, da Silva and Rebello presented a hierarchical MPI-based application management framework that provides fault tolerance by detecting failed tasks and framework components and restarting them or redistributing work to surviving framework components [64,65]. Similarly, Haupt et al. presented an approach for the autonomic execution of computational workflows which involves a distributed service-oriented architecture to monitor and respond to jobs or job steps in the distributed (grid-based) execution of a multidisciplinary design optimization workflow by rerunning them [66].

As we progress toward *exascale* HPC systems, it is expected that failures (both *hard* and *soft*) will become much more frequent events that the application must be prepared to handle [36, 24, 58, 67]. Bronevetsky and de Supinski explored the effect of soft errors on iterative linear algebra methods [68]. They used a fault injection technique to determine the impact of targeted soft errors and then explored algorithm-based fault tolerance techniques that could be integrated into the application to help detect and mitigate the effects of these errors. Currently, many HPC applications rely on checkpoint/restart rollback recovery fault tolerance techniques to support application recovery after a failure [10]. The storage requirements for checkpoint/restart techniques at *exascale* are a concern [27, 69]. Recent advancements in checkpoint targeted file-system optimizations [70-72] and innovative hardware designs [73] are likely to help mitigate this issue in the near term. Some applications are starting to experiment with algorithm-based fault tolerance (ABFT) techniques [11, 74, 75] and natural fault tolerance [12, 13] techniques as alternatives to traditional checkpoint/restart techniques. ABFT and natural fault tolerance techniques are intrusive approaches that allow an application to improve its ability to detect and

manage the effects of both soft and hard errors but require substantial changes to the code, leading to concerns regarding their widespread adoption in user communities, especially with legacy applications.

5. CONCLUSIONS AND FUTURE DIRECTIONS

The described methodology and cooperative fault management system enables the development of an application-centric fault detection and recovery mechanism to provide resilience on HPC architectures. Our methodology is not designed to detect all silent errors but only to capture the ones that impact application health. In combination with a set of user-defined rules and an FTB that allows the exchange of failure-related information between different layers, we demonstrated that it is possible to catch *silent errors* arising from memory or file-system (storage) failure. Moreover, we presented a methodology that could allow detection of the source of the errors, as well as the simulation steps where the failure started making an impact on the simulation. Based on this information, we were able to initiate an application recovery process from healthy checkpoints, thereby minimizing the resources wasted by erroneous calculations. Note that it is possible that the application monitor agent (or the system monitor) might issue a warning and no corroborating evidence be found. In such situations, it would be up to the user to define rules that would treat this warning as a silent error and attempt recovery, treat it as an application error and halt, or disregard the warning and continue.

This strategy was implemented for two different applications, MD using the LAMMPS code, and quantum chemistry (electronic structure) calculations using the GAMESS package. These two test cases were used to demonstrate the detection of simulated faults in the memory and I/O systems, which would remain undetected without the proposed scheme and would

require human intervention. The cooperative scheme demonstrates how future exascale systems and applications could be co-designed for improved resiliency and productivity.

As a proof of principle, the present work does not provide for resilience of the monitoring framework, which may be considered a requirement for a production-quality system. That issue could be addressed straightforwardly by implementing the monitoring framework in a redundant fashion. As the monitoring framework is not computationally intensive, implementing the proposed approach would not add significantly to the cost of a simulation. This approach could protect against both node failures and data corruption within the framework. Another important aspect of this research concerns the ability to distinguish between *unhealthy* behavior in an application due to silent error(s) and unhealthy behavior due to a bad simulation state. Sometimes bad simulation states result from bad input parameters or other issues associated with the simulations. There are strategies that could be used to distinguish between these two scenarios by rolling back the simulations and rerunning them with a different set of hardware resources. If the unhealthy state occurs again, it is most likely due to a bad simulation state. However, if it does not reoccur or if a new type of issue with the application is triggered, then there is a better chance the unhealthy state was caused by a silent error. The health monitoring strategies and framework described herein would allow such functionalities to be implemented in the future. Similar approaches may be useful to distinguish between software bugs and data corruptions.

Additional future work in this area will focus on generalization of the implementation to other domains, as well as collecting application-specific information that allows better definition of the user-defined rules for health monitoring. Another area of focus will be to integrate application timing–based rules similar to those in the AutomaDeD project. However, external

events can strongly affect the performance (timing) of applications, for example, a burst of I/O by another application. The proposed cooperative fault management system could also include statistical timing information for the application progress, and a violation in timing-based ruled rules could be used to trigger an investigation.

ACKNOWLEDGMENTS

Financial support for this work was provided by NIH (R21GM083946 and R01GM105978) to PKA, and the U. S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research through the project on Coordinated and Improved Fault Tolerance for High Performance Computing Systems (CIFTS) project. We thank various members of the CIFTS team, Sadaf Alam, Christian Engelmann, and Hai Ah Nam for discussions and feedback.

REFERENCES

[1] A. Geist, and D. A. Reed. A survey of high-performance computing scaling challenges. *The International Journal of High Performance Computing Applications*, *31* (1), 104-113, 2017.

[2] C. McNairy. Exascale fault tolerance challenge and approaches. In *Reliability Physics Symposium (IRPS), 2018 IEEE International*, pages 3-4. IEEE, 2018.

[3] A. Agrawal, G. H. Loh, and J. Tuck. Leveraging near data processing for high-performance checkpoint/restart. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, article 60. ACM, 2017.

[4] R. Gupta, P. Beckman, B. H. Park, E. Lusk, P. Hargrove, A. Geist, A. Lumsdaine, and J. Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *International Conference on Parallel Processing (ICPP),* 2009.

[5] A. Fang, and A. A. Chien. ABFR: Convenient management of latent error resilience using application knowledge. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 27-39, 2018.

[6] J.M. Brandt, A.C. Gentile, D.J. Hale, and P.P. Pebay. Ovis: a tool for intelligent, real-time monitoring of computational clusters. In *Parallel and Distributed Processing Symposium at the IPDPS 2006*, page 8, April 2006.

[7] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. BlueGene/L Failure Analysis and Prediction Models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'06),* pages 425–434, 2006.

[8] John P. Rouillard. Real-time log file analysis using the simple event correlator (SEC). In *Proceedings of the 18th USENIX conference on System administration*, pages 133–150, Berkeley, CA, USA, 2004. USENIX Association.

[9] Z. Lan, Y. Li, Z. Zheng, and P. Gujrati. Enhancing application robustness through adaptive fault tolerance. In *IEEE International Symposium on Parallel and Distributed Processing, (IPDPS)*, pages 1 –5, April 2008.

[10] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. *A survey of rollback-recovery protocols in message-passing systems.* ACM Computing Surveys, 34(3):375–408, 2002.

[11] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, 1984.

[12] A. Geist and C. Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors. *Journal of Parallel and Distributed Computing*, 2002.

[13] C. Engelmann and A. Geist. Super-scalable algorithms for computing on 100,000 processors. In *Proceedings of International Conference on Computational Science (ICCS)*, volume 3514, pages 313–320, May 2005.

[14] G. Bronevetsky, I. Laguna, S. Bagchi, B. R. de Supinski, D. H. Ahn, and M. Schulz. Statistical fault detection for parallel applications with AutomaDeD. In *IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE),* pages 1–6, Mar 2010.

[15] Bland, W., Bouteiller, A., Herault, T., Hursey, J., Bosilca, G., & Dongarra, J. J. An evaluation of user-level failure mitigation support in MPI. *European MPI Users' Group Meeting*, pages 193-203), September 2012.

[16] Chakraborty, S., Laguna, I., Emani, M., Mohror, K., Panda, D. K., Schulz, M., & Subramoni, H. EReinit: Scalable and efficient fault-tolerance for bulk-synchronous MPI applications. *Concurrency and Computation: Practice and Experience*, e4863, 2017.

[17] Gamell, M., Katz, D. S., Teranishi, K., Heroux, M. A., Van der Wijngaart, R. F., Mattson, T. G., & Parashar, M. Evaluating online global recovery with fenix using application-aware in-memory checkpointing techniques. *2016 45th International Conference on Parallel Processing Workshops (ICPPW),* pages 346-355, August 2016.

[18] Bernholdt, D. E., Boehm, S., Bosilca, G., Gorentla Venkata, M., Grant, R. E., Naughton, T., ... & Vallee, G. R. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, e4851, 2017.

[19] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. *Commun. ACM* 54, 2, 100-107, 2011.

[20] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *International Conference on Measurement and Modeling of Computer Systems SIGMETRICS'07,* San Diego, California, USA. 2007.

[21] B. Panzer-Steindel. Data integrity. Technical Report CERN/IT, 2007. http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1&materialId=paper&confId=13797

[22] J. Eddy. Silent data corruption in SATA arrays: A solution. White paper, NEC Corporation of America. 2008. https://www.necam.com/Docs/?id=54157ff5-5de8-4966-a99d-341cf2cb27d3

[23] Tiwari, D., Gupta, S., Rogers, J., Maxwell, D., Rech, P., Vazhkudai, S., ... & Carro, L. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331-342, February 2015.

[24] Levy, S., Ferreira, K. B., DeBardeleben, N., Siddiqua, T., Sridharan, V., & Baseman, E. Lessons learned from memory errors observed over the lifetime of Cielo. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 554-565, November 2018.

[25] W. M. Jones, J. T. Daly, N. A. DeBardeleben. Application Resilience: Making progress in spite of failure. In *Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID),* pages 789-794, 2008.

[26] P. K. Agarwal, S. Hampton, J. Poznanovic, A. Ramanthan, S. R. Alam, and P. S. Crozier. Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. *Concurrency and Computation: Practice and Experience* 25, 10, 1356-1375, 2013.

[27] B. H. Park, T. J. Naughton, P.K. Agarwal, D. E. Bernholdt, Al Geist, and J. L. Tippens. Realization of user level fault tolerant policy management through a holistic approach for fault correlation. In *IEEE International Symposium on Policies for Distributed Systems and Networks, 2011.*

[28] Price, D. Pentium FDIV flaw-lessons learned. *IEEE Micro*, *15*(2), 86-88, 1995.

[29] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz. 2007. Extending stability beyond CPU millennium: A micron-scale atomistic simulation of Kelvin-Helmholtz instability. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing* (SC '07), article 58, 2007.

[30] E. Apra, A. P. Rendell, R. J. Harrison, V. Tipparaju, W. A. deJong, and S. S. Xantheas. Liquid water: Obtaining the right answer for the right reasons. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (SC '09), article 66, 7 2009.

[31] Malkowski, K., Raghavan, P., & Kandemir, M. Analyzing the soft error resilience of linear solvers on multicore multiprocessors. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1-12, April 2010.

[32] Shantharam, M., Srinivasmurthy, S., & Raghavan, P. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 69-78, June 2012.

[33] Shantharam, M., Srinivasmurthy, S., & Raghavan, P. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the international conference on Supercomputing*, pages 152-161, May 2011.

[34] Elliott, J., Hoemmen, M., & Mueller, F. Evaluating the impact of SDC on the GMRES iterative solver. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1193-1202, May 2014.

[35] Elliott, J., Hoemmen, M., & Mueller, F. A numerical soft fault model for iterative linear solvers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 271-274, June 2015.

[36] Guhur, P. L., Zhang, H., Peterka, T., Constantinescu, E., & Cappello, F. Lightweight and accurate silent data corruption detection in ordinary differential equation solvers. In *European Conference on Parallel Processing*, pages 644-656, August 2016.

[37] Di, S., & Cappello, F. Adaptive impact-driven detection of silent data corruption for HPC applications. *IEEE Transactions on Parallel and Distributed Systems*, *27*(10), 2809-2823, 2016.

[38] Guhur, P. L., Constantinescu, E., Ghosh, D., Peterka, T., & Cappello, F. Detection of Silent Data Corruption in Adaptive Numerical Integration Solvers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 592-602, September 2017.

[39] Dubey, A., Fujita, H., Graves, D. T., Chien, A., & Tiwari, D. Granularity and the cost of error recovery in resilient AMR scientific applications. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 492-501, November 2016.

[40] Fang, A., Cavelan, A., Robert, Y., & Chien, A. A. Resilience for stencil computations with latent errors. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 581-590, August 2017.

[41] Fang, A., & Chien, A. A. ABFR: convenient management of latent error resilience using application knowledge. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 27-39. June 2018.

[42] S. Plimpton. Fast parallel algorithms for short-range molecular-dynamics. *J. Comp. Physics,* 117, pages 1-19, 1995.

[43] http://lammps.sandia.gov/

[44] S. S. Hampton, S. R. Alam, P. S. Crozier, and P. K. Agarwal. Optimal utilization of heterogeneous resources for biomolecular simulations. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (SC '10), pages 1-11, 2010.

[45] A. Nallamuthu, M. C. Smith, S. Hampton, P. K. Agarwal, and S. R. Alam. Energy efficient biomolecular simulations with FPGA-based reconfigurable computing. In *Proceedings of the 7th ACM international conference on Computing frontiers* (CF '10), pages 83-84, 2010.

[46] Intel Corporation. Intel (c) 64 and IA-32 Architectures Software Developer's Manual—Volume 3A: System Programming Guide, Part 1, April 2011. Order Number: 253668-038US. http://www.intel.com/products/processor/manuals

[47] Andi Kleen. mcelog: Memory error handling in user space, September 21-24, 2010. Proceedings of Linux Kongress 2010. http://www.linux-kongress.org/2010

[48] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S .Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery. General atomic and molecular electronic structure system. *J. Comput. Chem.,* 14, pages 1347-1363, 1993.

[49] http://www.msg.ameslab.gov/gamess/

[50] Chien, A., Balaji, P., Dun, N., Fang, A., Fujita, H., Iskra, K., ... & Richards, D. (2017). Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. In *The International Journal of High Performance Computing Applications*, *31*(6), 564-590.

[51] Lu, G., Zheng, Z., & Chien, A. A. When is multi-version checkpointing needed? In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*, pages 49-56, June 2013.

[52] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing,* 30(7):817 – 840, 2004.

[53] KyoungSoo Park and Vivek S. Pai. CoMon: A mostly-scalable monitoring system for PlanetLab. In *ACM SIGOPS Operating Systems Review*, 40(1):65–74, January 2006. Special Issue on PlanetLab.

[54] Matthew J. Sottile and Ronald G. Minnich. Supermon: A high-speed cluster monitoring system. In *Proceedings of IEEE International Conference on Cluster Computing (Cluster'02),* pages 39–46, 2002.

[55] V. Ahlgren, S. Andersson, J. M. Brandt, N. Cardo, S. Chunduri, J. Enos, P. Fields, A. C. Gentile, R. Gerber, M. Gienger, J. Greenseid, A. Greiner, B. Hadri, Y. He, D. Hoppe, U. Kaila, K. Kelly, M. Klein, A. Kristiansen, S. Leak, M. Mason, K. T. Pedretti, J-G Piccinali, J. Repik, J. Rogers, S. Salminen, M. Showerman, C. Whitney and J. Williams, Large-Scale System Monitoring Experiences and Recommendations, In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 532-542, 2018.

[56] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN).* IEEE Computer Society, June 25- 28, 2007.

[57] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series,* 78(012022), 2007. SciDAC 2007.

[58] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh 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*, article 44, 2017.

[59] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry L. Peterson. Lightweight, high-resolution monitoring for troubleshooting production systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 103–116. USENIX, December 8-10, 2008.

[60] Bagchi, S.; Srinivasan, B.; Whisnant, K.; Kalbarczyk, Z.; Iyer, R.K., Hierarchical error detection in a software implemented fault tolerance (SIFT) environment, *Knowledge and Data Engineering, IEEE Transactions on* , 12(2) pages 203-224, Mar/Apr 2000.

[61] Kalbarczyk, Z.; Iyer, R.K.; Wang, L., Application fault tolerance with Armor middleware, *Internet Computing, IEEE* , 9(2), pages 28-37, March-April 2005.

[62] Li, G., Pattabiraman, K., Hari, S. K. S., Sullivan, M., & Tsai, T. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 27-38, June 2018.

[63] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland, A concise introduction to autonomic computing, *Advanced Engineering Informatics*, 19(3), pages 181–187, July 2005.

[64] J. da Silva and V. Rebello, Low cost self-healing in MPI applications. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, F. Cappello, T. Herault, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 4757, pages 144–152, 2007.

[65] J. A. da Silva and V. E. F. Rebello, A hybrid fault tolerance scheme for EasyGrid MPI applications, In *Proceedings of the 9th International Workshop on Middleware for Grids, Clouds and e-Science*, pages 4:1–6, 2011.

[66] T. Haupt, N. Sukhija, and I. Zhuk, Autonomic execution of computational workflows, In *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 965 –972, 2011.

[67] Nie, B., Tiwari, D., Gupta, S., Smirni, E., & Rogers, J. H. A large-scale study of soft-errors on GPUs in the field. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 519-530, March 2016.

[68] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS),* Jan 2008.

[69] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[70] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.

[71] G. Bronevetsky and A. Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report LLNL-TR-415791, Lawrence Livermore National Laboratory, 2009.

[72] R. Prabhakar, S. S. Vazhkudai, Y. Kim, A. Butt, M. Li, and M. Kandemir. Provisioning a multi-tiered data staging area for extreme-scale machines. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS 2011)*, pages 1–12, Mar 2011.

[73] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.

[74] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM Journal of Scientific Computing*, 30(1):102–116, 2007.

[75] Z. Chen. 2013. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN*

symposium on Principles and practice of parallel programming (PPoPP '13), pages 167-176, 2013.