Algorithmic Approaches to Low Overhead Fault Detection for Sparse Linear Algebra

Joseph Sloan, Rakesh Kumar
University of Illinois,
Urbana-Champaign
jsloan,rakeshk@illinois.edu

Greg Bronevetsky
Lawrence Livermore National Laboratory,
Livermore,CA
bronevetsky@llnl.gov

***Abstract*—The increasing size and complexity of High-Performance Computing systems is making it increasingly likely that individual circuits will produce erroneous results, especially when operated in a low energy mode. Previous techniques for Algorithm - Based Fault Tolerance (ABFT) [20] have been proposed for detecting errors in dense linear operations, but have high overhead in the context of sparse problems. In this paper, we propose a set of algorithmic techniques that minimize the overhead of fault detection for sparse problems. The techniques are based on two insights. First, many sparse problems are well structured (e.g. diagonal, banded diagonal, block diagonal), which allows for sampling techniques to produce good approximations of the checks used for fault detection. These approximate checks may be acceptable for many sparse linear algebra applications. Second, many linear applications have enough reuse that pre-conditioning techniques can be used to make these applications more amenable to low-cost algorithmic checks. The proposed techniques are shown to yield up to $2x$ reductions in performance overhead over traditional ABFT checks for a spectrum of sparse problems. A case study using common linear solvers further illustrates the benefits of the proposed algorithmic techniques.**

***Index Terms*—ABFT, sparse linear algebra, numerical methods, error detection**

I. Introduction

As High-Performance Computing (HPC) systems grow more capable, they also grow larger and more complex. This means that as the number of components in the systems rises, so does the probability that one of them will suffer from a fault. Soft faults in chip circuitry are among the most worrying for system designers and application developers because they can corrupt the application's computations and produce incorrect output. Tera-scale systems are already vulnerable to soft errors, with ASCI Q experiencing 26.1 CPU failures per week [18] and a L1 cache soft error occurring about once every five hours on the 104K node BlueGene/L system at Lawrence Livermore National Laboratory [8]. Looking into the future, according to the International Technology Roadmap for Semiconductors, the soft error rates (SER) will grow with smaller chip sizes, with SRAM SER growing exponentially with chip size [1]. This and the fact that Exascale systems in 2020 will have a total of 4 million electronic chips with feature sizes as low as 12nm [1] has led the DARPA Exascale Computing study [6] to warn that "traditional resiliency solutions will not be sufficient". Hardware-based approaches for fault detection have been proposed for many computing systems. However, their reliance on redundancy makes them impractical for future HPC systems which will be increasingly power-constrained. In fact, evolutionary extensions of today's high performance computing (HPC) systems (CrayXT, BlueGene) will be unable to reach exaFLOP performance by 2020 within a power budget of 20MW, the typical limit of modern computing centers [6]. As such, fault detection for exascale systems will need to increasingly rely on software or algorithmic approaches, a fact that motivated the Exascale study to identify "Algorithmic-level Fault Checking and Fault Resiliency" as a key research thrust. This paper focuses on algorithmic low overhead fault detection for sparse linear algebra applications. Sparse linear algebra forms the core of a large class of high performance computing (HPC) applications such as linear solvers, differential equation solvers, and graph analysis problems [11, 19]. It also forms the core of a large number of emerging recognition, mining, and synthesis (RMS) applications [5]. Algorithmic approaches to fault detection for sparse linear algebra will eliminate the need for high overhead hardware approaches to fault detection for exascale systems and systems running RMS applications.

Our algorithmic approach builds upon ABFT-based approaches that encode computations using linear error correcting codes [20, 2]. Such approaches have been proposed previously for dense linear algebra [20]. Unfortunately, these traditional ABFT approaches cannot be used directly for sparse linear algebra problems as sparse linear algebra problems have lower algorithmic time complexity than equivalent dense problems. A direct use of the previously proposed approaches can result in high overheads for sparse linear algebra problems (Section V). In this paper, we propose algorithmic optimizations focused on low overhead checksum-based fault detection for sparse linear algebra-based applications. Our fault detection techniques rely on two insights. First, many sparse applications have inherent structure within the data and computation (e.g. diagonal, banded diagonal, block diagonal). These structures may be exploited to improve the performance of traditional ABFT checks (dense checks) by checking a representative, randomly sampled, subset of the computation at the cost of a minor reduction in fault coverage. Second, linear applications have significant reuse. This makes it possible to precondition the linear problem to be more amenable to low cost algorithmic checks (Section III).

This paper focuses on fault detection for sparse matrix-vector multiplication (MVM), the most common operation in sparse linear algebra. We make the following contributions:

- We demonstrate that previous algorithm-based techniques for fault detection in MVM are expensive for sparse matrices.
- We observe that enough structure exists in many sparse problems that simple sampling techniques provide almost the same coverage as exact techniques with reduced performance overhead. We present two sampling techniques:
  - Approximate Random (AR) checking - randomly samples the problem
  - Approximate Clustered (AC) checking - samples based on the problem's structure
- We show that there exists sufficient reuse for many sparse linear applications that performance can be improved by preconditioning the problem to reduce the cost of detection for similar coverage, in spite of increased setup cost. We propose two preconditioning techniques:
  - Identity Conditioning (IC) - computes a code that creates additional structure for a given problem
  - Null Conditioning (NC) - creates structure by finding a code which lies in the null space of the sparse problem
- We quantify the benefits of the proposed techniques in the context of MVM itself and as a subroutine of linear solvers. For MVM, the dense checks are shown to have high overheads (up to 100%, 32% on average) for sparse problems. The proposed sparse techniques are shown to reduce the detection overhead by up to $2x$ (average overhead is 17%) for the same fault detection accuracy. Our linear solver implementations with sparse techniques are 20% faster than the corresponding implementations using traditional ABFT (dense checks).

The paper is organized as follows. Section II describes related work and explains the limitations of prior checksum-based techniques when applied to sparse linear algebra problems. Section III describes the opportunities for exploiting the structure of sparse linear problems for reducing fault detection overhead and introduces our approach. Section IV discusses the methodology for evaluating the effectiveness of the techniques. Section V presents the results. SectionVI concludes.

II. Related Work

There exists a large body of work on different aspects of sparse linear algebra-based applications. In particular iterative solvers for sparse linear systems are an important tool for scientific computing and research [26, 9, 28, 19]. Common examples include conjugate gradient and multigrid solvers. While the majority of the research on sparse linear algebra addresses parallelization and performance, this paper focuses on making them resilient to soft faults.

There has been prior work on checksum-based algorithmic approaches to fault-tolerance of linear algebra-based applications. Algorithm-Based Fault Tolerance (ABFT) [20] was proposed to detect and correct errors in matrix multiplication operations. It has more recently been generalized [2] and extended to more general linear algebra algorithms [23, 25, 3] such as the multi-grid solver [24] as well as to multiprocessors [3].

The traditional ABFT check works by encoding a linear operation using linear error correcting codes. For example, the check for MVM (Ax: matrix A, vector x) works by verifying the identity:

$$c^T(Ax) = (c^T A)x$$

Intuitively, the check computes the projection of the result Ax onto the vector c in two different ways. If there are any computation errors, the two projections will very likely be unequal (e.g. the difference between projections surpasses a given threshold, τ)

In the common case where $c = \bar{1}$ (a vector of all 1's), the projection is equivalent to multiplying x by the vector containing the sums of matrix A's columns. Because the dense check focuses on the projection of the result onto a specific vector, it requires only three operations: (i) a matrix-vector product that must be done for each matrix A and (ii) two dot-products that must be performed on every MVM. This check is therefore very efficient for dense matrices, requiring $O(n^2)$ setup time and $O(n)$ time for each MVM operation, compared to the $O(n^2)$ time required for the original multiplication. However, the check becomes very expensive for sparse matrices, where MVM takes only $O(n)$ time, meaning that both the original operation and the check have equal asymptotic complexity. In this paper, we address this problem by exploiting the properties of sparse linear algebra applications to reduce the constant factor of the the ABFT check, making it significantly cheaper than the original operation. Indeed, to the best of our knowledge, it is the first work to address application-level fault tolerance in the context of general sparse linear algebra.

III. Algorithmic Fault Detection

In this section, we discuss two opportunities for sparse linear algebra applications that can be exploited to reduce the overhead of algorithmic fault detection for such applications. We then describe four techniques that exploit these opportunities.

A. Motivation

Sparse problems frequently have well defined structures. Common examples of structure are diagonal, banded diagonal, and block diagonal matrices. For example, *qpband* (Figure 1), which represents a canonical indefinite optimization problem, illustrates a typical banded diagonal structure (the nonzero pattern is on the left). Similarly, the matrix *bcsstm37* (Figure 3), which represents a track ball stiffness matrix [4] and the matrix *msc00726* (Figure 2), representing a structural engineering problems from the Boeing test matrix group [4], also contain banded diagonal type structures. The matrix *Oregon-1* (Figure 4), representing an undirected graph based on the network included in a portion of the Internet, shows a block diagonal type structure.

Such structures in sparse problems commonly translate into fairly uniform distributions of the column sums. Since, as described in Section II, the traditional check for $c = \bar{1}$ is

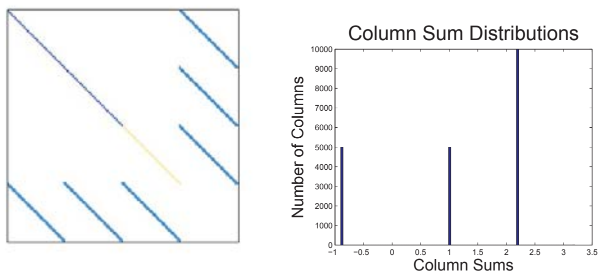

Fig. 1. qpband (Variance $= 1.6071$) The matrix has a well defined and low variance ($< 1e3$) column sum distribution and is a good candidate for both Approximate Random and Approximate Clustering.

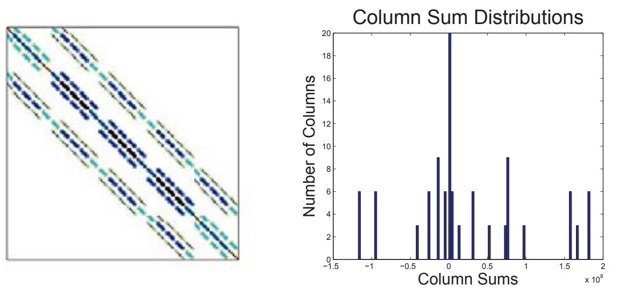

Fig. 2. msc00726 (Variance $= 9.4724e14$) The matrix has high variance ($> 1e3$) column sums. This matrix is a good candidate for clustering given the finite sets of unique values shown above.

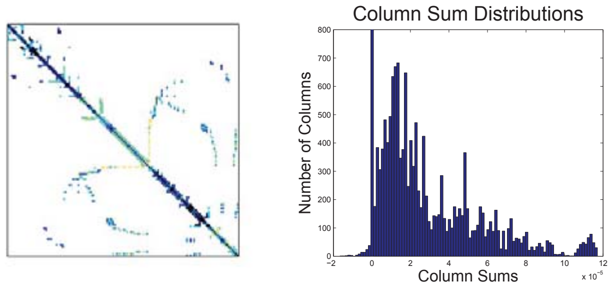

Fig. 3. bcsstm37 (Variance= $= 6.1668e-10$). The matrix has a well defined column distribution with low variance ($< 1e3$) and is particularly well suited for Approximate Random Technique

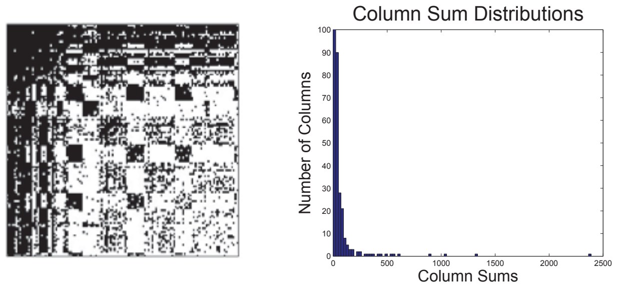

Fig. 4. Oregon-1 (Variance $= 1065.5$). The matrix has column sums that are less well defined and have high variance ($> 1e3$). Conditioning is a good candidate for this particular problem.

equivalent to multiplying x by the sums of columns, matrices with well-defined distributions of column sums present an opportunity to use a sparse check that samples only a fraction of the columns. Such a sampled check would give up a small degree of coverage (some errors may be missed) for a significant reduction in overhead. This flexibility can be very valuable in contexts where some errors can be tolerated (e.g. iterative methods that converge to accurate solutions) or where even the reduced fault coverage results in a very high mean time to failure. When the fault detection computation itself is susceptible, note that by reducing the number of operations in the check, we may also be improving the accuracy under a given set of conditions (discussed later in Section V-A).

The second opportunity for reducing algorithmic detection overhead for sparse linear algebra applications is that many such applications typically use the same matrix as part of many individual operations. For example, iterative solvers for linear systems ($Ax = b$) use MVM multiple times during each iteration, and solvers such as conjugate gradient (CG) or iterative refinement (IR) (see Section V), can take thousands of iterations to converge to an acceptable solution. This property of frequent data reuse makes it possible to analyze the structure of a given matrix or precondition the matrix to have a more favorable structure, thus amortizing the setup cost by using lower overhead checks for subsequent MVM operations.

The following algorithmic fault detection techniques exploit the above opportunities.

B. Approximate Random

The traditional check verifies the identity

$$c^T(Ax) = (c^T A)x$$

If $c = \bar{1}$, this computation is equivalent to computing the vector of A's column sums (i.e. $c^T A$) and multiplying it by x. Thus, if a matrix's column sums have a relatively simple structure, it may be possible to perform the check over a randomly sampled subset of columns. For example, the variance in column sums is only 1.6 in matrix *Qpband* (Figure 1) and $6e-10$ in *bcsstm37* (Figure 3). This technique, called Approximate Random (AR), works by setting c to be a binary vector, with 1's in a some random locations and 0's everywhere else. The detection overhead is therefore reduced by avoiding computations associated with dimensions of the check containing 0's. It is further refined by observing that the primary cost of the check is the sparse matrix-vector product on the right-hand side, while the left-hand side is a dense dot-product, which is faster due to its much better memory behavior. Moreover, sampling the left-hand side will cause the check to incur a more significant loss in fault coverage since the left-hand side is a function of the faulty MVM output directly. As such, this check only uses the sampled c on the right-hand side, uses $c = \bar{1}$ on the left-hand side and then normalizes the left-hand side to adjust for the difference.

The accuracy of AR depends on the distribution of the values in x, in addition to depending on the distribution of the matrix columns. In the context of computational science, x typically corresponds to the state of a physical system. Since different regions of the physical space will have similar states, x will have a regular structure, which enables AR to work well. However, in cases where the physical system is chaotic (high variance) or x comes from a non-physical system, the sampling technique may need to take the distribution of x into account as well (e.g. a scaling factor related to the input variance).

C. Approximate Clustering

For matrices with more variable column sums that still have some structure it is possible to improve the quality of the sampled columns by clustering their sums. Instead of computing c by randomly sampling $\bar{1}$, Approximate Clustering (AC) runs a clustering algorithm on the set of column sums and randomly samples the clusters to ensure that all the major types of column sums are appropriately represented. This approach trades off additional setup overhead for improved accuracy and is thus applicable for applications with more reuse. Further, it works with a broader range of matrices than AR since it only requires column sums to be homogeneous within local regions, rather than globally similar.

Matrix *msc00726* (Figure 2) is an example of a structured problem (it is banded diagonal) that is too complex for AR because it contains one set of values that are orders of magnitude larger than another set of values. However, because this matrix has only a small set of unique column sums (20 unique values or about 1% of columns are unique), clustering can be used to identify these different classes of columns and evenly sample among these classes. As such, even though the structure of a problem may not be well suited for random sampling, there may exist enough structure that can still be exploited by using clustering to represent each subgroup of similar values equally in the sampled distribution.

A variety of methods, with varying complexities, can be used to cluster the column sums. We use an agglomerative clustering algorithm [21], modified to ensure that (i) only those clusters are formed which are sufficiently different from other values (i.e. a distance threshold of $1e-6$) and (ii) the number of clusters identified do not exceed the total number of samples dictated by the sampling rate specified to the algorithm. This algorithm is run over the entire set of column sums.

In some cases, the sparse problem may not be amenable to either AR or AC, such as the matrix *Oregon-1* (Figure 4). *Oregon-1* has high column sum variance ($> 1e3$) and a large dissimilarity between a majority of its' sums. Therefore, the efficiency of AR and AC for this problem will be limited. We may need to first precondition the problem in order to yield a more uniform distribution of column sums.

D. Identity Conditioning

In situations where sparse linear algebra applications have reuse, but also have matrices that are not directly amenable to sampling or clustering (i.e. the column sums are too variable to produce an accurate check), preconditioning can be used to transform the check into a form more amenable for low overhead algorithmic fault detection.

Identity conditioning (IC) transforms the high variance column sum distribution of the original matrix (A) into a more uniform set of values by using a check vector tailored to the given problem, instead of the traditional checksum: $c = \bar{1}$. IC finds such a tailored check vector by solving the system:

$$c^T A = \bar{1}^T \qquad \text{(identity equation)}$$

When the identity equation is solved exactly, the effect of A and the variance of the column sums is eliminated entirely:

$$c^T y = (c^T A)x = \bar{1}^T x = \sum x \qquad \text{(IC)}$$

This makes the problem directly amenable to low-cost sampling as the variance in A now has a smaller effect on the product $c^T A$, making the sampling in AR and AC more representative than when sampling the check vector $c = \bar{1}$. We denote as ICAR the algorithm that preconditions the problem with IC and checks individual MVM operations using AR. Similarly, ICAC is the combination of IC and AC. Also, in many scenarios, the sum of the elements of x may be known or can be inferred in advance (e.g. when $\sum x$ is used to check prior linear operations). In such scenarios, the runtime overhead of the check may be reduced significantly.

While IC can be highly effective, it has two major limitations. First, the exact solution to the identity equation does not exist for all matrices (e.g. symmetric or overdetermined matrices). Second, computing this equation can be very expensive. We resolve these issues by computing the equation approximately, solving $min\|A^T c - \bar{1}\|$ with a relaxed accuracy tolerance. In practice we find that that the iterative least squares algorithm in LAPACK [7] provides a good approximation of c (residual < 100) in $1-2$ iterations. Still, given that computing this approximation is equivalent to multiple MVMs, it is primarily useful for applications that reuse the same matrix in many MVMs, amortizing the cost of computing the conditioned vector c over many checks.

E. Null Conditioning

While Identity Conditioning eliminates the influence of A on the check, additional conditioning can also eliminate the influence of x. The *Null Conditioning* (NC) algorithm finds a check vector in the null space of the matrix A, solving the equation

$$c^T y = (c^T A)x = 0 \qquad \text{(NC)}$$

This significantly reduces the runtime overhead of the check, since the right side of the check requires no additional computation (e.g. the sum equals zero) and the memory locality is improved since the input is no longer read in the check.

Finding a vector in (or near) the null space of A is done by computing its smallest singular value using singular value decomposition (SVD). A singular value (σ) of a problem satisfies:

$$Ac = \sigma u \text{ and } A^* u = \sigma c$$

where u and c are unit length vectors. The singular vector associated with the smallest singular value is used in the check as: $c^T A \approx 0$.

The accuracy of fault detection for NC depends on the size of the problem's smallest singular value. In our experiments, singular values below 1e-6 are more than sufficient to provide high accuracy fault detection. However, problems with larger singular values may still use the associated singular vector to improve the efficiency of the sampling based techniques, since the conditioned column sums ($c^T A$) still contain fairly uniform distributions. We observe that the conditioned column sums slowly become less uniform as the size of the singular values increase (e.g. variance is less than1e-4 as the singular values increase up to 1-100).We call the strategy of initially using NC to precondition a sparse problem, and AR during the runtime, NCAR. Similarly, the strategy of using AC during the runtime, NCAC.

Note that this check can be used for dense algebra problems as well. For square symmetric matrices, which are common in practice, SVD reduces to the eigenvalue problem. SVD/Eigen decomposition can be implemented in various ways depending on properties of the data and goals. In particular, our SVD implementation does not need to compute all the singular values , but instead only computes the smallest singular value and the associated vector [16]. This feature, along with the possibility of relaxing the tolerance of the required singular value, allows us to reduce the overall cost of computing the complete SVD by several factors. I.e. by finding only the smallest singular value and reducing the input and output tolerance by 3-5 orders of magnitude, there was a $2X$ to $10X$ speedup in execution time, with a minimal increase in variance of the column sums.

IV. Methodology

A. Fault Model

Our evaluation focuses on transient faults that affect the outputs of numerical computations. Other manifestations of transient faults, such as memory corruption, deviations of control flow or memory access errors are assumed to be accounted for by using simple low overhead techniques [12, 14], unless they manifest as numerical data errors which the proposed techniques cover. This is a widely addressed fault model [20, 15, 10].

Our evaluations are done for MVM. MVM is composed of a series of multiply and accumulate operations. Faults are injected into MVM by adding a random numeric error to the output of individual multiply/addition operations. Since the timing of each fault is assumed to be independent, fault times are sampled from an exponential distribution with a rate λ. $\frac{1}{\lambda}$ is the expected the number of arithmetic operations between consecutive faults. Faults are also similarly injected into the arithmetic operations that compose the checks themselves.

Our experiments examine different fault rates that model phenomena ranging from physical faults arising from infrequent particle strikes (3-4 soft errors per day) to frequent errors arising from the use of aggressively designed (error-prone) technologies at large scales (multiple errors per second).

When a fault occurs, it is modeled by drawing a value from one of the fault distributions below and adding it to the target operation. These distributions are selected to model the arithmetic effects of circuit-level faults at a high level, making it possible to parameterize them to represent multiple low-level fault models:

Symmetric Faults: The following distributions model faults that affect the output of circuits, and that have equal probability of being positive or negative

- 1: Distribution with two Gaussian modes. The modes are centered at $1e5$ and $-1e5$ and have variance $1e2$;
- 2: Distribution with two Gaussian modes centered at $\pm 1e10$, each with variance $1e5$;
- 3: Gaussian distribution with mean 0 and variance 100.

Memory Faults:

- 4: An exponential distribution represents a single bit flip in the binary representation of a floating point number.

Non-Symmetric Faults:

- 5: Gaussian distribution centered at $1e5$ and with variance 100 - represents a one sided error distribution (e.g. unsigned representation);
- 6: Mixture of models 1 and 2, each sampled half the time - models timing errors in functional circuit units, which are biased toward most and least significant digits [22].

B. Benchmarks

The algorithmic fault detection techniques were implemented within the SparseLib library [9] of core sparse linear algebra operations, including MVM. To understand the effectiveness of our technique in a wide range of practical contexts, we evaluated them on 100 randomly chosen square linear systems from the University of Florida Sparse Matrix Collection and Matrix Market [4, 17] with the following properties: matrix size $\in [100, 40000]$ and sparsity < 10 (i.e. number of non-zero elements in each matrix divided by size is less than 10). These represent a variety of physical phenomena and real algorithms, including model reductions, computational fluid dynamics, and circuit simulation.

We evaluate our fault detection techniques both in the context of individual MVM operations as well as larger applications that use MVM as a subroutine. The applications we have focused on in this study are sparse linear solvers since they are very common in computational science and make extensive use of MVM. We consider two linear solvers: the Iterative Refinement (IR) method and the Conjugate Gradient (CG) method.

IR, also called the Richardson Iteration, is relatively simple and easily parallelizable. In each iteration it computes the residual of the system ($b - Ax$) and adds this to current approximate solution (x_i) with a scaling factor to generate the next approximation (x_{i+1}). Convergence is achieved by carefully selecting each step's scaling factor.

CG is a popular solver well suited for very large and sparse problems. It expresses x as a linear function of n vectors $p_1, p_2, ...p_n$, with each pair of vectors conjugate in A ($p_i A p_j = 0$). Although the p_i's can be computed directly, in practice a small subset of the p_i's is needed to achieve accuracy within machine precision. As such, CG approximates the solution $x = q_1 p_1 + ... + q_n p_n$ with only a few vectors. The initial approximation is x_0; the residual $r_0 = b - Ax_0$, which is the direction of the error in x_0, serves as the first conjugate vector, p_0. Subsequent iterations compute the residual r_k and use it to compute the next conjugate vector p_k. To ensure that p_k is conjugate to prior p_i's, $p_k = r_k - \frac{r_{k-1}^T - r_{k-1}}{r_{k-2}^T r_{k-2}} p_{k-1}$.

The coefficients α_k are computed as $\frac{r_k^T r_k}{p_k^T A p_k}$. This process is repeated until r_k falls below some threshold.

Depending on the problem, preconditioning techniques (e.g. Jacobi, Incomplete Choleksy, and LU factorizations) may also be used to improve the performance of the linear solvers [9]. In Section V-B, we study both non-preconditioned solvers (CG & IR) and the preconditioned solvers (CG-pre & IR-pre, which use incomplete LU factorization and a simple diagonal preconditioner respectively [9])

Our fault detectors are applied by comparing the difference between both sides of the identity $c^T(Ax) = (c^T A)x$, If these values differ by more than a given threshold τ a fault is declared and otherwise, the results of the computation are considered valid. Since MVM operations in linear solvers are applied to vectors of different magnitudes during the course of the run, the detection threshold used in our linear solver experiments is computed as $\tau = \tau_0 \|x\|$. Here x is the algorithm's current estimate of the solution and τ_0 is a scaling factor fixed throughout the solver's execution.

C. Exploring Solver/Parameter Space

The performance of the proposed techniques can vary significantly depending on the parameters of the fault detector (e.g. detection threshold τ and sampling rate), system properties (fault model and rate) and characteristics of the sparse problem (e.g. amount of reuse and matrix structure). Table I list all the parameters that are relevant to the effectiveness of fault detection. Our experiments with MVM and linear solvers sweep this entire space. For each combination of parameters, we run MVM with 50 different input vectors, the values of which are chosen uniformly at random from the range $[-1, 1]$. Also, for each vector, we perform 50 separate fault injection runs. Thus, for each parameter configuration, we run 2,500 runs of MVM on each of the 100 example matrices, for total of over 600 million runs. Further, for each configuration we run 50 runs of each linear solver (total of over 20 million runs each).

Different techniques work best in different scenarios and indeed, not every technique is even applicable to all linear systems. For example, Null Conditioning only works for systems that have small singular values. Since it is thus critical to choose the best technique for a given scenario, we evaluate two ways to make this selection. The "Oracle" algorithm chooses the technique and combination of parameters that maximizes the effectiveness of our techniques (it may choose the dense check or any sparse check), using the evaluation metrics described in Section IV-D; it represents the optimal solution. We also evaluate a realistic solution by using our parameter sweep experiments to train a statistical classifier, the J48 decision tree, as implemented in WEKA [13]. This "Decision Tree" selects for each linear system, fault model and rate the fault detection algorithm along with its parameters. An example of a decision tree used to predict the best technique and configuration is shown in Figure 5.

MVM and Solver Parameters	Values
Techniques	Dense,AR,AC,NC,IC, ICAR,ICAC,NCAR,NCAC
Fault rates	0, 1e-6, 1e-5, 1e-4,1e-3,1e-2,1e-1
LSQ tolerance (IC)	1e-10,1e-6,1e-3,1e-1,1, 1e1,1e3,1e6,1e10
LSQ input condition num. (IC)	1e15
Eigen solver tolerance (NC)	1e-10,1e-6, 1e-1 1
Sample rate (AR,AC)	0.001, 0.01, 0.05,0.1,0.2,0.3, ... 1.0
Other Solver Parameters	**Values**
Linear Solver	CG, IR, CG-pre, IR-pre
Detection Threshold Factor(τ_0)	1e-5, .1,1,10,1e3,1e5,1e7

TABLE I
MVM AND LINEAR SOLVER PARAMETERS

D. Metrics

To evaluate the effectiveness of our detectors with MVM we compute the following metrics for all the 2,500 runs associated with a given detector configuration, system parameters and input matrix:

- The number of true positives (TP)/false negatives (FN) - experiments where a fault was injected and was detected/not detected, and
- The number of false positives(FP)/true negatives (TN) - experiments where no fault was injected and the detector did/did not signal a fault.

From these we compute the F-Score, a metric commonly used to summarize an algorithm's overall effectiveness [27], defined as:

$$\text{F-Score} = \frac{2 * TP}{2 * TP + FP + FN}$$

The F-Score ranges between 0 and 1 and values closer to 1 indicate a more accurate and useful detector. Further, we discuss detector effectiveness in terms of the false positive rate (FPR) $\frac{FP}{TP+FP}$ and true positive rates (TPR) $\frac{TP}{TP+FP}$.

In the context of linear solvers the effectiveness of our detectors is only meaningful in terms of how it affects the solver's performance. Specifically, there can be two types of issues. Mistaken error detections cause unnecessary recovery actions, which can be expensive. In our experiments we assume that the linear solver is rolled back to its starting point and re-executed on an error detection. Conversely, errors that are not detected affect the solver's convergence properties, with errors that have larger numeric magnitude having a more significant effect.

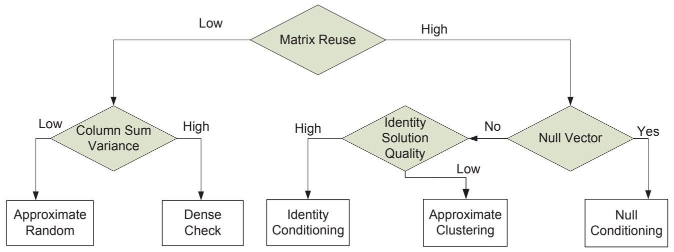

Fig. 5. Decision Tree showing the basic decisions and features when considering which techniques to utilize when protecting sparse operations.

V. RESULTS AND ANALYSIS

This section evaluates the accuracy of our detectors in the context of individual MVM operations (Section V-A) and linear solver applications with rollback-restart (Section V-B).

A. Algorithmic Fault Detection for MVM

We compare each detection technique when applied to a single MVM operation over a set 100 problems from the University of Florida Sparse Matrix Collection [4]. Our analysis includes the overhead incurred during the execution of the MVM operation and excludes the set-up cost, such as clustering and conditioning. Experiments in Section V-B, which focus on linear solvers, take set-up overhead into account and measure the degree to which this cost is amortized in practice, by repeated execution of MVM operations.

The utility of our fault detection algorithms depends on both their detection accuracy and performance overhead. Because practical uses of fault detection require that most faults are detected and most alerts correspond to real faults, our analysis focuses on configurations of the fault detectors that achieve an F-Score greater than 0.9. Figure 6 shows the overhead of all our techniques using parameters that achieve this F-Score on each matrix. The eight columns on the right-hand side correspond to each base technique, highlighting their individual capabilities. Recall that the four techniques on the far right are combinations of the others (e.g. ICAR is IC + AR, while NCAR is NC + AR). For a given technique and input problem, we choose the configuration parameters (detection threshold, sampling rate, conditioning quality) that minimize its overhead while meeting the F-Score bound. The three columns on the left-hand side correspond to the three full algorithms we're evaluating: the traditional dense check, the Oracle algorithm, and the Decision Tree algorithm. The threshold parameter of the dense check is selected optimally as above and the same is true of the Oracle, which always picks the optimal algorithm (recall, Oracle may choose the dense check or any sparse check). In contrast, the Decision Tree algorithm chooses, for every given input problem, a detection algorithm and a setting of its parameters based on the tree that was trained on the initial experimental runs. If the chosen algorithm/configuration does not produce an F-Score > 0.9, the Decision Tree counted as failing for this system even if there was another algorithm/parameter setting that could have achieved this F-Score.

Figure 6 shows the overhead of all our techniques when MVM is injected with faults from model 1 and the fault rate is 1e-3. The boxed ranges within each column represent the $25th$, $50th$, and $75th$ percentiles of the detector's overhead across all of the problems (roughly the mean $\pm$ one standard deviation). The lines within each column indicate the lowest and largest overhead within 1.5 Interquartile Range (IQR) of the lower and upper quartiles respectively. The detector overheads outside this range are represented as outliers with circles. The bars in Figure 7 show the fraction of problems on which each detection technique achieved the target F-Score.

These results show that the traditional dense check has an average overhead of 32%, ranging from 5% for denser problems to 80% for larger sparse problems with poor locality (e.g. *m3plates* and *bcsstm11*). While the fault detection overhead depends strongly on the size, sparsity, and locality of the problem, Figure 6 and 7 illustrate that, in general, a direct application of the dense check to sparse linear algebra can be expensive. This motivates the need for other algorithmic fault detection techniques that exploit the structure of sparse problems.

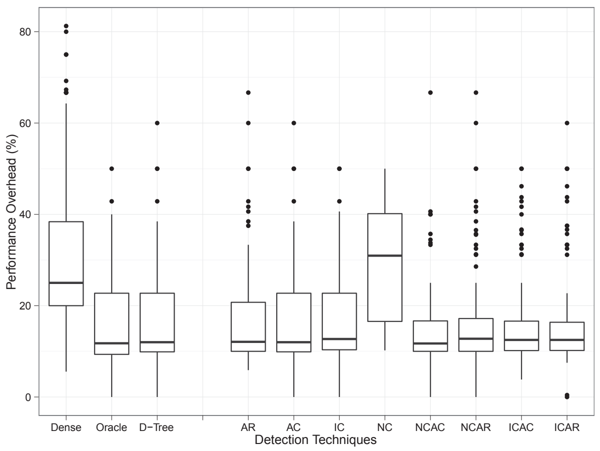

Fig. 6. Runtime overhead of each technique. F-Score target=0.9, Fault Rate=$1e-3$, FaultModel=1

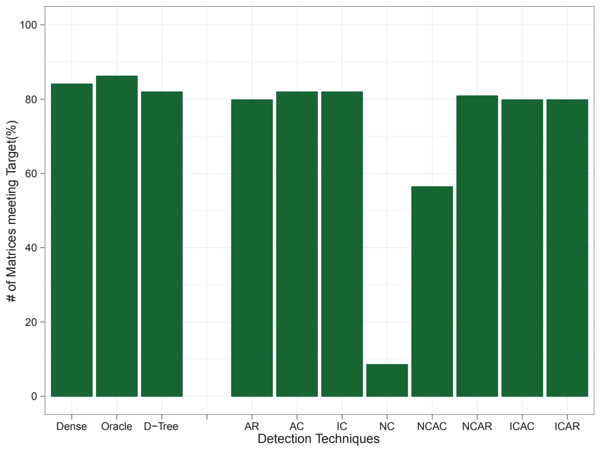

Fig. 7. Number of problems meeting F-Score target. F-Score target=0.9, Fault Rate=$1e-3$, FaultModel=1

In contrast, the overhead of AR was 16% on average, over the same set of sparse problems (i.e. 50% lower than the traditional dense check). This reflects the fact that AR's sparse samples are representative of these problems as a whole, which is most pronounced on problems with low sparsity and column sum variance such as such as *poli_large* and *t3dl_e*. Where this is not true, AR shows little improvement (e.g. less than 5% improvement for *qpband* and *impcol_d*) and the AC technique is needed exploit more complex structure.

The average overhead of AC is 17%, although the overhead in some problems (e.g. *impcol_d*, *big*, *tols2000*, and *chem97tz*) was reduced nearly in half relative to AR. AC was particularly useful for problems like these that contain low variance patterns within segments of their column sum distribution. The setup overhead of AC is considered in Section V-B in the context of linear solvers.

IC featured 18% average overhead and its effectiveness depends on the accuracy of the solution found for the identity equation ($c^T A = 1^T$). For many problems it was only necessary to run the least squares algorithm on $c^T A = 1^T$ to a tolerance of 1e-1, which corresponds to only 1-3 iterations of the algorithm.

NC had an average overhead of 29%. While this result may seem surprising, considering that the NC check does not need to compute $(c^T A)x$, since it is very close to 0, the reason for this was that the smallest singular value in most problems is too large (greater than $1e - 6$), making $(c^T A)x$ too far from 0 to produce an accurate check. Another problem is that the eigenvectors associated with small singular values often have many zeros, which may cause faults to be masked. Indeed, NC achieved F-Score above 0.9 for less than 10%, in contrast to $\geq$ 80% for the other techniques. For problems *netz4504*, *mimo28x28_system*, and *zeros_nopss_13k*, that do contain small singular values, the overheads were actually the least (11%) of any of the techniques. Therefore, NC is the best choice in certain scenarios.

The techniques combining sampling and conditioning had an average overhead of 17%. Note that the NCAC and NCAR were able to achieve larger success rates than NC (i.e. 82% rather than 8%), since NCAC/NCAR used sampling with the smallest eigenvector, instead of assuming that the smallest eigenvector was also near the null space and that the resulting check was zero.

The data shows that to achieve good performance and accuracy it is necessary to choose the detection technique and its parameters based upon the properties of the given problem. In particular, the diversity in the strengths of the different fault detectors provides significant power to leverage problem structure to optimize overhead and accuracy. The Oracle technique, which makes this choice optimally, achieves 15% overhead and reaches the F-Score of 0.9 with 92% of the problems, compared to only 81% for the individual techniques. The more practical Decision Tree algorithm is close to this optimum with providing 16% overhead, with an F-Score above 0.9 on 81% of problems. Such decision trees can be used by developers to pick the best check for their problem.

The benefits from the proposed checks are not only in terms of performance. Figure 8 and 9 illustrate a scenario (i.e. the same data, fault model, and F-Score as Figure 6 and 7, but with a less frequent fault rate of 1e-6), where the dense check becomes significantly more brittle, meeting the F-Score target with only 10% of the problems. In contrast, the Oracle can combine checks to cover 94% of the problems and the Decision Tree succeeds with 77%. This is because faults directly in the check are more likely to occur with the traditional dense check, which performs more operations than the proposed techniques. These errors can significantly distort the accuracy of the check to detect faults. This is primarily an issue at error rate=1e-6 because there is a high probability that an error occurs in the dense check but not in the main computations. When fault rates are significantly higher, the odds are high that both the main computation and the check

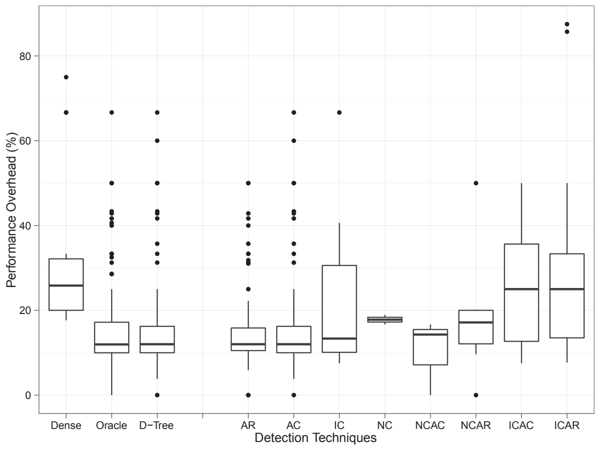

Fig. 8. Runtime overhead of each technique. F-Score target=0.9, Fault Rate=1e-6, FaultModel=1

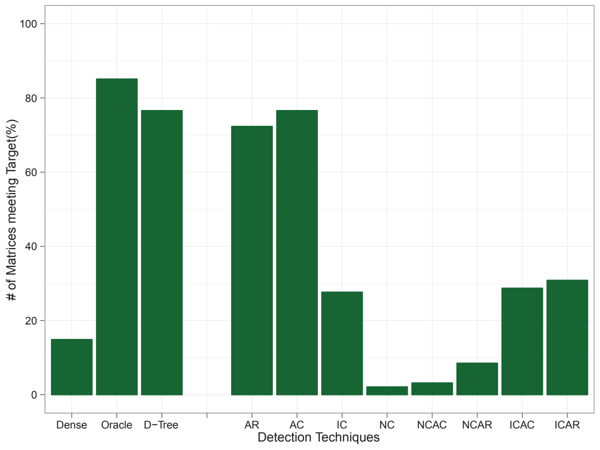

Fig. 9. Number of problems meeting F-Score target. F-Score target=0.9, Fault Rate=1e-6, FaultModel=1

will be hit. For significantly smaller fault rates, it is likely that neither will be hit.

Figures 11 and 12 generalize our results further, showing that they hold across different F-Score targets, fault models and fault rates. Figure 11 displays the average performance overhead of each technique and Figure 12 shows the fraction of matrices for which a given F-Score target is met. The left-most graphs in each figure plot these results for F-Score targets of 0.5, 0.75 and 0.9, showing that the results are not sensitive to the detector accuracy target. The middle graphs vary the fault models used to generate the faults, demonstrating that the detectors are equally capable of detecting all the different fault types discussed in Section IV. Finally, the right-most graphs vary the fault rate. Fault detection is easiest for large ($\geq$ 1e-4) and small ($\leq$ 1e-7) fault rates. When errors are common, signaling a fault on even slight signals is usually safe. Similarly, when errors are rare, it is safe to reserve the fault signal for only the most obvious faults. Detection is most difficult in the middle (1e-6 to 1e-5) where only finely-tuned detectors do well. This is visible in the right-most graph of Figure 12, which shows that detectors are most likely to miss their F-Score target in this range. Further, Figure 11 shows that the performance of the detectors is most erratic in this region. Importantly, the Decision Tree algorithm is largely resilient to the effect of this complexity, showing consistently better overhead and accuracy than the dense check across all the fault rates.

Figure 10 compares the different detection techniques to each other by showing how often each one is useful for different linear problems and under different fault models and rates. It shows the fraction of problem/fault scenario combinations for which each algorithm is chosen to be the best by both the Oracle and the Decision Tree algorithms. While each technique was useful in some cases, the sampling techniques were by far the most useful. AC works best for problems that have large column sum variances ($\in [5,1e6]$) and are not well-conditioned (condition number $>$1e6). NC performed well for problems with small singular values (smallest is $<$1e-6), high column sum variance (> 10), and a large condition number ($>$1e6). Problems that have a high column sum variance but a small condition number ($<$1e6) were best served by IC. IC was also useful for problems with high condition number when paired with sampling, since the overhead reduction of sampling more than made up for the less accurate solution to the identity equation. Similarly, NC often performed better for problems with large singular values when combined with sampling. High F-score targets and mid-range fault rates make more stringent demands on detection accuracy, which favors the clustering and conditioning techniques. The top level variables used by the Decision Tree to choose the best technique were the matrix condition number, matrix size, fault rate, column sum variance, and matrix sparsity.

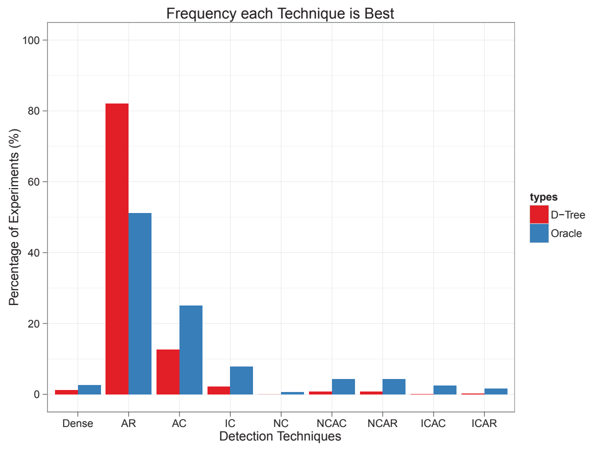

Fig. 10. Number of problems, fault models and rates for which each technique is the best choice.

B. Algorithmic Fault Detection for Linear Solvers

To evaluate MVM fault detection techniques in the context of real applications, we focus on the CG and IR sparse linear solvers described in Section IV-B. These solvers make extensive use of MVM, calling it in every iteration. This makes it possible to evaluate the impact of each detector's setup overhead, runtime overhead and detection accuracy in a realistic context, to determine how these properties ultimately affect each technique's utility.

Errors affect linear solvers in two ways. First, since iterative algorithms converge from a poor solution to an accurate one, undetected errors are likely to slow down the algorithm's convergence or even cause it to diverge. Further, detected errors are managed using the classic rollback-restart technique, where the application is rolled back to some prior point in its execution and its execution is resumed. Our experiments use the simplest variant of this technique where the solver rolls back to the start of the current iteration every detected fault. In order to insure forward progress, after several consecutive failed rollbacks, the most recent result is taken as the result and allowed to continue. In our experiments, each solver is executed until it reaches an error residual of 1e-6, meaning that if errors are detected, they may restart many iterations multiple times before they reach this goal.

We observed that although iterative solvers have some ability to recover from errors, in practice, non-trivial error rates can cause these algorithms to make little to no progress in many of the application runs. This means that the only way to run a linear solver, in face of many types of faults, is with some type of additional fault tolerance/check. Therefore, our experiments compared the performance of the linear solvers when using the sparse techniques to the solver performance when using the traditional dense checks.

Figures 13 and 14 make this comparison between the execution time of the solver implementations employing a sparse check (on the x-axis) and the corresponding implementation employing the traditional dense check, via a sequence of graphs. The difference is measured as

$$overhead = \frac{Time_{sparse\_check} - Time_{dense\_check}}{Time_{dense\_check}}$$

which means that a difference of -50% corresponds to the linear solver executing twice as fast with the sparse detector than with the traditional dense detector. The left two columns of graphs in both Figures 13 and 14 show the performance difference for CG and the right two columns for IR. Figure 13 shows results for basic CG and IR and Figure 14 focuses on their preconditioned variants. The first and the third column focus on the difference in just the time these algorithms spend on MVM operations. While the second and fourth column show the difference in overall execution times of the linear solvers. From top to bottom of Figures 13 and 14, the graphs show results for varying fault rates that range from 0 to 1e-4. Each detector and linear solver combination is evaluated on 5 different linear problems (separate sets for basic and preconditioned solvers). The average overhead over these matrices, for each detector, is shown in the middle of the boxed ranges. The upper and lower portions of boxed ranges within each column represent the mean $\pm$ one standard deviation of the overhead. The lines within each column indicate the maximum and minimum overheads. The set of problems, for use with the basic solvers, were chosen randomly from those used in Section V-A, such that the matrix showed little to

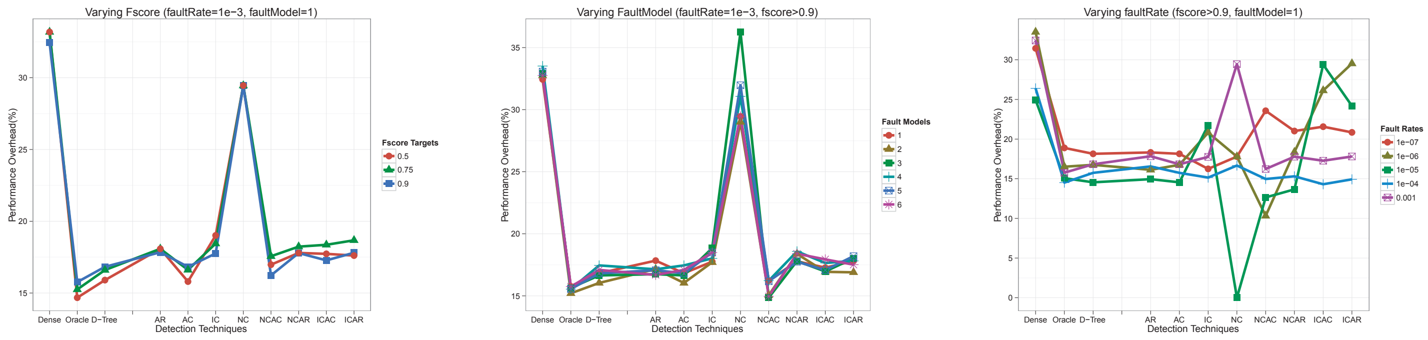

Fig. 11. Performance overheads with varying F-Score targets (left), Fault Models (center), and Fault Rates (right).

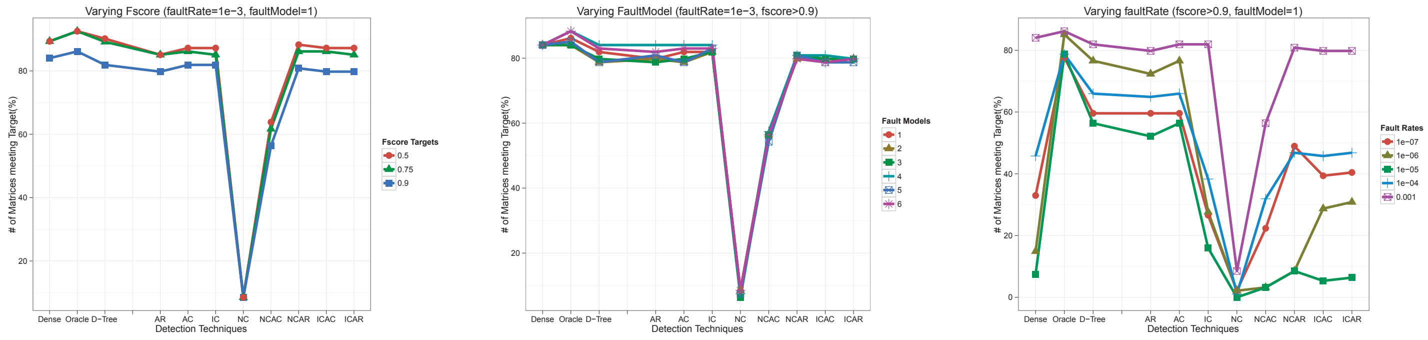

Fig. 12. The success rate or frequency at which problems meet the given F-Score target with varying F-Score targets (left), Fault Models (center), and Fault Rates (right).

no benefit with preconditioning ($< 10\%$). A different set of problems, used with the preconditioned solvers, were chosen randomly from the same set such that they achieved significant benefits with preconditioning ($> 2x$). Of the 5 problems solved successfully with CG-pre, only 1 of those same problems met the accuracy target successfully with IR-pre.

The results also show that in the context of linear solvers the dense checks can have fairly large performance overheads (30-50%). For CG, the sparse check based implementation spent 17% less time in MVM operations on average than the traditional dense check-based implementations. This corresponds to a total execution time that is 9% lower on average. For IR, the sparse check based implementations spent 10% less time in MVM operations than the dense check-based implementations on average. This corresponds to 5% lower total execution time on average.

The results show that the impact of larger setup overheads for some of the techniques (e.g. clustering and preconditioning), in the context of both the IR and CG, is fairly negligible ($< 0.01\%$), since the amount of reuse is high. We observed that the absolute amount of reuse in the context of CG is dependent on the conditioning of the problem which impacts the number of iterations required to reach the desired solution. The error rate can also have an impact on the number of iterations and hence the amount of reuse within the algorithm

Upon analyzing the performance of the techniques in the different scenarios shown in Figure 13, we observed that the overall overhead can vary greatly across different error rates. For example, with a fault rate of zero (first row of Figure 13), the impact of recovery overheads is reduced and the reduction in runtime overhead provided by the sparse fault detection techniques can be more fully utilized. As the error rate increases, the detection accuracy requirements become more stringent and the total overhead from missed faults and false positives must be properly balanced to provide the lowest performance overhead. By configuring the techniques to minimize the overhead from missed faults and false positives, the runtime benefits for many of the sparse techniques is also reduced to $< 5\%$.

When the error rate is sufficiently large, the impact of recovery overheads is again reduced allowing the sparse techniques to reduce the runtime and total overheads further. In many of the scenarios with smaller error rates, the NC based techniques do poorly due to the larger time required to find the smallest singular value for many of the sparse problems.

When using the solvers with preconditioning, Figure 14 shows that the total benefits from the sparse checks with CG-pre were relatively small on average ($5\% - 10\%$). For IR-pre, the sparse check based implementations spent about $30\% - 40\%$ less time overall than the dense check-based implementations. CG-pre, IR-pre, CG, and IR are four real application contexts that demonstrate that the sparse techniques are frequently able to exploit structure and reuse in sparse problems to reduce the overall overhead of algorithmic fault tolerance compared to the traditional dense checks.

VI. Conclusions

Future Exascale computing system will be prone to errors and severely energy constrained. On these systems it will be critical to detect and correct applications to ensure that applications can use them productively. This paper focuses on low overhead fault detection for sparse linear algebra algorithms which represents the core of a large class of HPC and emerging applications.

Previously proposed techniques for detecting errors in dense linear operations have high overhead (up to 100%, 32% on average). In this paper, we propose a set of algorithmic techniques that minimize the overhead of fault detection for sparse problems. The techniques are based on two insights. First, many sparse problems are well structured (e.g. diagonal, banded diagonal, block diagonal, etc.), which allows for sampling techniques to produce good approximations of the checks used for fault detection. These approximate checks are acceptable for many sparse linear algebra applications. Second, many linear applications have enough reuse that clustering and preconditioning techniques can be used to make these applications more amenable to algorithmic checks. We show that the proposed techniques exploit these opportunities to reduce overhead by up to $2x$ over traditional dense checks and maintain high error detection accuracy over a larger set of problems than the dense check. Further, the techniques also reduce overhead in the context of larger algorithms that use matrix-vector multiplications. Our experiments, which focus on the iterative linear solvers CG and IR show that the benefits were up to 40% when considering only the MVM operations, and up to 20% when considering non-MVM operations as well.

Acknowledgment

This research was supported in part by the FCRP Gigascale Systems Research Center (GSRC), Semiconductor Research Corporation, and the National Science Foundation. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-546472) and was partially supported by the Department of Energy Office of Science (Advanced Scientific Computing Research) Early Career Grant, award number NA27344.

References

[1] International Technology Roadmap for Semiconductors. White Paper, ITRS, 2010.
[2] J. Anfinson and F. T. Luk. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Comput.*, 37:1599–1604, December 1988.
[3] P. Banerjee, J.T. Rahmeh, C. Stunkel, V.S. Nair, K. Roy, V. Balasubramanian, and J.A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *Computers, IEEE Transactions on*, 39(9):1132 –1145, September 1990.
[4] Timothy A. Davis. University of florida sparse matrix collection. *NA Digest*, 92, 1994.
[5] Asanovic et. al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
[6] Berman et. al. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead. Technical report, DARPA IPTO, SEP 2008.
[7] E. Anderson et. al. Lapack: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, 1990.
[8] J. N. Glosli et. al. 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, pages 58:1–58:11, New York, NY, USA, 2007. ACM.
[9] Jack Dongarra Andrew et. al. A sparse matrix library in c++ for high performance architectures, 1994.
[10] Keun Soo Yim et. al. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, 2011.
[11] L. S. Blackford et. al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.
[12] Man-lap Li et. al. Swat: An error resilient system, 2008.
[13] Mark Hall et. al. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009.
[14] Nahmsuk Oh et. al. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51:111–122, 2002.
[15] Nithin Nakka et. al. An architectural framework for providing reliability and security support. In *In DSN*, pages 585–594. IEEE Computer Society, 2004.
[16] R. B. Lehoucq et. al. Arpack users guide: Solution of large scale eigenvalue problems by implicitly restarted arnoldi methods., 1997.
[17] Ronald F. Boisvert et. al. Matrix market: A web resource for test matrix collections. In *The Quality of Numerical Software: Assessment and Enhancement*, pages 125–137. Chapman and Hall, 1997.
[18] Sarah Michalak et. al. Predicting the Number of Fatal Soft Errors in Los Alamos National Laboratorys ASC Q Supercomputer. *IEEE Transactions on Device and Materials Reliability*, 5(3), 2005.
[19] Robert Falgout and Ulrike Yang. hypre : A library of high performance preconditioners. In *Computational Science, ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 632–641. Springer Berlin, Heidelberg, 2002.
[20] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518 –528, 1984.
[21] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.
[22] C. Kong. Study of voltage and process variation's impact on the path delays of arithmetic units. In *UIUC Master's Thesis*, 2008.
[23] Franklin T. Luk and Haesun Park. An analysis of algorithm-based fault tolerance techniques. *J. Parallel Distrib. Comput.*, 5:172–184, April 1988.
[24] Amitabh Mishra and Prithviraj Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Trans. Comput.*, 52(9):1089–1099, 2003.
[25] J. S. Plank, Y. Kim, and J. Dongarra. Fault tolerant matrix operations for networks of workstations using diskless checkpointing. *Journal of Parallel and Distributed Computing*, 43(2):125–138, June 1997.
[26] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.
[27] Cornelis Joost van Rijsbergen. *Information Retrieval*. 1979.
[28] P. Wesseling. *Introduction to Multigrid Methods*. Institute for Computer Applications in Science and Engineering (ICASE), 1995.

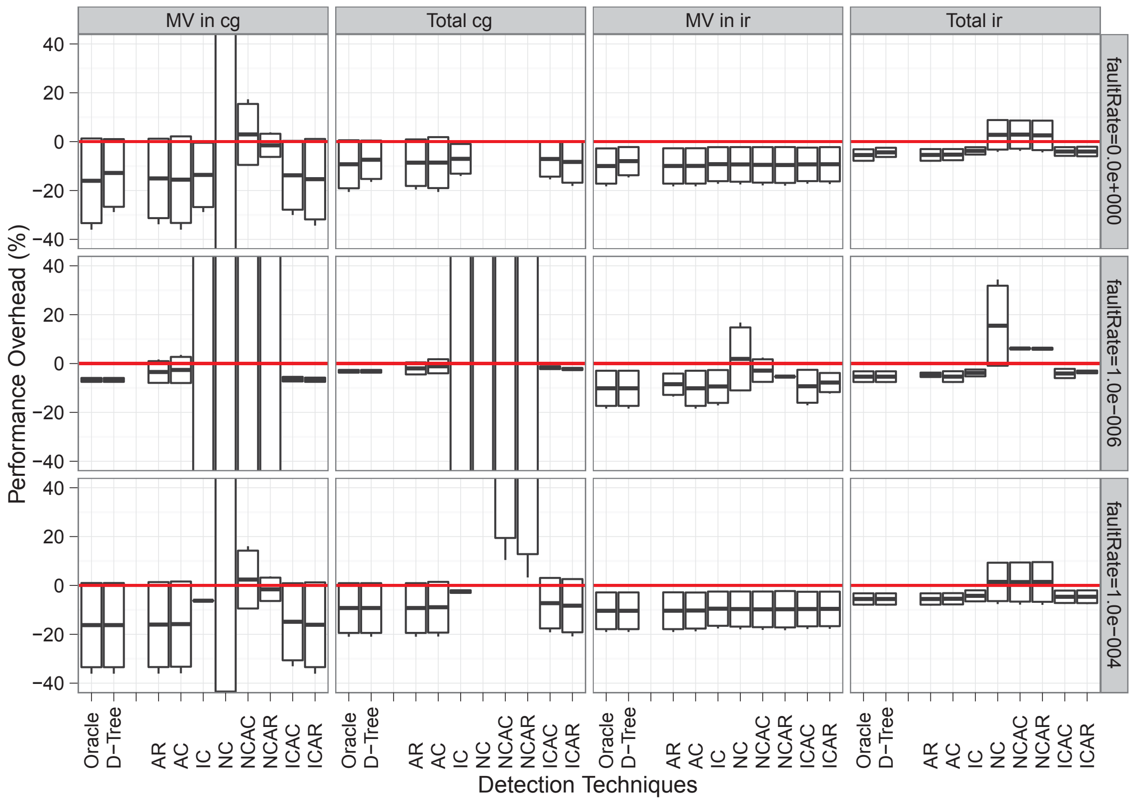

Fig. 13. Percent difference between the execution time of the sparse techniques vs. dense check applied to CG & IR. Columns $1-2$ are for CG and $3-4$ are for IR. Columns 2 and 4 show the total execution time overhead, and columns 1 and 3 show the MV execution time overhead.

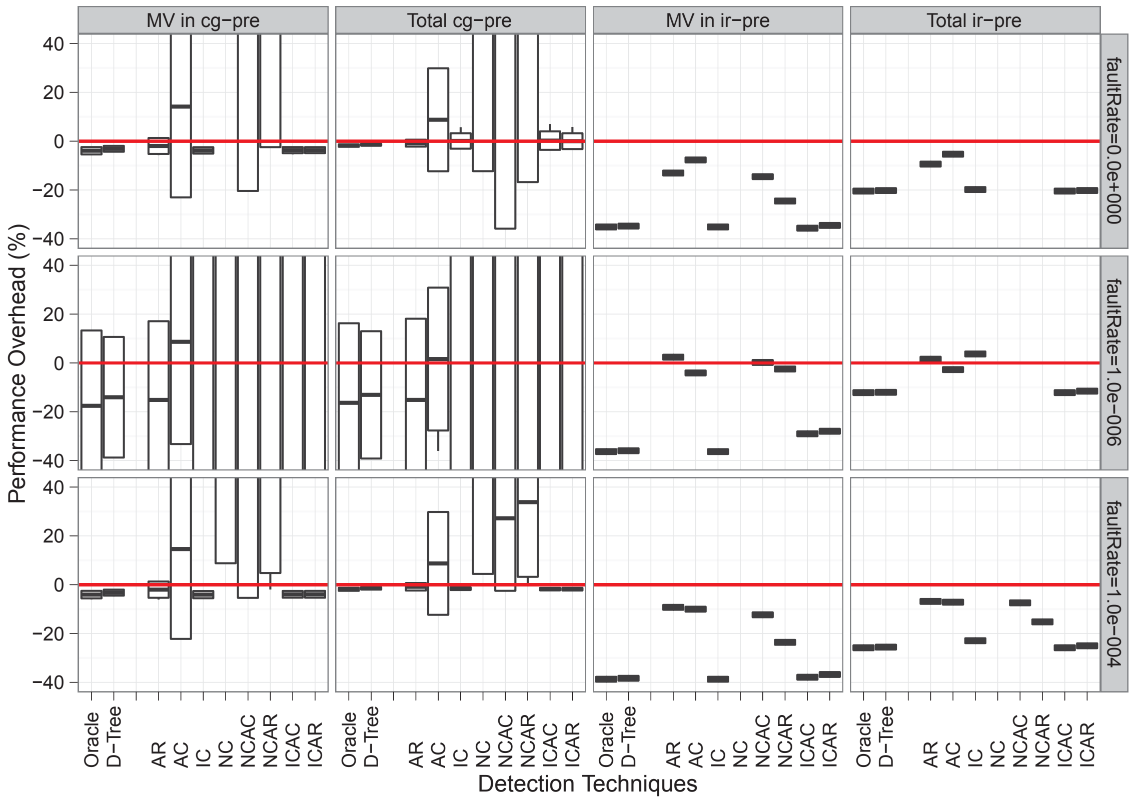

Fig. 14. Percent difference between the total execution time of the sparse techniques versus dense check applied to the CG & IR algorithms with preconditioning.