Error Checking and Snapshot-Based Recovery in a Preconditioned Conjugate Gradient Solver

Zachary Rubenstein*, James Dinan†, Hajime Fujita*‡, Ziming Zheng*, Andrew A. Chien*‡
*Department of Computer Science, University of Chicago
†Intel Corp.
‡ Argonne National Laboratory, Mathematics and Computer Science Division
zar1@cs.uchicago.edu, james.dinan@gmail.com, {hfujita, zimingzheng, achien}@cs.uchicago.edu

***Abstract*—Soft errors are a significant concern for high-performance computing systems in the exascale time frame. We apply our group's Global View Resilience (GVR) library to a preconditioned conjugate gradient solver, evaluating per-data-structure snapshots and varied error detection approaches to tolerate soft errors.**

Using 14 real-world matrices from the University of Florida Sparse Matrix Collection, we use error-injection to assess the viability of several detection and correction schemes. These studies show: 1) though inexpensive,residual-based detection performs poorly. To achieve acceptably low false negative rates, much higher (20x) false positives rates are required. 2) though more expensive, algorithm-based detection performs better overall, achieving much lower false negative rates at one fifth the false positive rate. Even this "expensive" error detection is inexpensive compared to a single iteration, and therefore is viable for linear solvers—particularly in high fault-rate systems.

***Keywords*-fault-tolerant computing; high-performance computing; numerical computing**

I. Introduction

The high-performance computing community is focused on a significant increase in performance from todays petascale systems (10^{15} sustained floating point operations per second) to exascale systems (10^{18} sustained floating point operations per second). That leap poses major challenges in terms of increased parallelism, variability of performance, and our focus here—an increase in system fault and error rates due to hardware scaling to deep submicron features and low-voltage for energy efficiency [1]–[6]. These rising error rates combined with the large size of future systems (100,000 to 1M nodes) threaten the usability of such exascale systems for large scientific computations (see blue-ribbon panel reports [7]–[9]) with projected MTTI ranging from a few minutes to an hour [4], [5], [10], [11] and making traditional, global checkpoint-restart approaches too expensive. In short, there is a need for novel approaches to ensure reliable computing for these systems.

While there are varied classifications of faults and errors [2], [12], [13], our focus here is on faults that give rise to *soft errors* and particularly *intermittent errors* that affect the execution of a particular computation. Such intermittent soft errors include memory errors, cache errors, interconnect errors, logic errors, timing errors, etc. that corrupt application data. For example, an error may cause an application floating point value to be changed to a different floating point value that is valid, but incorrect in the context of the computation.

A major concern with intermittent, soft errors is the potential for escaped errors (EE). We define an escaped error as a soft error that changes computation results and is not corrected. These errors are variously called *latent* or *silent* in the literature [7], [12], [14]. Escaped errors can cause an application to return an incorrect answer, but the application will be unable to inform the user that the returned answer is incorrect.

The major challenges with soft errors (and the risk of escaped errors) are detection and correction. Prior approaches include replication [15], or application-based techniques [16] to introduce and manage redundancy in data an computation. Examples of application-directed tolerance range from application-level checkpoint-restart (capturing application-specified data) to algorithms that are restructured to tolerate specific classes of errors [13], [16]. Here, we focus on application-directed tolerance in a widely used numerical algorithm: preconditioned conjugate gradient [17], [18], which is viewed as representative of a large class of solver applications. We study a range of techniques to detect soft errors , and we characterize their precision and recall. To evaluate the impact of this fault-tolerance (or resilience) on overall performance, we utilize the Global View Resilience (GVR) library [19] being developed by our group at the University of Chicago and Argonne National Laboratory. GVR provides applications with fine-grained control over data structures. Each application data structure can be separately versioned (persisted), and recovered (application-specific error handling).

Practical tolerance of *latent* soft errors must balance runtime overhead against achieving an acceptable escaped error probability. So, we study the effectiveness of error detection schemes in a PCG solver built on the well-known Trilinos library [20]. We study each scheme using random error injection and observe how each performs in terms of detection precision, detection recall, and escaped

error probability. We also characterize the runtime overhead, comparing the fault-tolerant version to a baseline PCG solver with no fault tolerance. Our results show that, in contrast to folklore, when all costs are factored in, residual-based detection methods are not viable for PCG. In contrast, more expensive but more accurate methods are preferable.

The specific contributions of this paper include:

- studies of PCG which show that despite its low cost, residual-based error detection is not viable, largely because achieving high recall produces unacceptably high rates of false positives. Specifically, to achieve recall of 90%, false positive rates increase to one false positive per iteration.
- studies of algorithm-based error detectors for PCG that are more expensive but are viable—achieving high recall with acceptable false positive rates. Specifically, algorithm-based detection can achieve a recall of over 95% while expecting a false positive on fewer than one fifth of the iterations.
- studies showing that algorithm-based detection costs can be small, averaging 25% of the basic PCG computation cost (excluding any versioning overhead).
- a first for GVR API: adding resilience to a PCG solver, and showing that GVR can be added to an application conveniently— requiring modification of only 2465 lines of code in an application that exercises 390512 lines of code (that is, $< 1\%$ of the total LOC).

The remainder of this paper is organized as follows. In Section II, we describe relevant background for linear systems and reliability. In Section III we describe our numerical experiments which use different detection and correction schemes in a faulty environment. Section IV discusses our observations from said experiments. Section V discusses related work. Section VI concludes the paper and discusses future work.

II. Background

A. Linear System Solvers

A large number of scientific applications require that the solution x be found to a linear system of the form $Ax = b$. These solutions can be found by way of direct solvers, such as Gaussian elimination. However, in the case of many large problems, direct methods can often require a prohibitively large amount of computation.

A popular alternative to direct solvers are iterative solvers. Iterative solvers approximate x with increasing accuracy at every iteration. Examples of iterative solvers include stationary methods like Successive Over-Relaxation (SOR), and Krylov subspace methods like Conjugate Gradient (CG) or Generalized Minimal Residual Method (GMRES). In this study, we focus on Preconditioned Conjugate Gradient (PCG).

1: $r = b - A * x$
2: $\text{iter} = 0$
3: **while** $(\text{iter} < \text{max\_iter})$ and $\|r\| > \text{tolerance}$ **do**
4: $\quad \text{iter} = \text{iter} + 1$
5: $\quad z = M^{-1} * r$
6: $\quad \rho_{\text{old}} = \rho$
7: $\quad \rho = r' * z$
8: $\quad \beta = \rho / \rho_{\text{old}}$
9: $\quad p = z + \beta * p$
10: $\quad q = A * p$
11: $\quad \alpha = \rho / (p' * q)$
12: $\quad x = x + \alpha * p$
13: $\quad r = r - \alpha * q$
14: **end while**

Figure 1. The preconditioned conjugate gradient algorithm. Given a matrix A, a solution b, and a preconditioner M, we attempt to converge on a solution x. Each iteration attempts to reduce the residual r by moving x in direction p.

1) Conjugate Gradient (CG): As in all Krylov subspace methods, CG aims to move x in a unique dimension of Krylov subspace every iteration. In every iteration, a direction orthogonal to the previous direction is found by a procedure resembling Gram-Schmidt orthogonalization, and then x is moved in that direction by a step size determined by other aspects of the state of the calculation.

2) Preconditioned Conjugate Gradient (PCG): One approach to speeding up the convergence of CG is by applying a preconditioner M to A and b and then solving the equation $MAx = Mb$ [18, p. 276]. The justification behind this procedure is that it may be less expensive to solve $MAx = Mb$ than to solve $Ax = b$. Figure 1 elucidates the PCG algorithm. There are a number of different choices for M. We opt for incomplete Cholesky factorization [21] using an arbitrary drop threshold of 0.001 [18, p. 321-330].

In relation to PCG, we will later refer to the vector p, which records the direction that x is moved in the current iteration; the vector r, which is principally identical to the residual $b - Ax$, but is updated in-place for optimization purposes rather than being calculated as $Ax - b$ in each iteration; and ρ, which records the norm of r from the previous iteration.

The norm residual $\|r\|$ for PCG is expected to converge at an exponential rate [18, p. 215].

B. Exploiting Global View for Resilience (GVR)

The Global View Resilience project (GVR) [14], [19], [22], [23] provides a library for parallel scientific applications to perform application-directed fault tolerance. GVR enables the application to create global data store (GDS) objects, which have a number of important properties:

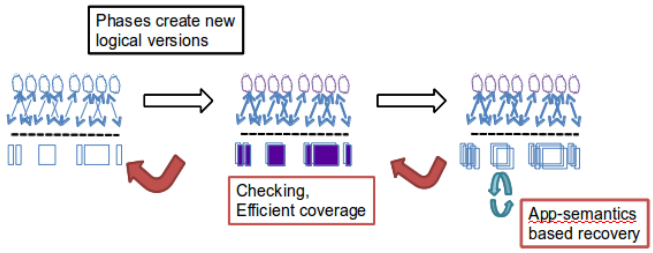

Figure 2. GVR utilizes a data-oriented view in which data in a given GDS is in a stable state, then the application operates on data, then the data is in another stable state. In this figure, data is represented by rectangles below the dotted lines, while some application-defined calculation is represented by the arrows and clouds above the lines. In the transition from the state on the left to the state in the middle, the data has been transformed by the application and reaches another stable state. When the application declares this new stable state, GVR may preserve the old version of the data, while checking to make sure that the new version is consistent with the expectations of the application. Transitioning from the middle state to the right state, GVR takes another version of the data. Finally, the application may utilize many preserved versions simultaneously in order to recreate a stable state after an error has occurred.

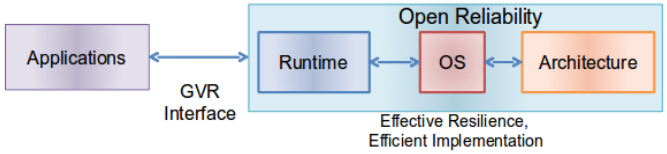

Figure 3. The GVR library provides a unified interface for checking for, signalling, and recovering from errors originating in either the application, or in different parts of the underlying system (runtime, OS, or underlying hardware).

1) A GDS object is has a global name, and is efficiently accessible via one-sided remote memory access (RMA or RDMA).
2) A GDS object can be versioned (user-defined persistent snapshot), and these persistent copies are used in error recovery (See Figure 2). These versions may be taken at application-defined "stable points," which are points in the computation at which the data are considered to be in a consistent state, and, consequently, fit for preservation. The GVR framework may choose either to take or not to take a version at a stable point, depending on hints from the application about the relative importance of performance versus being up to date for the GDS.
   Multiple versions are particularly useful for latent (or silent) error recovery. If an error persists across multiple versions, then the error may be present in at least one of the most recent snapshots. Consequently, it is necessary to recover from an older version in order to restore correctness.
3) Each GDS object has application-specific callback routines for error-checking and error-recovery (see Figure 3). Error-recovery routines can respond to errors raised by either the application or the system. When an error is raised, an application-specified error recovery routine can do a number of things. For example, it can recover using a number of old versions of the data in the GDS, it can recovery in some other way, or it can raise another error that will be handled by another recovery routine. Errors can be handled either by processes acting alone, or by processes acting in a coordinated manner.
4) Each GDS object has custom multi-versioning, error-checking, and error-recovery schemes. For example, GVR can take a snapshot of the GDS that preserves the x vector every iteration, while taking a snapshot of the GDS that preserves the p vector on every other iteration. Still other GDS objects may only utilize one persistent snapshot, and may utilize error checks and recovery methods that are not discussed in this paper, like using parity to verify correctness.

This experiment makes simple, basic use of GVR; GVR is used in its capacity to preserve and restore snapshots of the x, p and r vectors and the scalar ρ in PCG. It would also be within the capabilities of GVR to depend on GVR to execute error checking and recovery schemes and preserve other variables in the calculation.

III. Methods and Terminology

Axes of experimentation are enumerated in Table III. In the remainder of this section, we discuss the definition of the different experimental configurations, and how we choose between them to formulate experiments.

A. Linear Algebra Problems

We take as candidates the same 28 matrices as Shantharam from the University of Florida sparse matrix collection [24]. These are all symmetric, positive definite. See Shantharam [25, p. 157] for a complete list.

We eliminated 11 matrices that converged in less than 8 iterations. We further eliminated 3 matrices which required much large numbers of iterations - to avoid their skewing the overall study. This left 14 matrices requiring 8 to 128 iterations to converge in error-free circumstances. This regularized workload allows error detection and correction methods applied periodically to be reasonably compared.

Ultimately, we used the following 14 matrices: bcsstk01, bcsstk07, bcsstk09, bcsstk16, fv3, gyro_m, Kuu, lund_b, mhd4800b, mhdb416, msc00726, nos5, nos6 and plbuckle.

B. Error injection

Shantharam showed that errors affecting the row with the highest Euclidean norm in the A matrix have maximum impact on convergence [25]. So, we inject errors into said row for each matrix by multiplying the entry by 4. Other studies have shown that errors of large magnitude are often easier to detect, and those of smaller magnitude often fail

to affect the numerical outcome. [25], [26] We believe such intermediate magnitude errors are most likely to cause EE.

1) Taking Snapshots: Several of the error-correction methods require snapshots. In PCG we use snapshots to preserve current versions of x, r, p, and the scalar ρ. These snapshots can be used to restore correct program values.

C. Error-Detection

We use several schemes to detect errors based on the algorithmic structure of the PCG solver. These detectors are summarized in Table I.

1) Residual-Based Detection: When running a PCG, we expect the norm residual, $\|r\|$ to decrease generally as the solver converges. The tricky thing is that it does not always decrease from iteration to iteration. In residual-based detection, we monitor $\|r\|$ during the course of the solve. If $\|r\|$ does not decrease as expected, we signal that an error has occurred. In order to define what sort of behavior in $\|r\|$ we do or do not expect, We use two residual error-detection methods [27],

Multiple-Based Detection (MD) If the current residual is larger than the previous residual multiplied by some constant factor m, signal an error.

Average-Based Detection (AD) If the current residual is larger than the mean of the last a residuals, signal an error.

These two methods each have one free parameter, which we train using the following procedure, also derived from Bronevetsky [27], for each combination of matrix and solver:

1) For all test matrices other than the matrix in question, find the most relaxed value for the free parameter that flags no errors for an error-free run.
2) Eliminate the highest and lowest 10% of values.
3) Take the average of the remaining top third, middle third, and bottom third of values. We label these parameters "HIGH," "MED," and "LOW" respectively.

For some matrix-method combinations, a parameter that generates no false positives was not achievable. These combinations were not taken into account in the above procedure.

In our figures, each detection scheme appears next to its parameter. For example, "MD(HIGH)" signifies an MD solver with HIGH sensitivity.

2) Algorithm-Based Detection: We exploit algorithm structure, periodically checking two key invariants [28] that are inherent properties of the PCG algorithm and should only diverge from reality due to either large round-off error or faults.

1) An extra vector-vector dot product to ensure that the current direction vector, p, is orthogonal to the previous direction vector. If $p^{(i)}$ means vector p at iteration i, we test whether:

$$\frac{p^{(i+1)T} A p^{(i)}}{\|p^{(i+1)}\| \cdot \|Ap^{(i)}\|} > \text{tolerance}$$

2) An extra matrix-vector product to ensure that the residual vector, r, is still equal to $b - Ax$. Specifically, we test whether:

$$\frac{\|r + Ax - b\|}{\|b\| \cdot \|A\|} > \text{tolerance}$$

If either of these invariants proves false, we take it as a symptom that an error has occurred.

The algorithm-based detectors also have an implicit parameter—the amount of divergence from these invariants that should be tolerated. We take this tolerance to be the same as the maximum norm residual at which convergence will be declared, $1e-6$. We report results from running algorithm-based detection every 1, 2, 3, and 4 iterations. Figures refer to this detection scheme as "ABFT" followed by the detection interval. "ABFT(2)" means algorithm-based detection that is run on every 2nd iteration.

3) Baseline Detectors: Finally, we compare against two baseline detectors:

- The "Immediate" detector is an oracle that always detects an error immediately after it is injected.
- The "Ignore" detector never detects errors.

D. Error-Recovery

In the event that an error-detecting scheme signals an error, we employ one of these error-recovery schemes. These are summarized in Table II.

1) Snapshot-Based recovery: For all snapshot-based schemes, we preserve at some point x, p, r and ρ. The schemes differ according to the point in the solver when they take snapshots.

The two snapshot-based schemes are:

Restart We take a snapshot of mutable state before the first iteration of the linear solver. If an error is detected, then we restore the state from the beginning of the solver.

Snapshot/Restore Snapshots mutable state before the first iteration and every W iterations after that. If an error is detected, restores the state corresponding to the last snapshot. We experiment setting W as 1, 2, 3, and 4.

2) Baseline recovery: In addition, we have the **Nop** scheme, which does nothing to preserve or restore state.

E. Experiments

We chose the composition of our experiments by sampling two aspects of error injection stochastically, and then deciding on other parameters either by exhaustion or deterministically. Each of these steps corresponds to a row in Table III, in order from the first row to the last row.

1) Every trial employs a PCG solver using ILUT .001 as a preconditioner.
2) From all candidate matrices, we choose a matrix A randomly. Matrices are weighted in proportion to the number of iterations required for the solver to

Method	Abbreviation	Parameters	Description	
Residual-based	Multiple-Based	MD	LOW, MED, HIGH	Signal an error if the current value for $\|r\|$ is larger than the previous value multiplied by some constant m. The parameters LOW, MED, and HIGH signify an m that has respectively low, medium, and high sensitivity to unexpectedly high values of $\|r\|$.
	Average-Based	AD	LOW, MED, HIGH	Signal an error if the current value for $\|r\|$ is larger than the last a values of $\|r\|$. The parameters LOW, MED, and HIGH signify an a that has respectively low, medium, and high sensitivity to unexpectedly high values of $\|r\|$.
Algorithm-based	Algorithm-based	ABFT	1, 2, 3, 4	Periodically does extra linear algebra operations to check whether the current p is orthogonal to the previous p and that $r = Ax - b$. The parameters 1, 2, 3 and 4 signify the frequency of of this check. ABFT(1) checks every iteration, ABFT(2) checks every 2 iterations, etc.
Baseline	Immediate	Immediate	N/A	An oracle: signals every error on the iteration after the error is injected.
	Ignore	Ignore	N/A	Signals no errors.

Table I
ERROR DETECTION SCHEMES

Method	Abbreviation	Parameters	Description	
Snapshot-based	Restart	Restart	N/A	Takes a snapshot of x, p, r and ρ before the first iteration. If an error is detected, restores state to state to the initial state.
	Snapshot/Restore	SR	1, 2, 3, 4	Takes a snapshot of x, p, r and ρ before the first iteration and every W iterations after that. If an error is detected, restores state to the most recent snapshot. The parameter signifies the value of W.
Baseline	Nop	Nop	N/A	Does nothing to preserve or restore state.

Table II
ERROR CORRECTION SCHEMES

Parameter		Sampling	Description
1)	Solver	Constant	PCG with ILUT .001
2)	Matrix (A)	Stochastic	The subset of Raghavan's matrices which converge in 8-128 iterations
3)	Error Injection Iteration (i)	Stochastic	Any iteration from 1 to the expected final iteration in an error-free environment
4)	Error Injection Vector (v)	Exhaustive	p, x, and r
5)	Error Injection Entry of Vector	Deterministic	Corresponding to row of A with largest norm
6)	Error Injection Severity	Constant	Existing entry multiplied by 4
7)	Error Detection Schemes (d)	Exhaustive	MD(LOW), MD(MED), MD(HIGH), AD(LOW), AD(MED), AD(HIGH), ABFT(1), ABFT(2), ABFT(3), ABFT(4), Immediate, Ignore
8)	Error Correction Schemes (c)	Exhaustive	Restart, SR(1), SR(2), SR(3), SR(4), Nop

Table III
THE EXPERIMENTAL SPACE

converge to a solution to $Ax = b$ for the chosen A in an error-free environment and using no fault tolerance. If matrix A_1 requires 20 iterations to converge in an error-free environment using the The Ignore detector and the Nop corrector (the Ignore_Nop scheme) and matrix A_2 requires 10 iterations to converge in an error-free environment using Ignore_Nop, then we are twice as likely to choose A_1 for a given set of experiments as we are to choose A_2.

3) We choose an iteration i in which to inject the error. i will be an integer from 1 to the number of iterations required for A converge in an error-free environment using Ignore_Nop. All iterations in this set are weighted uniformly.

We use the procedure described above to stochastically sample 50 (matrix, iteration) tuples (A, i). Then, for each tuple, we exhaustively experiment with all possible combinations of the remaining parameters (as shown in Figure 4).

4) We attempt injection into each vector v in the set of vectors p, x, and r.
5) We inject an error into the element of v corresponding to the row of A with the largest Euclidean norm.
6) When injecting the error, the chosen element of v is multiplied by 4.
7) We attempt to detect errors with each detection scheme d in the set: {MD(LOW), MD(MED), MD(HIGH), AD(LOW), AD(MED), AD(HIGH), ABFT(1), ABFT(2), ABFT(3), ABFT(4), Immediate,

Ignore}

8) We attempt to correct errors with each correction scheme c in the set: {Restart, SR(1), SR(2), SR(3), SR(4), Nop}

For each configuration (A, i, v, d, c), we run 50 trials of our PCG solver to solve $Ax = b$. Our solver uses d for error detection and c for error correction. During the solve, we inject an error in vector v during iteration i.

For each trial, we take the following measurements:

- **False positives**, **False negatives**, and **True positives** for the detector.
- **Convergence** - does the solver converges to a solution?
- And if the solver converges:
  - **Convergence Time**
  - **Correctness** is the solution correct (i.e. are there escaped errors)?

F. Outcomes

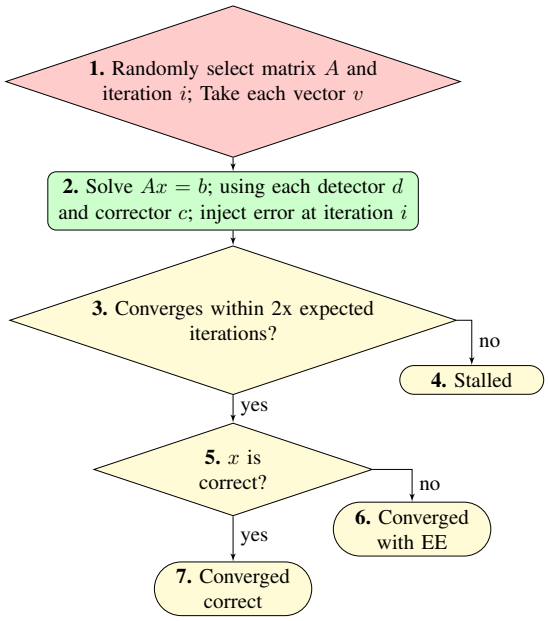

Figure 4. Experiment Flow for Error detection and recover study. For a trial, a matrix, error iteration, and vector are chosen (**1**), and we run PCG with all of the detection and correction schemes (**2**). Three outcomes are possible: convergence to the correct answer (**7**), convergence to an incorrect answer (**6**), and failure to converge (**4**).

Figure 4 depicts the progress of a series of experiments given a stochastically selected matrix and error injection iteration. After running the solver with a particular set of parameters (in step **2**), we see whether the solver converged in fewer than 2 times the number of iterations expected in and error-free environment (**3**). If so, we say that the solver has converged rather than stalled (**4**). If the solver has converged, it may still have converged to an incorrect answer due to EE. We test whether the solver returned an approximation for x such that Ax is adequately close to b (**5**). If the norm of $b - Ax$ is larger than tolerance, we say that the solver has **converged with escaped errors** (**6**). Otherwise, the solver has converged to the correct answer, so we say that the solver has **converged correctly** (**7**).

The question of which of these outcomes results is tied to the injection iteration, the injection vector, and the detector. If the detector fails to detect the injected error, then the detector has scored a **false negative**. A false negative, in turn, is likely to contribute to either a stalled solve or an escaped error.

A false negative can contribute to a stall if the corruption causes convergence to take longer, but does not change the result. For example, if p is corrupted, the next direction of search will likely not be orthogonal to the previous directions. Consequently, it will take more iterations to minimize over the same number of dimensions of Krylov space, so more iterations will be required to converge to tolerance. However, the residual vector r will still equal $b - Ax$, so the algorithm will not perceive that the norm residual is smaller than it actually is. Consequently, the algorithm will not converge with an escaped error.

A false negative can also change the result without requiring that convergence take a much longer time. For example, if r is corrupted, the solver will probably, in the next iteration, move it's approximate answer in a direction not orthogonal to the previous directions, since p is calculated from the last iteration's r. In addition to moving the wrong direction, r will no longer be equal to $b - Ax$, so the solver will not be aware of the actual norm residual. Consequently, the solver may converge at a time when $\|r\|$ is less than tolerance, but $\|b - Ax\|$ is not. In this eventuality, an escaped error has occurred, since the solver has converged to a value of x, but the value of x does not minimize $\|b - Ax\|$.

The detector may also score a **false positive** by flagging an error when none has occurred. A false positive will trigger an unnecessary restoration of the preserved vectors. A false positive will not affect correctness, but it will contribute additional runtime overhead due to the cost of restoration and the cost of rework from the time the last snapshot was taken. The rework will incur overhead not only in terms of runtime, but also in terms of iterations required to converge. Consequently, although a false positive cannot cause EE, a sufficient number of false positives will cause a stall.

G. Implementation and System

We ran all our experiments using a PCG solver built in C++ on the Trilinos library [20], [29]. This solver uses Trilinos linear algebra primitives. Trilinos is an object-oriented framework that implements a number of scientifically significant, parallelized algorithms, as well as the basic components necessary in order to implement those algorithms. The Trilinos implementation of the vector was augmented with GVR to provide multiversioning capabilities.

All experiments were run on single cores of 2.4 GHz Intel Xeon cpus on machines with 24 GB RAM.

Trilinos and GVR are both designed to allow scaling to large parallel cluster platforms without any source code change. Although this experiment only ran on single cores, a clear next step would be expanding the study to larger systems.

IV. Results

Our experiments' results are presented in the following order. First, we examine the data for stalls, convergence to EE, and convergence to the correct answer—varying error detector and recovery mechanism. These results represent the "bottom line" for each approach. Second, we examine the specific performance of the detectors in terms of false positives and false negatives. Finally, we will examine runtime overhead to reach a conclusion about the practical viability of the various schemes.

1) Overall Outcomes: Stalls, Convergence, and EE: Figure 5 examines the proportion of trials that stalled, converged with EE, or converged to the correct answer for each configuration. The independent axis specifies a particular detection and correction scheme. The detection scheme is first, then the correction scheme is next, separated by an underscore. For example, MD(HIGH)_SR(2) means that the bar aggregates over all trials using a high-sensitivity, multiple-based detector and taking snapshots every 2 iterations. The dependent axis specifies the proportion of trials with the given scheme that stalled, converged with EE, or converged correct.

For our perfect, immediate detector, most trials converge to the correct answer. Even for a perfect detector, an injection late in the calculation can still result in an EE. If the calculation is particularly short and the injection is late, an perfect detector can still undergo a stall.

For the detector that flags no errors, we see that PCG is naturally resilient to some errors. Although errors might be expected to delay convergence, in about $1/3$ of cases, the method still converges to the correct answer in fewer than 2 times the number of expected iterations. Most errors, however, cause EE. The prominence of EE underscores the fact that it is probably necessary to take some action in the real world to compensate for faults, even if a system appears to be functioning correctly.

Moving to our residual-based methods, we find a number of configurations that converge to the correct answer more frequently than Ignore. In the case of MD with SR(1), it appears that we can decrease EE without increasing stalls by increasing sensitivity. However, when we look at other snapshot intervals, it becomes apparent that increasing sensitivity usually decreases EE at the cost of admitting more stalls. This disparity is likely an artifact of our criteria for defining a stall. If false positives cause a solver to repeat every iteration, but it takes a snapshot at every iteration, then the method will require at most 2 times the number of expected iterations to converge. Consequently, a scheme using SR(1) is likely to converge. We conclude that residual-based methods generally trade reduced EE for increased likelihood to stall, or at least an increased number of false positives.

For algorithm-based detection, it appears that increasing detection frequency increases the proportion of converged correct trials while decreasing both the proportion of stalled trials and the proportion of trials with EE. That is, more frequent detection increases converged correct trials without any drawbacks that are apparent from this particular metric. In addition, we find that with our most aggressive algorithm-based scheme, ABFT(1)_SR(1), the solver converges correctly in over 80% of trials, which is unmatched in any of the other detection methods except for the perfect, immediate detector and the dubious MD(HIGH)_SR(1).

2) Error Detector Performance: Figure 6 shows the average number of false negatives for each each detection scheme. As we would expect, the proportion of false negatives generally follows the percentage of trials that converge with EE in Figure 5. We do notice, however, that for residual-based methods, it appears that the number of trials that stalled contributes to the number of trials that did not score false negatives. We attribute this correlation to high sensitivity, since a highly sensitive method that lacks precision will simultaneously increase the likelihood of true positives, the likelihood of false positives, the number of trials that converge without EE, and the number of stalled trials.

Figure 7 shows the average number of false positives for each detection scheme. For residual-based methods, we see a number of false positives corresponding with sensitivity. For algorithm-based methods, false positives remain remain relatively restricted.

We then perform two parameter sweeps across sensitivity parameters to better understand the tradeoff between false positives and false negatives for our detection schemes.

In Figure 8, we scatter-plot the average performance of each detector, using false negatives and false positives. For residual-based methods, a decrease in false negatives corresponds to a sharp increase in false positives. We find that each decrease in expectation of false negatives of 0.10 incurs an additional cost of about 2 false positives. We conclude that the sensitivity parameter for these methods is probably more significant to the likelihood of true positives than whether or not an error has actually occurred. The number of false positives at the extreme end (MD(HIGH)) is exceedingly large. If we divide the number of false positives by the average number of trials in an error-free run across our matrices, we get a value exceeding 1. This is possible because errors and error-recovery increase the number of iterations required, but the implication is still that, if the solver were to converge to an answer rather than stalling, we

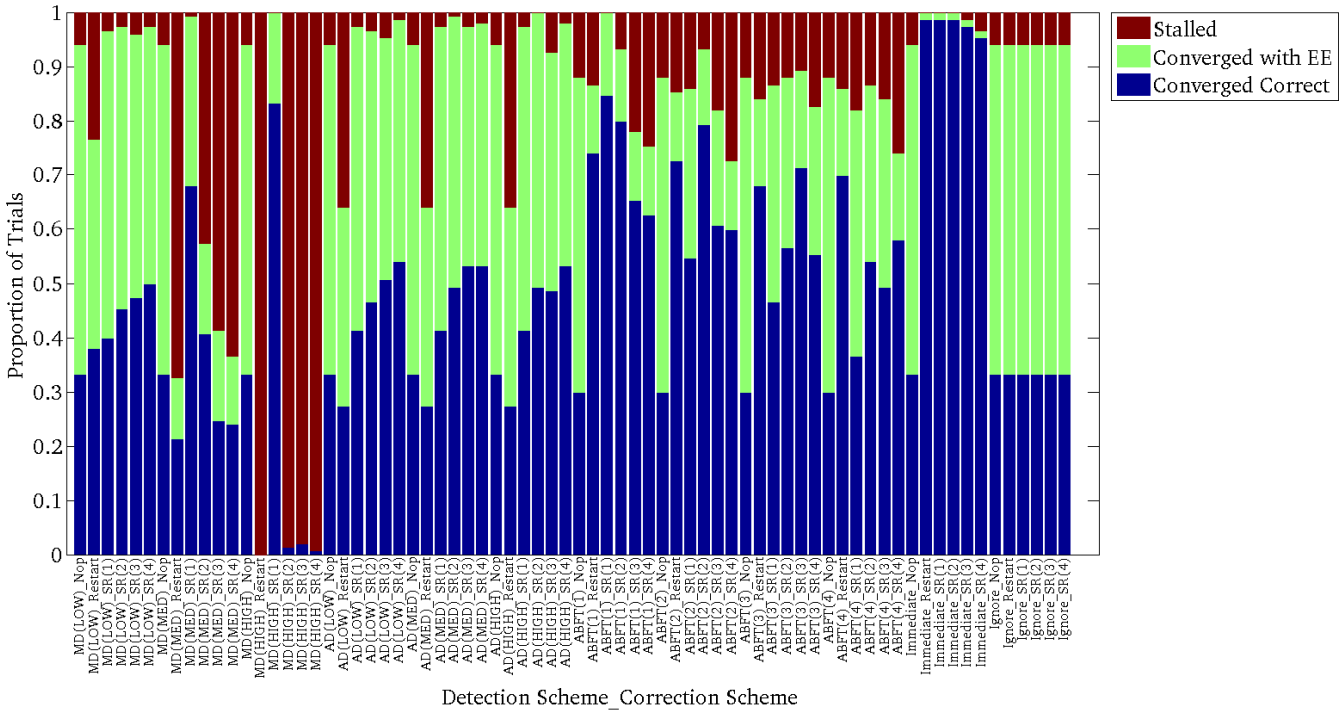

Figure 5. Residual-based methods tend to undergo a large proportion of either escaped errors or stalled trials, depending on sensitivity. Algorithm-based methods, in contrast, can provide both a small number of escaped errors and a high likelihood of convergence. In addition, EE can be observably decreased by increasing the frequency of algorithm-based checks.

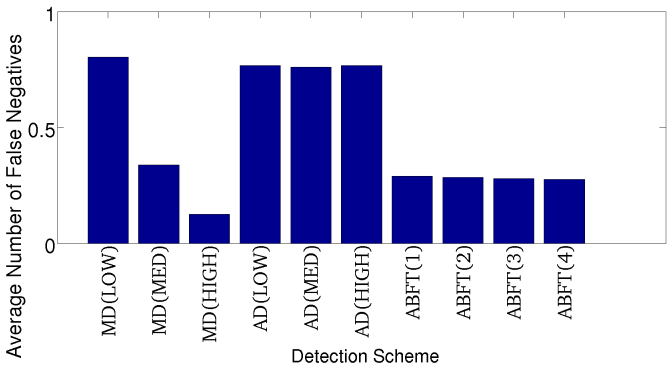

Figure 6. Average number of false negatives per trial. For the residual-based detectors, the strong inverse relationship between false negatives detected and sensitivity suggests that residual-based detectors have poor precision relative to recall. For Algorithm-based detectors, the imperfect recall suggests a dependence on the tolerance parameter.

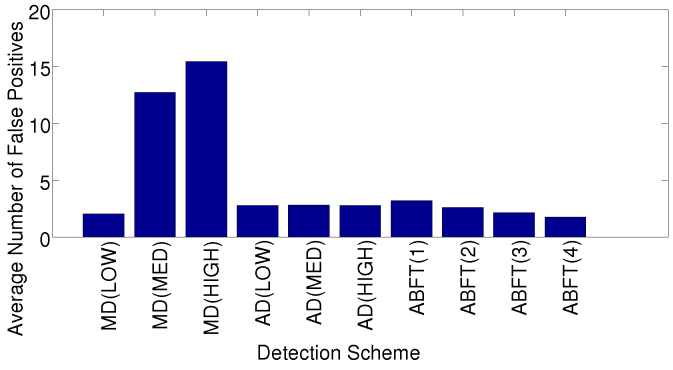

Figure 7. Average number of false positives per trial. For the residual-based detectors, the strong inverse relationship between false negatives detected and sensitivity suggests that residual-based detectors have poor precision relative to recall. For Algorithm-based detectors, the imperfect recall suggests a dependence on the tolerance parameter.

would expect a false positive at at least every other iteration.

In our primary experiment, we kept the tolerance parameter constant at a value equal to the tolerance for the stopping condition—that is, $\|r\| < 1e-6$. However, we can adjust the sensitivity of algorithm-based detection by adjusting the tolerance parameter. If we decrease tolerance of algorithm-based detection, we have fewer false negatives. If we increase tolerance, we have fewer false positives.

Figure 9 shows average number of false positives and false negatives for Algorithm-based detection with different tolerances. In this case, we require only 3 false positives to decrease expected false negatives below 5%. This works out to an expected false positive about every 0.18 iterations, which is still high, but significantly improved over residual-based methods. We see that, until we reach machine precision, we can decrease false positives while incurring relatively few additional false negatives. This tradeoff is much more favorable than for residual-based methods. It

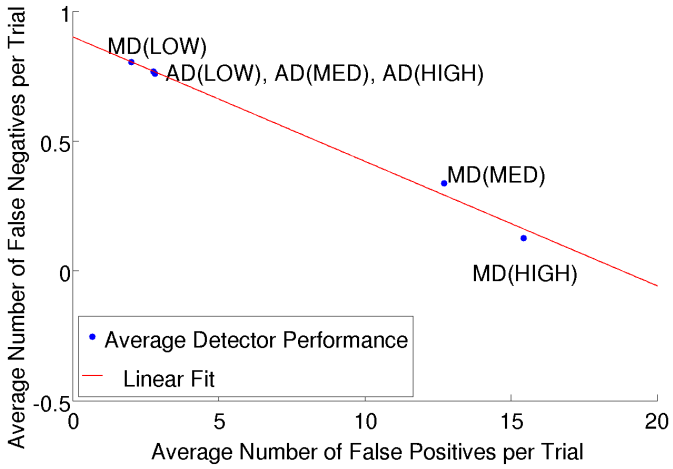

Figure 8. In the case of residual-based methods, any gain in recall comes with a high cost in precision. Each point represents the performance of a detection scheme. The cluster of points that nearly overlap are the various AD schemes. Shown with a linear fit (R^2=0.9918).

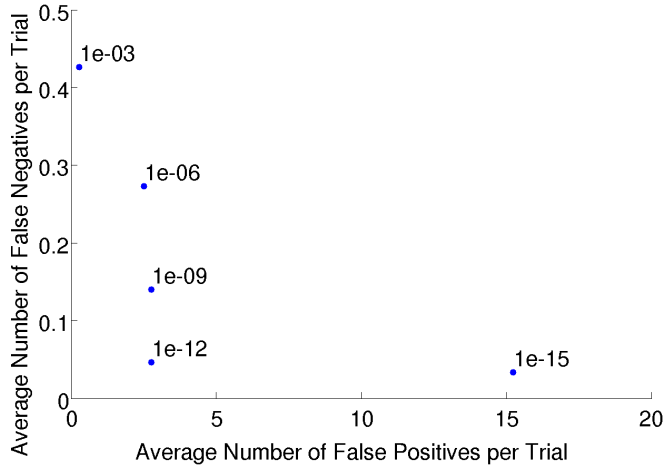

Figure 9. Until machine precision is reached, algorithm-based detection methods can decrease false negatives by reducing tolerance without severely increasing the number of false positives.

appears that we can make the event of a false negative relatively unlikely while incurring a relatively small amount of overhead from false positives by using an algorithm-based method.

3) Runtime Overhead: Figure 10 shows the average runtime overhead for each detector configuration. We compare the overhead when an error is not injected (blue) and when an error is injected (red). All overhead is normalized in terms of the time required to run the solver in an error-free environment with no detector or corrector augmentation (that is, the Ignore_Nop scheme). Runtime results for each matrix and injection iteration are normalized against the results for error-free Ignore_Nop for that particular matrix. The results for all (matrix, injection iteration, injection vector) combinations are averaged together for each detection/correction scheme.

It is important to note that this graph does not include stalled trials. If a scheme frequently stalls (like MD(HIGH)_SR(2)), then it will contribute only a small number of data points to the graph.

Pure detection overhead is captured in subset of error-free (blue) series that utilize the Nop correction scheme. For example, we see that the residual-based detectors, like MD(LOW)_Nop or AD(MED)_Nop, incur negligible overhead. For Algorithm-based detectors, like ABFT(1)_Nop, overhead is more substantial, but still smaller than other costs. For ABFT(1)_Nop, overhead is about 25%, and it decreases in inverse proportion to detection frequency.

Pure snapshotting overhead is captured in the subset of error-free (blue) series that utilize the Ignore detection scheme. We see that, for Ignore_SR(1), overhead approaches 400%, while for Ignore_SR(4), overhead is 70.5%.

Two general observations:

First, from the dramatic difference between overheads of variations of MD, its clear that false positives increase runtime. Even at our lowest snapshot frequency, we reach almost 600% overhead due to false positives with our most sensitive MD.

Second, from the relative similarity between algorithm-based detector frequencies, its clear that algorithm-based detection does not feature prominently in runtime. For ABFT(1)—our most aggressive scheme—checks only account for about 25% overhead over snapshotting. Since we have established in our discussion of outcomes that increasing algorithm-based detection frequency can increase the proportion of trials that converge correctly, it appears that aggressive Algorithm-based detection is likely to work to the benefit of the application while incurring a relatively small increase in overhead.

Our experiments inflate snapshot overhead by requiring very high snapshot frequencies to match the short solver runs. In an actual deployment we would expect much longer snapshot intervals, and therefore correspondingly lower overheads.

A. GVR Evaluation

Although we utilized only a few features of GVR (take periodic snapshots and restore), we found it easy to manage temporal redundancy in the application on a per-data-structure basis. Quantitatively, we modified on 2465 lines of code, in an application the exercise 390512 lines of code. This amounts to less than %1 of the total lines of code.

V. Discussion and Related Work

We first discuss work dealing with intermittent errors with linear systems in particular. Subsequently we discuss broader approaches.

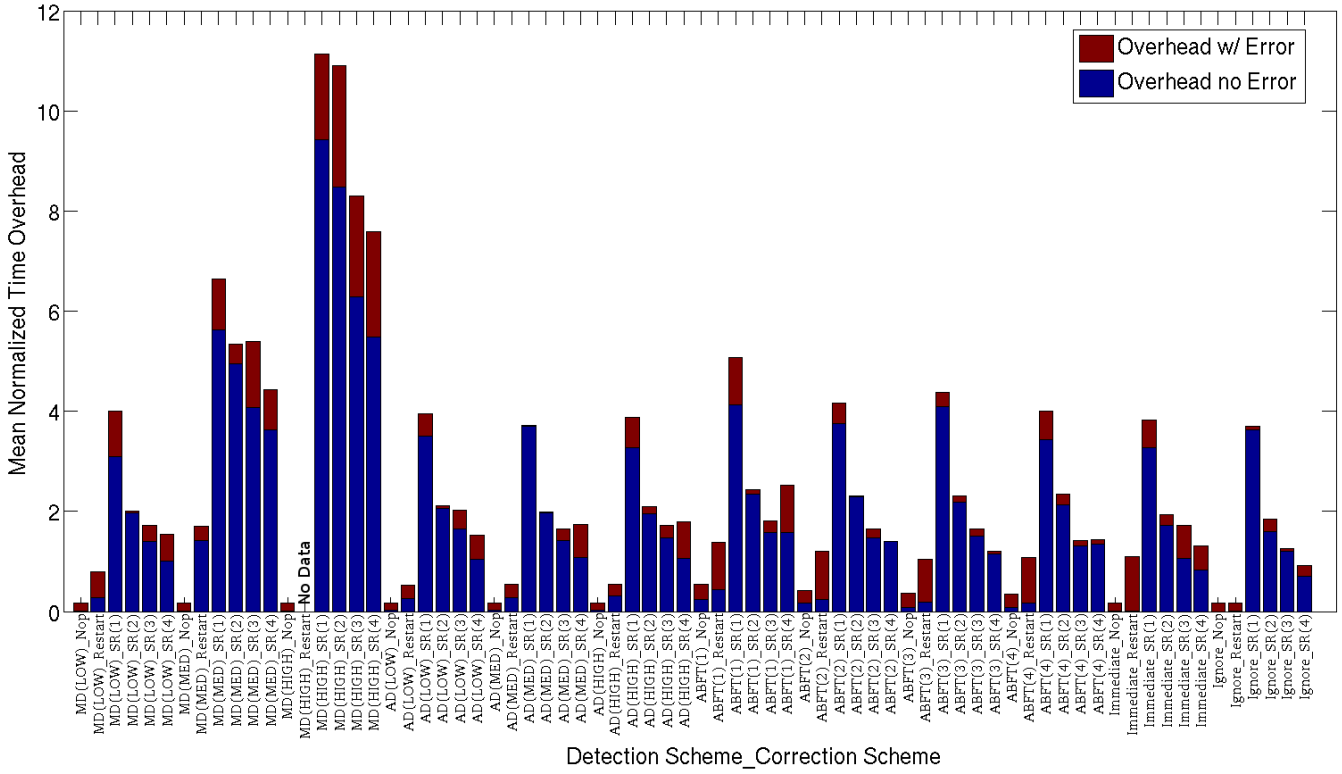

Figure 10. Overhead vs. an error-free, baseline PCG solver each respective matrix. The blue bars represent versioning, detection overhead, and false positives. The red bars represent real error recovery. Trials that produced incorrect answers or stalled are omitted.

Fault-tolerance specific to linear solvers may draw from algorithm-based fault-tolerance [16] (ABFT) which encodes parity into linear algebra operations. This extra parity can potentially be costly, but recent work [28], [30]–[32] explores different methods to make ABFT less costly. The GVR model can include ABFT at the application-programmer's discretion, although this particular study attempts to avoid the overhead of ABFT by employing residual-based error-detection methods.

It is worth noting that Chen [32] prescribes a recovery method that avoids preserving r, as ours does. Consequently, there appears to be further potential to reduce the runtime overhead from snapshotting when using algorithm-based detection.

Li et al. [33] designed a method for algorithm-based fault-tolerance in a number of linear algebra applications that takes advantage of integration with the underlying system in order to decrease the overhead for ABFT. If the DRAM can report on data corruption detected by ECC, then there is less linear algebra required to verify the correctness of state. This work presents a promising way to lower overhead of ABFT given underlying hardware that cooperates with the application in fault-tolerance. We see this sort of work as an example of the potential of clever algorithm-based tolerance that cooperates with the underlying system.

Several studies examine the stability of iterative linear systems, because of their importance to a broad class of scientific and engineering computation.

Bronevetsky and de Supinski [27] evaluated the soft error vulnerability for a number of linear solvers given a scheme of random error injection. The paper also evaluated the effectiveness of error detection and checkpointing techniques, including the error detection techniques used in our study. Bronevetsky and de Supinski's study suggested that there were specific combinations of detection methods and correction methods that could reduce the probability of silent data corruption while still introducing only a small amount of overhead. In contrast to Bronevetsky, our study incorporates results from [25], and focuses exclusively on the most vulnerable cell in each vector.

This paper also draws upon Shantharam et. al [25], which demonstrated that a single soft error in the solution vector of an iterative method will propagate to every entry in the vector within a small, bounded number of iterations, depending on the adjacency matrix involved. This paper further demonstrated that the consequences of the induced error are proportional to the L2 norm of the corresponding row of the matrix. Finally, the paper presented results suggesting that stationary methods (particularly SOR) are more tolerant to single soft errors than non-stationary methods

(particularly PCG). Our paper further examines the sorts of error-injection experiments that Shantharam performed, as well as examining times of injection, error-detection methods, and error-recovery methods.

A classical approach to dealing with errors is to use redundant hardware [34]. Although the increased hardware cost is generally a concern, some recent work [35] argues that modular redundancy may be viable in some cases. The global view resilience model focuses on redundancy in time, but does not preclude hardware or process redundancy.

A large body of previous work deals with temporal redundancy—usually by way of an optimization of checkpoint/restart. Ways to optimize checkpoint-restart include choosing optimal checkpoint intervals [36], taking more or less expensive checkpoints at different times during computation [37], [38], and modifying hybrid checkpointing to make it resilient [39]. Checkpoint/restart is similar to GVR's snapshot/restore scheme, except GVR integrates version capture and restore into the application programming model.

One trend in dealing with generally faulty systems is to distinguish between instructions that must be executed error-free and instructions that can tolerate errors [40]–[42]. Similarly, GVR allows different data structures to be more or less rigorously preserved, examined, and recovered.

VI. Summary and Future Work

We used GVR to explore application-driven error detection and correction schemes in a PCG solver. We found that, for residual-based detection methods, the number of false positives required to reduce false negatives is probably unacceptable, while for algorithm-based methods, the tradeoff is potentially acceptable. In addition, the cost of algorithm-based detection is small compared to the cost of taking periodic snapshots, so the usage of aggressive algorithm-based detection, along with sparing use of periodic snapshots, present a viable strategy for utilizing a Krylov subspace solver in a fault-tolerant environment.

This study focused on relatively simple correction schemes. We feel there is potential to reduce overhead by catering detection and correction schemes to different data structures on a per-structure basis. For example, we believe that a corruption of the p vector is less likely to cause EE than a corruption in the x or r vectors. Consequently, we may be able to maintain acceptable rates of correct convergence while decreasing overhead if we take snapshots of p more often than we take snapshots of x and r. Further, we did not consider corruption in the matrix A or the preconditioner matrix M. Both of these structures must remain unchanged through the duration of the solve, which implies the optimality of still other detection and correction schemes.

Further exploration with more complex, per-data-structure schemes will likely give rise to latent errors. We feel it will also be important to explore the implications of taking multiple snapshots of certain data structures—so that each snapshot corresponds to that structure's state at a different point in time—in order to combat the problem of latent errors [14].

Finally, our PCG solver should be able to easily scale to large clusters. There is good opportunity to explore how resilient linear solvers behave at scale.

Acknowledgments

This work supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, under Award DE-SC0008603 and Contract DE-AC02-06CH11357.

We acknowledge the authors of Shantharam et al. [25] for providing us with baseline data and a baseline implementation for our own research.

We acknowledge Mark Hoemmen for his help with implementing a PCG solver in Trilinos.

References

[1] 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," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 111–122, 2012.

[2] 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*. ACM, 2013, p. 22.

[3] C. Constantinescu, "Trends and challenges in vlsi circuit reliability," *Micro, IEEE*, vol. 23, no. 4, pp. 14–19, 2003.

[4] Peter Kogge, et. al., "Exascale computing study: Technology challenges in achieving an exascale systems," 2008, dARPA IPTO Study Report for William Harrod, available from http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf.

[5] e. a. Vivek Sarkar, "Exascale software study: Software challenges in extreme scale systems," 2009, dARPA IPTO Study Report for William Harrod, available from http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/ECSS%20report%20101909.pdf.

[6] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science–VECPAR 2010*. Springer, 2011, pp. 1–25.

[7] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir, "Toward exascale resilience," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 374–388, 2009.

[8] 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 (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[9] R. Geist and K. S. Trivedi, "Reliability estimation of fault-tolerant systems: Tools and techniques," *Computer*, vol. 23, no. 7, pp. 52–61, 1990.

[10] A. Geist, "A paradigm shift is coming-continuous failure," in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*. IEEE, 2012, pp. 371–371.

[11] M. Elnozahy, "System resilience at extreme scale: A white paper," 2009, dARPA Resilience Report for ITO, William Harrod.

[12] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[13] M. Hoemmen and M. Heroux, "Fault-tolerant iterative methods via selective reliability," in *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society*, 2011.

[14] G. Lu, Z. Zheng, and A. A. Chien, "When is multi-version checkpointing needed?" in *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale*. ACM, 2013, pp. 49–56.

[15] X. Ni, E. Meneses, N. Jain, and L. V. Kalé, "Acr: Automatic checkpoint/restart for soft and hard error protection," in *Proceedings of 2013 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[16] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *Computers, IEEE Transactions on*, vol. 100, no. 6, pp. 518–528, 1984.

[17] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

[18] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.

[19] "Global view resilience (gvr)," https://sites.google.com/site/uchicagolssg/lssg/research/gvr, accessed: 2013-12-07.

[20] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps *et al.*, "An overview of the trilinos project," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 397–423, 2005.

[21] D. S. Kershaw, "The incomplete choleskyconjugate gradient method for the iterative solution of systems of linear equations," *Journal of Computational Physics*, vol. 26, no. 1, pp. 43–65, 1978.

[22] S. Hogan, J. Hammond, and A. A. Chien, "An evaluation of difference and threshold techniques for efficient checkpointing," in *2nd Workshop on Fault-Tolerance at Extreme Scale FTXS 2012 at DSN 2012*, June 2012.

[23] H. Fujita, R. Schreiber, and A. A. Chien, "It's time for new programming models for unreliable hardware," in *ASPLOS 2013 Provocative Ideas session*, March 2013.

[24] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[25] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Characterizing the impact of soft errors on iterative methods in scientific computing," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 152–161.

[26] J. Elliott, F. Mueller, M. Stoyanov, and C. Webster, "Quantifying the impact of single bit flips on floating point arithmetic," Dept. of Computer Science, North Carolina State University, Tech. Rep. TR 2013-2, March 2013.

[27] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 155–164.

[28] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 73–84.

[29] "Trilinos," http://trilinos.org/, accessed: 2013-12-07.

[30] J. Sloan, R. Kumar, and G. Bronevetsky, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *Proceedings of The 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2013.

[31] M. Shantharam, S. Srinivasmurthy, and P. Raghavan, "Fault tolerant preconditioned conjugate gradient for sparse linear system solution," in *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 2012, pp. 69–78.

[32] Z. Chen, "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*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 167–176. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442533

[33] D. Li, C. Zizhong, W. Panruo, and S. Vetter Jeffrey, "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC) Log in to post comments Google Scholar BibTex Tagged XML RIS Archives News Archive Colloquium Archive Conference Archive ORNL— Computing and Computational Sciences Directorate— Computer Science and Mathematics Division— ORNL Disclaimer*, 2013.

[34] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[35] K. Ferreira, J. Stearley, J. H. Laros III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, "Evaluating the viability of process replication reliability for exascale systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 44.

[36] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.

[37] A. Moody, G. Bronevetsky, K. Mohror, and B. de Supinski, "Detailed modeling, design, and evaluation of a scalable multi-level checkpointing system," *Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-440491*, 2010.

[38] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "Fti: high performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 32.

[39] A. Guermouche, A. Schiper, F. Cappello, T. Martsinkevich, T. Ropars *et al.*, "Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC" 13)*, no. EPFL-CONF-189836, 2013.

[40] P. G. Bridges, M. Hoemmen, K. B. Ferreira, M. A. Heroux, P. Soltero, and R. Brightwell, "Cooperative application/os dram fault recovery," in *Euro-Par 2011: Parallel Processing Workshops*. Springer, 2012, pp. 241–250.

[41] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*. ACM, 2011, pp. 164–174.

[42] Y. Saad, "A flexible inner-outer preconditioned gmres algorithm," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 461–469, 1993.