LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-400659

Soft Error Vulnerability of Iterative Linear Algebra Methods

Greg Bronevetsky, Bronis de Supinski

January 22, 2008

International Conference on Supercomputing
Kos, Greece
June 7, 2008 through June 12, 2008

Disclaimer

Soft Error Vulnerability of Iterative Linear Algebra Methods

Greg Bronevetsky and Bronis R. de Supinski
{bronevetsky1, bronis}@llnl.gov
Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory,
Livermore, CA 94551, USA

Abstract: Devices are increasingly vulnerable to soft errors as their feature sizes shrink. Previously, soft error rates were significant primarily in space and high-atmospheric computing. Modern architectures now use features so small at sufficiently low voltages that soft errors are becoming important even at terrestrial altitudes. Due to their large number of components, supercomputers are particularly susceptible to soft errors.

Since many large scale parallel scientific applications use iterative linear algebra methods, the soft error vulnerability of these methods constitutes a large fraction of the applications' overall vulnerability. Many users consider these methods invulnerable to most soft errors since they converge from an imprecise solution to a precise one. However, we show in this paper that iterative methods are vulnerable to soft errors, exhibiting both silent data corruptions and poor ability to detect errors. Further, we evaluate a variety of soft error detection and tolerance techniques, including checkpointing, linear matrix encodings, and residual tracking techniques.

1 The Soft Error Problem

Soft errors are one-time events that corrupt a computing system's state but not its overall functionality. They include bit-flips in memory and logic circuit output errors and may be caused by a variety of phenomena, including cosmic radiation, radiation from chip packaging [3], high temperatures, and voltage fluctuations.

Modern electronics are increasingly susceptible to data corruption from soft errors [1]. DRAM soft error rates (SERs) have been stable over the past several technology generations, but SRAM SERs have been growing exponentially as larger and larger memory chips come into use (1,000-10,000 FIT/Mb is typical, where a FIT is one failure per billion hours of operation) [3]. A cluster with 1000 processors, each supporting a 10Mb cache with 1600 FIT averages 10 errors per month [3]. Soft errors also impact SRAM-based FPGA designs: Xilinx reports SERs ranging from 401 FIT/Mb for 150 micron designs, to 51 FIT/Mb for newer 90nm designs [12]. Historically, soft errors primarily occur in memory. However, soft errors in microprocessor logic will soon also become common [19]. In particular, latches make up a large fraction of processor area, used in a variety of internal data structures. Since latch design is similar to but somewhat larger than SRAM cell design, they share many of the same vulnerability properties. Further, soft errors are a critical concern in the operation of real systems [10]. ASCI Q experiences 26.1 CPU failures per week [15]. A similarly-sized Cray XD1 supercomputer is estimated to experience 109 errors per week in CPUs, memory and FPGAs, if placed at the same altitude [17].

Given the high vulnerability of the large supercomputing systems, we must understand the impact of soft errors on scientific applications. To provide initial insight, we examine the soft error vulnerability of linear methods since many scientific applications rely on them [7]. We focus on linear methods that many believe are relatively immune to soft errors: iterative solutions to sparse linear systems, which we briefly describe in Section 2. In Section 5, we demonstrate that simple bit-flip errors frequently lead to erroneous solutions and runtime errors such as aborting despite the iterative approach. Detecting these errors is more complex than simply examining residual values, as we show in Section 6. Section 7 examines methods to minimize the cost of tolerating soft errors in iterative methods. Finally, we model the cost and benefit of the combined detection and tolerance mechanisms for a large scale parallel application that repeatedly uses iterative methods in Section 8. Overall, we find that the mechanisms support a trade-off between overhead and reduced soft error vulnerability. Low cost combinations have overheads as low as 3.4% while still reducing vulnerability by a factor of 1.5; alternatively, we can reduce vulnerability by a factor of 133 at the cost of increasing run time by 143%.

[0]This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (UCRL-ABS-XXXXXX).

2 Iterative Linear Methods

The linear system, $A\ x = b$, underlies many scientific computing applications. Methods that directly compute an exact solution for x, such as Gaussian elimination, are generally expensive, particularly if A is sparse. Thus, most applications use iterative methods, such as multigrid. These methods start with a sample solution and then iteratively refine it to find an approximate solution with an estimated error below an acceptable threshold.

For example, the Conjugate Gradient (`CG`) method 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 is to achieve accuracy within machine precision. As such, `CG` approximates the solution $x = \alpha_1 p_1 + ... + \alpha_n p_n$ with only a few vectors.

Under `CG`, the initial approximation is x_0; the residual $r_0 = b - A\ x_0$, 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. However, to ensure the that p_k is conjugate to prior p_i's, $p_k = r_k - \frac{{r_{k-1}}^\top r_{k-1}}{{r_{k-2}}^\top r_{k-2}} p_{k-1}$. The coefficients α_k are computed as $\frac{{r_{k-1}}^\top r_{k-1}}{{r_{k-2}}^\top p_k^\top} A p_k$. This process is repeated until r_k is below some threshold. Although other iterative methods compute subsequent approximations differently, all follow a similar pattern.

Two main properties of iterative linear methods shape the general perception of their soft error vulnerability. First, they begin with an imprecise solution and iterate to within some level of accuracy. As such, soft errors that do not corrupt the data of the matrix A, the vector b or control state, such as a pointer to a vector, should have little impact. Second, their residual norm, which tracks convergence towards a solution, can be used to detect errors by testing its progress for any abnormalities.

3 Target Iterative Methods

We focus on SparseLib [6], a sparse matrix library that includes several iterative solvers and linear operations on a variety of sparse matrix storage formats. We examine the soft error vulnerability of six iterative methods: Conjugate Gradient (`CG`); Conjugate Gradient Squared (`CGS`); Biconjugate Gradient (`BiCG`); Biconjugate Gradient Stabilized (`BiCGSTA`); Preconditioned Richardson (`PR`); and Chebyshev (`Cheby`). We evaluate these methods with 39 matrixes, each from a different group of the University of Florida Sparse Matrix Collection [5]. Since `CG` and `Cheby` only work on symmetric matrixes, we use each group's largest symmetric matrix; some groups have no symmetric matrixes, in which case we use the largest unsymmetric matrix. The collection provides the right-hand side, b, for many matrixes; for each matrix that does not include b, we use a right-hand side that corresponds to a solution vector x of all ones.

In order to establish a baseline for our soft error experiments, we applied each iterative method to each matrix to identify the smallest residual that the method achieves in under a minute. We used residual thresholds <1 since SparseLib's initial guess for x produces a residual of 1. We did not consider residuals <1e-150 since smaller residuals lead to numerical instability in most matrix/method combinations. Different matrix/method combinations have different minimum residuals: some methods only execute for a fraction of a second on some matrixes while others diverge for all target residuals

4 Fault Injection Methodology

We model the impact of soft errors by flipping a single randomly chosen bit at a randomly chosen time in the target iterative method's data structures. We implement fault injection into any object on the stack or heap through manual instrumentation of SparseLib. We do not inject errors during the reading of the matrix A and vector b since scientific applications use linear solvers as part of larger numerical algorithms that read the input data once but execute the linear solver many times. We also do not inject errors into system-dependent state since we focus on the soft error vulnerability of sparse iterative methods in general, rather than a specific implementation with a specific compiler. In particular, this means we do not flip bits in registers, `malloc` object headers, function return pointers and application code, since these all depend on a particular compiler or system library. Our results demonstrate that the soft error vulnerability of the methods is significant; injecting additional errors would only increase that vulnerability.

We perform our experiments on 2.4Ghz dual-core Opterons, with 2GB RAM each. We evaluate each method's fault vulnerability for the ten largest matrixes for which the method satisfies the constraints discussed in Section 3. We inject faults in the base methods (Section 5) and in methods enhanced with soft error detection (Section 6) and tolerance (Section 7) techniques. For each combination of iterative method, matrix, and error detection and/or correction technique we performed 500 trials. We analyze our data for the impact of a single bit flip on a single run of each method and on a parallel application that uses iterative methods internally and runs for one day on a thousand processors using 10FIT/MB memory, consistent with typical 1,000-5,000FIT/MB DRAM [18] with 90%-98% effective error correction.

5 Impact of Soft Errors

We first consider the four possible outcomes of a single bit flip on a single run of an iterative method:

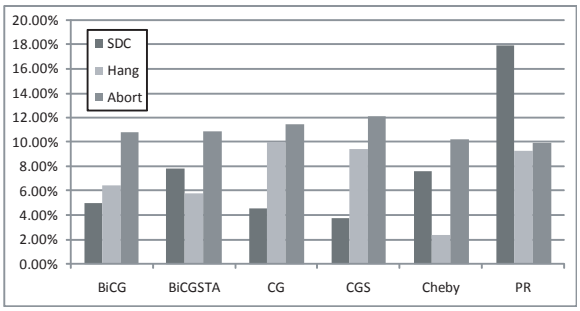

Figure 1: SDC, Hang and Abort Rates

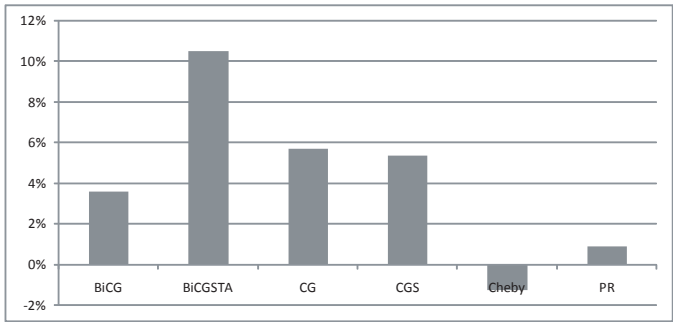

Figure 2: Percent Change in Time to Convergence

- Successful completion: the method converges to its target tolerance, and the error in the solution x is $\leq$10% larger than the fault-free error;
- Silent Data Corruption (SDC): the method converges with a final error >10% larger;
- Hang: execution time exceeds fault free execution time by a factor of at least ten, indicating divergence or convergence but above our threshold;
- Abort: a failed internal SparseLib test or an error such as a segmentation fault aborts the method.

Figure 1 shows the probability of a bit-flip resulting in an SDC, a hang or an abort in each iterative method. All three outcomes are quite common, with SDC rates ranging from 4% for `CGS` to 18% for `PR`. Hangs range from 2% for `Cheby` to 10% for `CG` and aborts range from 10% for `PR` to 12% for `CGS`. Considering only runs that completed successfully, Figure 2 shows the effect of soft errors on the iterative method's run time. This ranges from a 10% slowdown for `BiCGSTA` to a 1% speedup for `Cheby`.

Our observed rates of SDCs, hangs and aborts demonstrate that iterative methods are vulnerable to soft errors despite converging from imprecise estimates of x to more accurate ones. Figure 3 explains this perhaps counter-intuitive result: the matrix A dominates method state. Specifically, A's value array accouts for 55% of application state, while its row index and column pointer arrays take up 27% and 2%, respectively. The rest of the state is taken up by various vectors and the diagonal preconditioner matrix. Bit flips in the matrix A or vector b

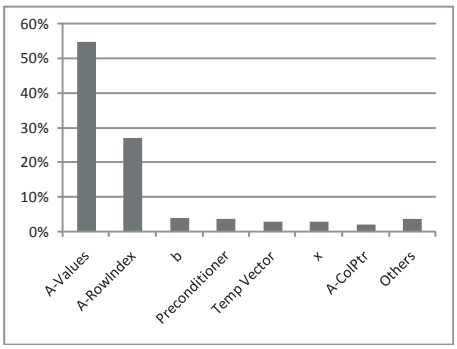

Figure 3: Application State Per Data Structure

can change the linear system being solved. If a method still converges to the target tolerance on this new linear system, the solution x could be very different from the solution to the original linear system. Further, the method will loop forever at higher residuals if it cannot converge within the threshold for the new system. Finally, the application could attempt to access unallocated memory if the bit flip corrupts A's row or column arrays, resulting in an abort. Thus, the above high rates of SDCs, hangs and aborts reflect the high percentage of application state occupied by the most vulnerable data structures.

Although we have shown iterative methods are vulnerable to soft errors, we still must consider how this translates to vulnerability of real applications in realistic soft error environments. We must scale our observations to account for the rarity of soft errors. Further, we must provide error estimates for applications running at realistic parallel scales since soft errors are already becoming prevalent on large scale parallel platforms.

We use our observations to compute the impact of soft errors on a model application that uses iterative methods. The model application runs on multiple processors, where each processor repeatedly executes an iterative method in a loop, using the result of one execution to determine the input of the next. The application does no work outside these calls to the iterative method. If soft error causes an SDC for the method in one execution of the iterative method, we assume the entire application produces an SDC. If an execution of an iterative method hangs or aborts due to a soft error, we assume the application restarts the method, which increases the application's run time without affecting the output. The model application represents a conservative upper bound on the vulnerability of applications that use iterative linear methods.

Figure 4 shows the impact on the model application's run time and its predicted SDC rate, assuming the application executes on one thousand processors for one day, using 10FIT/MB memory. We use a one day run time that is consistent with typical usage patterns of large scale clusters where users are more constrained by resource limits

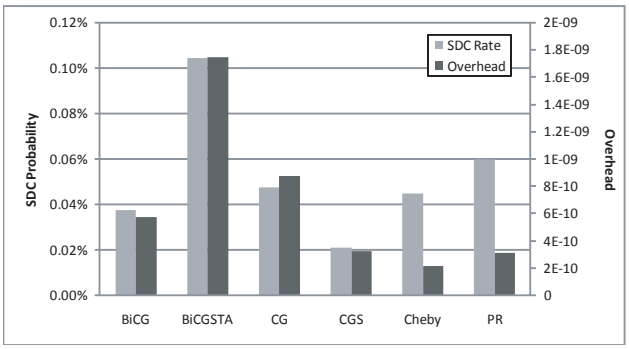

Figure 4: Overhead and SDC of Unprotected Applications

than by scientific objectives. As a reference point, if the above application typically uses 100MB of RAM per processor, there is a 2.4% probability that its memory will be corrupted by a soft error. We determine the number of executions of the iterative method during the one day run from the method's average run time without any fault injection. For all methods, soft errors have negligible impact on our model application's run time since our scenario results in a very low probability of a soft error occurring during a one day run. The SDC rate varies between 0.02% and 0.11%; although these rates may seem low, possible unreported errors in their results trouble most application scientists at even those frequencies. The silent data corruptions accumulate in the application's state: longer running times, more processors or less reliable memory cause the SDC rate to increase linearly. The high vulnerability of sparse linear iterative methods and their significance to application scientists highlights the need for techniques to detect and to tolerate soft errors in these methods.

6 Soft Error Detection

We evaluate three types of random error detectors for iterative methods. Our *Native Detector* (ND) uses the correctness tests implemented as part of each SparseLib iterative method to determine whether an error has occured.

Convergence detectors examine the sequence of residual norms. Since this sequence converges to our threshold over time in error free executions, an error has probably occurred if later norms are increasing. However, the methods can exhibit some increases even with correct execution so our convergence detectors must tolerate them. We consider the following convergence detectors:

- `Multiple-based Detection (MD(m))`: Signal an error if the immediately preceding residual's norm was a factor m smaller than the current residual's norm;
- `Averaging-based Detection (AD(a))`: Signal an error if the current residual's norm exceeds the average norm of the last a residuals.

Algorithm-Based Fault Tolerance(ABFT) [9] encodes all matrixes and vectors using a linear error correcting code. We augment each vector with an extra entry that contains the sum of the other vector entries. Similarly, we augment each matrix with an extra checksum row and/or column, where each entry in the extra row is the sum of its respective column and vice versa. Linear operations such as matrix-matrix multiplication, matrix-vector multiplication and matrix factorization on encoded matrixes and vectors produce encoded matrixes and vectors as their output. Our ABFT detectors report an error when for some vector, row or column $|currentSum - recordedSum| > |max(currentSum, recordedSum)| * tol$, where $currentSum$ is the current sum of the vector's entries, $recordedSum$ is the recorded sum and tol is a free parameter. We consider the following AFBT detectors:

- `ABFT_NRC(tol)`: Standard ABFT scheme with a checksum row and column; to prevent redundant error detections, we update the corresponding checksum entry to the current sum of the corrupted vector, row or column when an error is detected;
- `ABFT_RC(tol)`: Extends `ABFT_NRC` to update the checksum every time a given vector or matrix row or column is checked, which could help tolerate numerical instabilities.
- `ECC_NRC(tol)` and `ECC_RC(tol)`: Simplified variants of the ABFT detectors in which we encode the data structures used by SparseLib's vectors and matrixes with the same linear checksum code, which supports faster checks due to better spatial locality and lack of sparse matrix indirection logic but does not protect against errors due to erroneous computation.

Our detectors other than `ND` have free parameters that allow us to tailor detection accuracy to a specific iterative method and matrix. For each method and matrix combination, we set the free parameter by running the iterative method without error injection on all its other target matrixes. For each of these matrixes we identify the most relaxed value for the free parameter that does not detect any errors (i.e., has no false positives). We then remove the top and bottom 10% of free parameter values this method identifies and use the average of the top, middle and bottom thirds of the remaining values. Throughout our remaining experiments, we identify these settings as `top`, `middle` and `bottom`.

We compare each iterative method's run time with each of our seven detectors to the method's fault free run time with no error detection. Since they require little additional computation, we evaluate the `MD`, `AD` and `ND` detectors each iteration. Since they require significant additional computation, we study the impact of how frequently we use the encoding-based detectors (`ABFT_RC`, `ABFT_NRC`, `ECC_RC` and

ECC_NRC). We evaluate these detectors every p iterations for four values of p: 1, n, n^2 and n^3, where n^3 is the total number of iterations that the method requires to converge for a particular matrix.

In our figures, we specify each configuration as *errDet-tolerance-testEvalPeriod*, where:

- *errDet* is the error detector;
- *tolerance* is the detector's free parameter setting: *bottom*, *middle* or *top*; and
- *testEvalPeriod* is the number of iterations between different evaluations of the test: 1, n, n^2 or n^3.

Figure 5(a) shows the overhead averaged across all methods and matrixes (5 runs for each configuration) that each detector incurs relative to the fault free run time. For each detector with a free parameter, we show the overhead for each detection tolerance setting: `bottom`, `middle`, `top`. For the encoding-based methods, we show all four detection periods (1, n, n^2 and n^3). `MD`, `AD` and `ND` have minimal overhead (0% - 1.5%) despite our evaluating them every iteration. `ND` is the least expensive. `MD` and `AD` exhibit little difference between detection tolerances. Our encoding-based detectors impose a much higher overhead, ranging from 60% to 80% when evaluated every iteration. As the detection period rises, this overhead drops dramatically to less than 15% for the two least frequent detection periods.

Figure 5(b) shows the average number of errors detected for each detector, as a fraction of the total number of iterations. All such detections are false positives. As expected, tighter detection tolerances result in more false positives: `bottom` has more false positives than `middle`, which has more than `top`. `MD(bottom)` shows many more false positives than other convergence-based tests, which is consistent with smaller values of m mistaking ordinary fluctuations in the residual's norm for errors. The `ABFT` tests have more false positives than the `ECC` tests, which indicates that numerical instabilities can mislead these detectors. The encoding-based tests that reset the encoding after each test ($\_RC$) have more false positives than those that do not ($\_NRC$); apparently frequent updates of the checksum actually exacerbate numerical instabilities.

We examine the overhead of `ABFT` and `ECC` further in Figures 6 and 7, which show the impact on the cost of matrix-vector multiplication with `ABFT` and `ECC` enabled. Matrix-vector multiplication is the most expensive operation that the iterative methods perform. We consider the regular ($M*v$) and transpose ($v*M$) and average the cost over all matrixes, with 50 runs for each combination. Figure 6 shows the overhead of using the four `ABFT` and `ECC` variants with both matrix-vector multiplications. `ABFT` incurs between 6% and 18% overhead, while `ECC` imposes less than 1% overhead. The performance impact on transpose multiplication is two to three times that on regular multiplication. Figure 7 shows a scatter-plot for `ABFT_RC` (one point per matrix), with the run time overhead on the x-axis and on the y-axis the relative increase in the number of matrix non-zeros due to the encoding. Both regular and transpose multiplication show strong correlations between the overhead and the number of extra non-zeros.

7 Soft Error Tolerance

Our soft error tolerance techniques combine checkpoint-based recovery mechanisms with our detectors. Periodically checkpointing the entire application state is sufficient but can be expensive for applications with significant state. Thus, we evaluate two checkpointing options:

- `ChkptAllVars`: checkpoint all variables periodically;
- `ChkptWOnce`: checkpoint only the write-once variables (e.g., A and b) before the main iteration;

We also combine these options with the perfect detector `PD`, which signals an error one iteration after we inject a bit-flip and, thus, is an upper bound on the protection any detector can provide. We record checkpoints in RAM because the iterative methods have low run times that make disk-based checkpointing impractical. For `ChkptAllVars` we checkpointed the application once every p iterations, using the same four values of p as in Section 6. With the encoding-based detectors, we omit checkpointing periods shorter than the error-detection period. In this section and the following, we append the checkpointing period to our naming scheme described in Section 6.

Figure 8 shows the difference between the overhead of using `ChkptWOnce` with each error detector and the detector's overhead from Figure 5(a), without any error injection. The overhead of `ChkptWOnce` varies with the detector used. Although each scheme has the same one time checkpoint cost, different schemes have different error detection rates. Since each error detection causes the application to roll back its write-once data, more frequent error detection incurs higher overhead. Thus, error detectors with lower false positive rates incur a smaller overall overhead. The checkpoint cost itself is quite small, as indicated by the overhead under 1% for detectors with the lowest false positive rates. `PD`'s overhead is zero since it incurs no false positives.

Figure 9 shows the overhead with `ChkptAllVars`, computed as with `ChkptWOnce`, grouping results by detector and tolerance setting. Each bar color corresponds to a checkpointing period. The overheads with `ChkptAllVars` follow the same pattern as with `ChkptWOnce`. More frequent checkpointing and high false positive rates incur overhead as high as 300%. Alternatively, infrequent checkpointing can drop the overhead to 10%-40%. Simple detectors have the lowest overheads with infrequent checkpointing, generally between 5% and 30%. `AD` shows a somewhat different pattern with overheads increasing

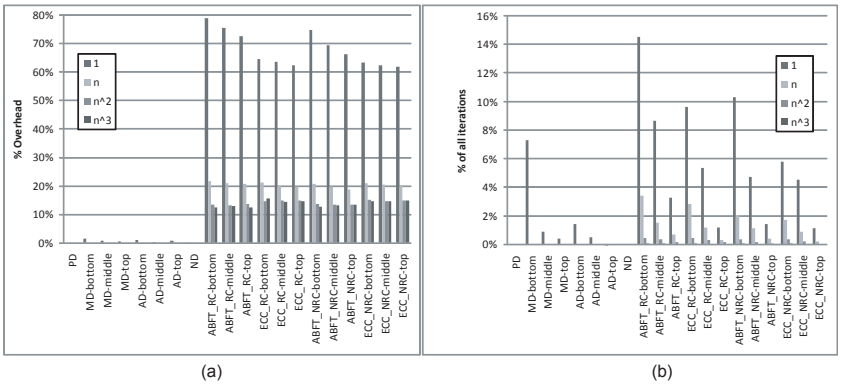

Figure 5: Error Detection - (a) Overhead and (b) False Positives

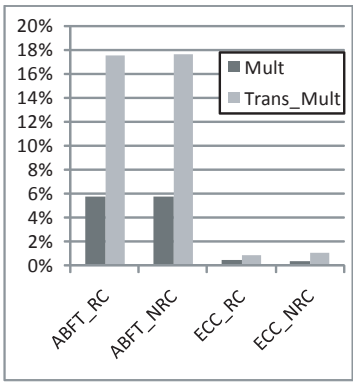

Figure 6: ABFT and ECC Overhead

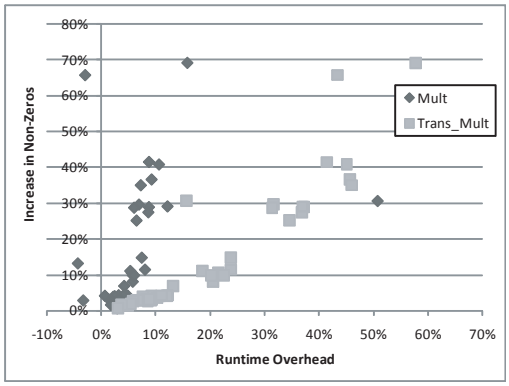

Figure 7: Matrix-Vector Multiplication Overhead vs. Non-Zeros

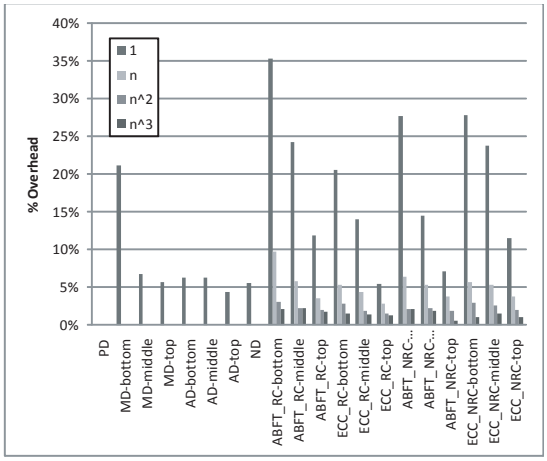

Figure 8: Overhead of `ChkptWOnce`

with larger checkpointing periods since the false positive rate of AD increases with an increasing checkpointing period, which leads to more rollbacks.

8 Application Impact of Detection and Tolerance

We now apply our soft error detection and tolerance techniques to the model application scenario described in Section 5. We also evaluate the detectors separately from the checkpointing mechanisms. We again assume that the fault-free model application run takes one day. However, some of our soft error tolerance techniques can increase the iterative method's run time significantly. To compensate, the model application executes each iterative method the same number of times for all error tolerance techniques. Thus, the application does the same work but takes longer and is therefore more vulnerable to soft errors with the more expensive techniques. When using a detector only, we assume the application re-executes a given run of an iterative method again whenever the detector signals an error. In all cases we assume that if a given run of an iterative method aborts or times out, the method will be re-executed by the application.

The overhead of each detection/tolerance technique configuration is computed by taking its average run time without error injection and computing the probability that it will be affected by a soft error during this time. This probability is then multiplied by the configuration's average overhead among the four possible cases: success, SDC, hang and abort. In the case of hang and abort the overhead is increased to account for the re-execution of the iterative method. We compute the SDC rate by multiplying the soft error probability by the probability that the soft error will result in an SDC.

Figure 10(a) provides a scatter plot of the overhead and SDC probabilities of all our techniques, capturing the impact of the tolerance setting. Figure 10(b) focuses this plot on the more promising techniques, showing only those with overhead less than 100% and SDC rates below 0.06%. We observe no obvious correlation between the tolerance setting and the resulting SDC rate, with Figure 10(b) showing that all threee settings appear frequently even among the options with the lowest SDC rates and overheads. While further study is needed to determine if this trend holds for a wider tolerance range, we observe that the appropriate tolerance stringency varies widely.

Figure 11 provides scatter plots that compare the efficacy of the detectors. We show points for PD (omitted from Figure 10 since it has no tolerance parameter) and *Base*, which uses no error detection or tolerance technique. We observe that we can choose some tolerance setting and checkpoint mechanism for every detector to reduce the SDC rate with relatively low overhead. However, convergence-based detectors usually result in a higher SDC rate but lower overhead than encoding-based methods. Further, encoding-based methods must be used carefully since we observe the lowest and the highest SDC rates correspond to techniques that use them.

The *Base* configuration features a very low overhead and a moderate SDC rate. Interestingly, its SDC rate (identified in Figure 11 by the horizontal line) is lower than many configurations that use complex tolerance techniques since *Base* executes more quickly than the other configurations and, thus, it has the lowest probability of being affected by a soft error.

Not surprisingly, the PD configurations offer the best trade-off between SDC rate and overhead. PD, combined with ChkptWOnce, provides an SDC rate of .12% and 4.5% overhead. PD performs even better when combined with ChkptAllVars using large checkpointing periods, providing a .003% SDC rate and 4%-6% overhead. Despite perfect detection, PD can incur an SDC if the error occurs during checkpointing, which produces invalid checkpoints. As such, PD has the highest SDC rate with ChkptAllVars when using the shortest checkpointing period.

We observe that significant reductions in SDC are possible with realistic soft error tolerance techniques. For example, WOnce-ABFT_NRC-top-n^2-1 reduces the SDC rate to 0.0001%, a factor of 51 improvement over the 0.053% *Base* SDC rate at a cost of 76% overhead, while AllVars-ABFT_RC-bottom-1-1 reduces the SDC rate by a factor of 133 relative to *Base* at a cost of 143% overhead. Meanwhile, AV-MD-top-n^3 and AV-MD-middle-n^2 impose a 3.4% and 12% overhead, respectively, while reducing the SDC rate to 0.034% and 0.029%, factors of 1.5 and 1.8 improvement over *Base*.

This discussion and our scatter plots show the difficulty of identifying a "best" error tolerance technique. Not only must we trade-off between reduced vulnerability and overhead but a technique's performance depends on its tolerance setting and test evaluation frequency. Therefore, we compare the techniques based on $Overhead + SDC * c$, where c is a constant that can focus the metric to favor solutions that are either more efficient (low values of c) or more reliable (high values of c). We list the top 20 error detection/tolerance configurations for several values of c in Table 1, in which we prepend our detector naming scheme used in our figures with WO (for ChkptWOnce), AV (for ChkptAllVars) or ED (for no checkpointing, i.e., "Error-Detection" only) to indicate the checkpoint option. When performance is most important the best configurations are *Base* and the variants of MD and ED. As we adjust c to focus on low SDC rate, the encoding-based detectors and full checkpointing become more appealing, tighter tolerance settings generally producing the best SDC results. $c = 1,000$ appears to be the point where the set of top techniques switches from performance-oriented techniques to reliability-oriented techniques. We observe that

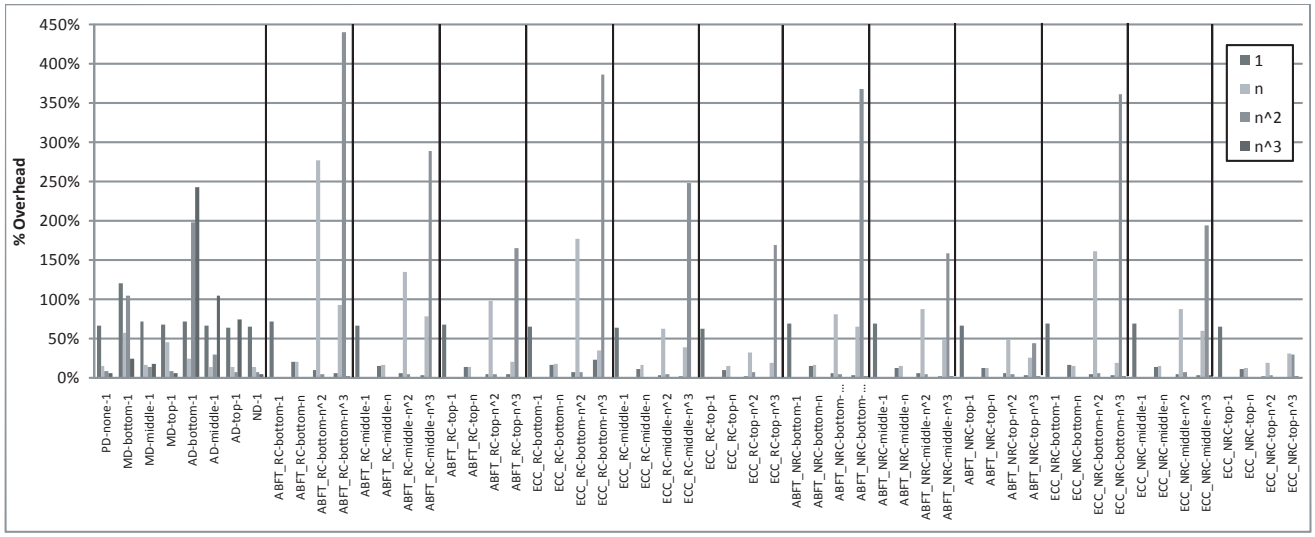

Figure 9: Overhead of `ChkptAllVars`

Figure 10: Comparing Overhead and SDC for Tolerance Settings

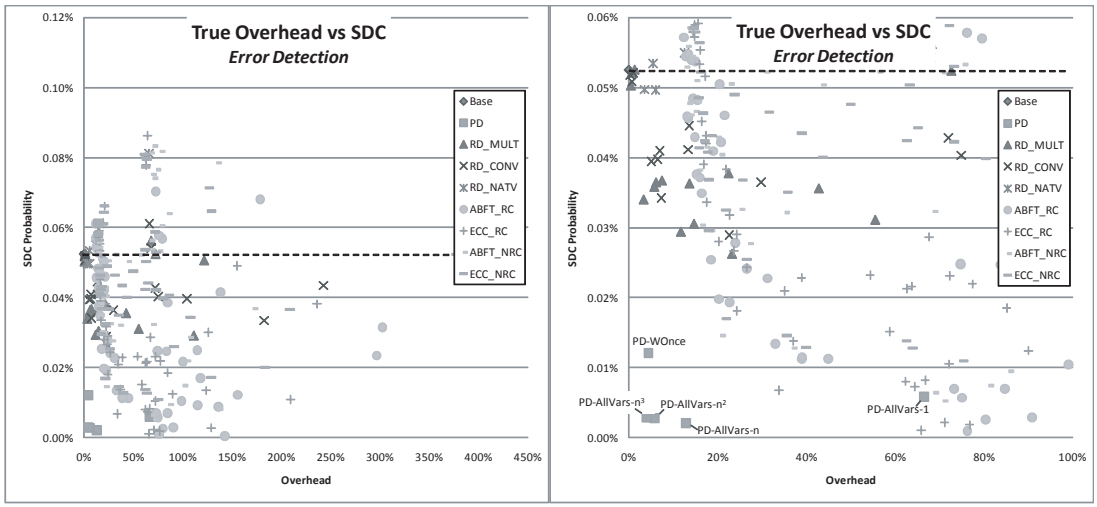

Figure 11: Comparing Overhead and SDC for Different Detectors

few ED (i.e., no checkpointing) configurations appear for large values of c, indicating that some checkpoint mechanism is essential to tolerate soft errors. Furthermore, ND only appears among the top detectors when SDCs are not important, implying that these simple sanity checks are not sufficient for detecting soft errors.

9 Prior Work

Prior work on fault injection falls into two general categories: (i) low-level studies of specific hardware and (ii) high-level studies of specific applications. Low-level studies focus on an SER evaluation of a specific piece of electronics such as a microprocessor [8][11][13], an FPGA [21] or a fault tolerant architecture [2]. Such work typically examines the raw error rate of the examined hardware but does not examine how these errors will affect real applications. Although such studies typically use specific applications as part of their experiments, these applications are either low-level testers [8][11][13][21] or simple applications [8][21][2]. For example, Kudva, etal. [11] used an IBM Power6 architectural verification program, Hiemstra and Baril [21] used a Windows NT workload generator and Arlat, etal. [2] used a controller application that kept a ball moving in a circle on a tiltable plane. While this category of work accurately estimates the soft error rates of physical devices, it provides little insight into the soft error properties of real applications. At best it provides raw error rates that may be used by higher-level studies to perform a more detailed application-level analysis.

High-level studies focus on the soft error vulnerability of a short list of specific applications. In particular, Lu and Reed [4] evaluated the soft error vulnerability of three MPI applications, showing correlations between error injection sites and the application's vulnerability to such errors. Skarin, etal. [20] took a similar approach to evaluate the soft error vulnerability of a brake-by-wire system for automobiles. Although both studies thoroughly evaluate the soft error vulnerability of their target applications, they provide little insight about the vulnerability of other applications, which makes it difficult to generalize the results. Alternatively, Messer, etal. [14] evaluated the soft error vulnerability of a realistic software stack. Although they focused on a specific combination of software components, they separated the effects of the operating system and the software stack, which illuminates the soft error properties of other applications running on the same OS and the same application running on different OSs.

The primary limitation of prior work is its restricted ability to predict the soft error properties of arbitrary applications. In contrast, this study represents an early step in developing this generic capability by characterizing the soft error properties of iterative linear methods, a common component of many scientific applications. Furthermore, this study presents and evaluates a variety of fault detection and tolerance mechanisms, informing future efforts to protect scientific applications from errors. This includes the first experimental evaluation of Algorithm-Based Fault Tolerance [9][16] encoding techniques on sparse matrixes.

10 Summary

We experimentally measure the soft error vulnerability of iterative methods and modeled the impact on large scale parallel applications based on them. Contrary to

c=1	c=10	c=100
ED-ND-1	ED-ND-1	ED-ND-1
Base	Base	Base
ED-AD-middle-1	ED-AD-middle-1	ED-AD-middle-1
ED-MD-top-1	ED-MD-top-1	ED-MD-top-1
ED-AD-top-1	ED-AD-top-1	ED-AD-top-1
ED-MD-middle-1	ED-MD-middle-1	ED-MD-middle-1
ED-AD-bottom-1	ED-AD-bottom-1	ED-AD-bottom-1
ED-MD-bottom-1	ED-MD-bottom-1	ED-MD-bottom-1
AV-MD-top-n^3	AV-MD-top-n^3	AV-MD-top-n^3
AV-ND-n^3	AV-ND-n^3	AV-ND-n^3
WO-AD-top-1	WO-AD-top-1	WO-AD-top-1
WO-ND-1	WO-ND-1	AV-MD-top-n^2
AV-MD-top-n^2	AV-MD-top-n^2	WO-MD-top-1
AV-ND-n^2	WO-MD-top-1	WO-AD-middle-1
WO-MD-top-1	AV-ND-n^2	WO-AD-bottom-1
WO-AD-middle-1	WO-AD-middle-1	WO-ND-1
AV-AD-top-n^2	AV-AD-top-n^2	AV-ND-n^2
WO-AD-bottom-1	WO-AD-bottom-1	AV-AD-top-n^2
WO-MD-middle-1	WO-MD-middle-1	WO-MD-middle-1
AV-MD-middle-n^2	AV-MD-middle-n^2	AV-MD-middle-n^2
c=1,000	c=10,000	c=100,000
AV-ABFT_NRC-bottom-n^2-n^2	AV-ECC_RC-bottom-1-n^3	AV-ABFT_RC-bottom-1-n^3
AV-MD-top-n^3	AV-ABFT_RC-bottom-1-n^3	AV-ECC_RC-bottom-1-n^3
AV-ECC_NRC-bottom-n^2-n^2	AV-ECC_RC-bottom-1-n^2	AV-ABFT_RC-bottom-1-1
AV-ABFT_RC-middle-n^2-n^2	AV-ECC_RC-bottom-1-n	AV-ECC_RC-bottom-1-n
AV-ECC_RC-bottom-n-n	AV-ECC_RC-bottom-n-n	AV-ECC_RC-bottom-1-n^2
AV-MD-middle-n^2	AV-ABFT_RC-bottom-1-n^2	AV-ABFT_RC-bottom-1-n^2
WO-AD-bottom-1	AV-ABFT_RC-bottom-1-n	AV-ABFT_RC-bottom-1-n
AV-MD-top-n^2	AV-ABFT_NRC-bottom-1-n^2	AV-ECC_RC-bottom-1-1
AV-ABFT_RC-bottom-n^2-n^2	AV-ABFT_RC-middle-1-n^2	AV-ABFT_NRC-bottom-1-n^2
AV-ECC_RC-bottom-n^2-n^2	AV-ABFT_NRC-bottom-1-n^3	AV-ABFT_RC-middle-1-n^2
WO-MD-top-1	AV-ECC_RC-middle-1-n^2	AV-ABFT_NRC-bottom-1-n^3
AV-ABFT_RC-bottom-n^3-n^3	AV-ABFT_RC-middle-1-n^3	AV-ECC_RC-bottom-n-n
WO-MD-middle-1	AV-ECC_RC-middle-1-n^3	AV-ABFT_RC-middle-1-n^3
WO-AD-top-1	AV-ABFT_RC-bottom-1-1	AV-ABFT_RC-middle-1-n
AV-MD-middle-n^3	AV-ECC_RC-bottom-n-n^3	AV-ECC_RC-middle-1-n^2
WO-AD-middle-1	AV-ABFT_RC-middle-n-n	AV-ABFT_NRC-bottom-1-n
AV-ABFT_RC-top-n-n	AV-ABFT_RC-top-n-n^3	AV-ECC_RC-middle-1-n^3
AV-ABFT_NRC-bottom-n^3-n^3	AV-ABFT_RC-middle-1-n	AV-ECC_RC-bottom-n-n^3
AV-ECC_NRC-bottom-n^3-n^3	AV-ABFT_NRC-bottom-1-n	AV-ABFT_NRC-bottom-1-1
AV-ABFT_NRC-middle-n^2-n^2	AV-ECC_RC-bottom-1-1	AV-ABFT_RC-middle-1-1

Table 1: Top 20 error detection/tolerance configurations for different c's

common opinion, we demonstrate that the methods and, thus, the applications can show high rates of hangs, aborts and silent data corruptions. We also show that simple soft error detectors can provide low overhead mechanisms with few false positives. However, these techniques probably do not provide an acceptable reduction in application soft error vulnerability. Instead, we show that the trade-off between SDC rate and overhead almost always favors checkpoint-based techniques. Overall we demonstrate that consideration of the soft error vulnerability of sparse iterative linear methods is important for successful supercomputing.

References

[1] International technology roadmap for semiconductors. White paper, ITRS, 2005.

[2] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Gunther H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, December 2003.

[3] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, September 2005.

[4] Charng da Lu and Daniel A Reed. Assessing fault sensitivity in mpi applications. In *Supercomputing*, November 2004.

[5] Tim Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 97(23), June 1997.

[6] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance architectures. In *Object Oriented Numerics Conference*, pages 214–218, 1994.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins University Press, 1996.

[8] David M. Hiemstra and Allan Baril. Single event upset characterization of the pentium mmx and pentium ii microprocessors using proton. *IEEE Transactions on Nuclear Science*, 46(6):1453–1460, December 1999.

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

[10] F. Irom and F.F. Farmanesh. Frequency dependence of single-event upset in advanced commercial PowerPC microprocessors. *IEEE Transactions on Nuclear Science*, 51(6), November 2004.

[11] P. Kudva, Jeffrey W. Kellington, Pia N. Sanda, Ryan McBeth, John Schumann, and Ron Kalla. Soft error derating of ibm power6 microprocessor using statistical fault injection. In *IEEE Workshop on Silicon Errors in Logic - System Effects*, April 2007.

[12] Austin Lesea and Joe Fabula. The rosetta experiment: Atmospheric soft error rate testing in differing technology fpgas - 90 nanometer update. In *Workshop on System Effects of Logic Soft Errors*, April 2005.

[13] P.T. McDonald, W.J. Stapor, and B.G. Henson. Pc603e 32-bit risc microprocessor radiation effects study. White paper, Innovative Concepts Inc., 1999.

[14] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D.D. Mannaru, A. Riska, and D Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Transactions on Computers*, 53(12):1557 – 1568, December 2004.

[15] Sarah Michalak. Estimation of the expected weekly number of soft errors in QA and QB. Technical Report LA-UR-04-5162, Los Alamos National Laboratory, 2004.

[16] Paula Prata and Joao Gabriel Silva. Algorithm based fault tolerance versus result-checking for matrix computations. In *International Symposium on Fault-Tolerant Computing*, pages 4–11, 1999.

[17] H. Quinn and P. Graham. Terrestrial-based radiation upsets: a cautionary tale. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 193–202, April 2005.

[18] Terrazon Semiconductor. Soft errors in electronic memory. White paper, Terrazon Semiconductor, 2004.

[19] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[20] Daniel Skarin, Martin Sanfridson, and Johan Karlsson. Impact of soft errors in a brake-by-wire system. In *IEEE Workshop on Silicon Errors in Logic - System Effects*, April 2007.

[21] Hamid R. Zarandi and Seyed Ghassem Miremadi. Dependability evaluation of altera fpga-based embedded systems subjected to seus. *Microelectronics and Reliability*, 47(2-3):461–470, 2006.