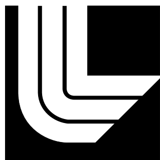
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UNT Digital Library



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-CONF-237305

Soft Error Vulnerability of Iterative Linear Algebra Methods

G. Bronevetsky, B. de Supinski

December 17, 2007

Workshop on Silicon Errors in Logic - System Effects
Austin, TX, United States
April 3, 2007 through April 4, 2007

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

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 become increasingly vulnerable to soft errors as their feature sizes shrink. Previously, soft errors primarily caused problems for space and high-atmospheric computing applications. Modern architectures now use features so small at sufficiently low voltages that soft errors are becoming significant even at terrestrial altitudes.

The soft error vulnerability of iterative linear algebra methods, which many scientific applications use, is a critical aspect of the overall application vulnerability. These methods are often considered invulnerable to many soft errors because they converge from an imprecise solution to a precise one. However, we show that iterative methods can be vulnerable to soft errors, with a high rate of silent data corruptions. We quantify this vulnerability, with algorithms generating up to 8.5% erroneous results when subjected to a single bit-flip. Further, we show that detecting soft errors in an iterative method depends on its detailed convergence properties and requires more complex mechanisms than simply checking the residual. Finally, we explore inexpensive techniques to tolerate soft errors in these methods.

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 [2], 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 FIT is failures per billion hours of operation) [2]. A cluster with 1000 processors, each supporting a 10Mb cache with 1600 FIT averages 10 errors per month [2]. 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 [7]. Historically, soft errors primarily occur in memory. However, soft errors in microprocessor logic will soon also become common [10]. For example, modern high-performance architecture techniques such as speculation (on anything) require tables to track history and on which to make predictions. These tables are equivalent to on-chip SRAM memory, and are just as susceptible to soft effors. Further, soft errors are a critical concern in the operation of real systems [6]. ASCI Q experiences 26.1 CPU failures per week [8]. 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 [9].

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 [5]. In this paper we focus on a subset of linear methods that many believe are relatively immune to soft errors: iterative solutions to sparse linear systems, which we briefly describe in Section 2. We then present experimental results that explore their soft error vulnerability. In Section 3.1, we demonstrate that simple bit-flip errors frequently lead to erroneous solutions as well as runtime errors such as aborting despite the iterative approach. Detecting these errors is a more complex task than simply examining residual values, as we show in Section 3.2. Finally, Section 3.3 examines methods to minimize the cost of tolerating soft errors in iterative methods.

2 Iterative Linear Methods

The linear system $A\ x = b$ is the cornerstone of linear algebra and appears widely in scientific computing applications. Methods that directly compute an exact solution for x, such as Gaussian elimination, are generally expensive. Thus, many methods, including multi-grid 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 ($p_i \ A \ p_j = 0$). Since the number of possible p_i's is too large to compute directly for large matrixes A, 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 $r_0 = A \ x_0 - b$, 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 becomes sufficiently small. 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 simplify soft error detection by testing its progress for any abnormalities.

3 Experiments

We focus on SparseLib [3], a sparse matrix library that includes several iterative solvers and linear operations on a variety of sparse matrix storage formats. We examined the soft error vulnerability of five iterative methods: Conjugate Gradient Squared (CGS); Biconjugate Gradient (BiCG); Biconjugate Gradient Stabilized (BiCGSTAB); Quasi-Minimal Residual (QMR); and Preconditioned Richardson (PR). We used the "Steve Hamm 20 bit adder" linear problem from the Harwell-Boeing Sparse Matrix Collection [4] for the test matrix A and vector b. Each method's target tolerance was chosen to achieve a total running time of a few seconds (tolerance and the number of iterations are listed in Table 1).

Each experiment flips a random bit of the solver's state at a randomly chosen iteration's end. We set injection locations to ensure that we inject a fault into every a variable that the application actually uses.

Method	Target Residual	Number of Iterations
CGS	1e-140	1,527
BiCG	2e-150	2,725
BiCGSTAB	1.e-80	8,299
QMR	1.1e-15	1,036
PR	1.e-7	24,828

Table 1: Iterative methods examined

3.1 Effect of Soft Errors

We divide the outcomes of our experiments into four categories:

- Successful completion: the residual reached the target tolerance, and the solution's error ($Ax-b$) was within 10% of the fault-free error;
- Silent Data Corruption (SDC): the residual reached the target tolerance but the error exceeded the fault-free error by more than 10%;
- Hang: the application executed much longer than in the fault-free case, which we judge indicates divergence or convergence to a local maximum;
- Abort: the application aborted.

All aborts in our experiments were caused by a segmentation violation. Even when the application completes successfully, it may take more or less iterations than in a fault-free execution. Figure 1 shows that the amount of time required for the tests that completed successfully is within 5% of the fault-free running time. However, even when the methods converge, Figure 2 shows SDCs are common, ranging from 1.3% to 8.5% of the tests for all the codes except PR, where it did not occur. Further, many runs hang (ranging from 1% for BiCGSTAB to 21% for BiCG) or abort (ranging from 5.4% for QMR to 9.8% for PR). Overall, no iterative method is immune to bit-flips, with BiCG showing the worst resilience.

3.2 Soft Error Detection

We evaluate four methods to detect random errors in iterative methods. Three methods observe the change in the residual across iterations while the fourth relies on correctness checks built into the implementations.

- `Multiple-based Detection (MD)`: Compare current residual p_k to the last residual p_{k-1} and signal error if p_k exceeds t times p_{k-1};
- `Averaging-based Detection (AD)`: Compare current residual p_k to average of last a residuals $(p_{k-1}, ..., p_{k-1-a})$ and signal error if p_k exceeds it;
- `History-based Detection (HD)`: Compare min of last n residuals, $(p_k, ..., p_{k-n})$, to max of m preceding residuals, $(p_{k-n-1}, ..., p_{k-n-1-m})$, and signal error if min exceeds max;
- `Native Detection (ND)`: Signal error if code invariants are violated.

Although the residual converges over time for all methods, it may increase by several orders of magnitude between successive iterations, which limits the detection accuracy of MD, AD and HD. We combine our convergence-based detectors with the standard OS segmentation fault detector

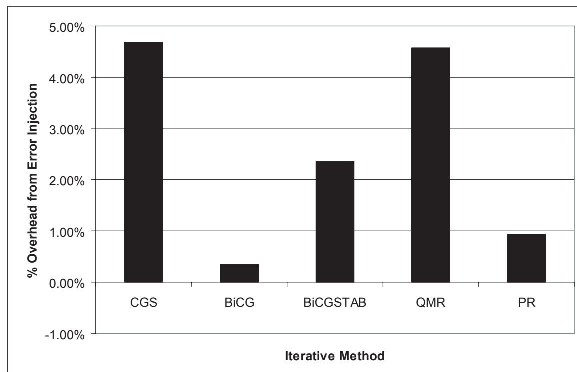


Figure 1: Percent change in run time for convergence

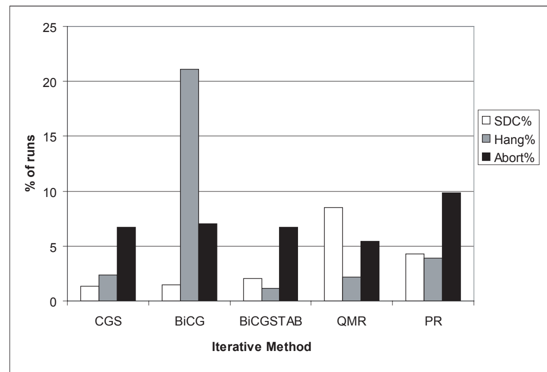


Figure 2: SDC, Hang and Abort rates

so that our accuracy rates refer only to runs that did not abort due to a segmentation fault.

We choose the free parameters for the `MD`, `AD` and `HD` to provide a conservative estimate of their error detection capabilities that gives a lower bound on their ability to detect errors. These parameter choices, which are shown in Table 2, ensure that there are no false positives for the given input matrix when no faults are injected.

Method Name	MD	AD	HD	
	t	a	n	m
`CGS`	5,000,000	75	6	6
`BiCG`	10,200	85	22	8
`BiCGSTAB`	800	1,900	120	130
`QMR`	1.2 5	95	35	33
`PR`	1.7	690	2	1

Table 2: Error detection parameters

We evaluate the detection accuracy of the three methods based on two definitions of an erroneous run:

- `BitFlip`: any run that was injected with a bit-flip;
- `OutFail`: any run that exhibits an SDC or a hang.

`ND` results are not shown for `PR` since this code does not have a native detector. We show the percentage of all runs with fault injection (i.e., `BitFlip`) for which the soft error was not detected, that is the false negative rate, in Figure 3. Our detection methods all exhibit high false negative rates for all codes, ranging from 75% to 100%, which indicates that they provide little error detection capability when we ensure no false positives for runs with no soft errors.

For the `OutFail` error model, we consider: (i) runs with fault injection that complete successfully and (ii) runs with fault injection that result in an SDC or a hang. As would be expected with our parameter choices, all detection methods rarely incorrectly identified a successful run as erroneous as shown by the false positive rates of no more than 12% in Figure 4. However, Figure 5 shows that they had generally high false negative rates (31% to

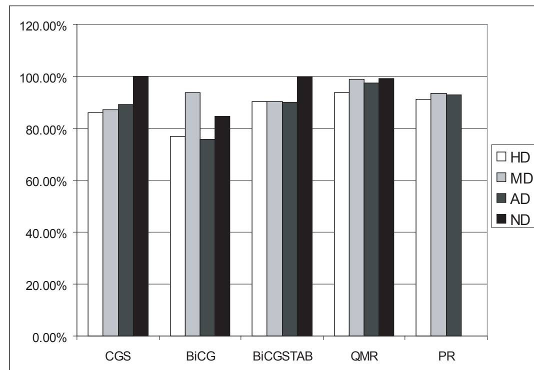


Figure 3: False negative rates for `BitFlip` error runs

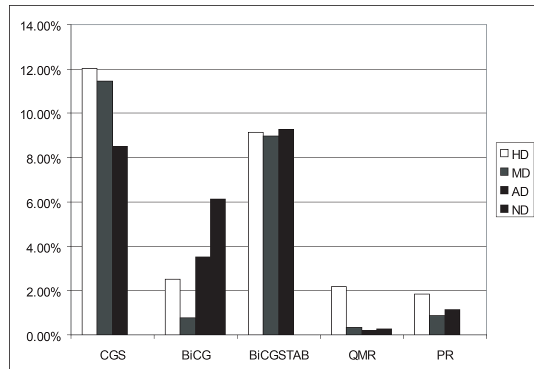


Figure 4: False positive rates type (i) `OutFail` runs

100%) for the erroneous runs. The only exceptions were `HD` on `PR` and `BiCG` and `AD` on `BiCG`, which performed fairly well. The performance of `ND` on `BiCG` was comparable to that of the convergence-based detectors. It was very poor in the other codes.

Accurate detection mechanisms are a topic for future work since none of ours work well overall.

An additional error detection concern is the ability of the application to detect errors in the final solution. While this error can be determined by computing $||Ax - b||$ after the last iteration, errors in A and b can corrupt this computation, causing it to produce an incorrect error esti-

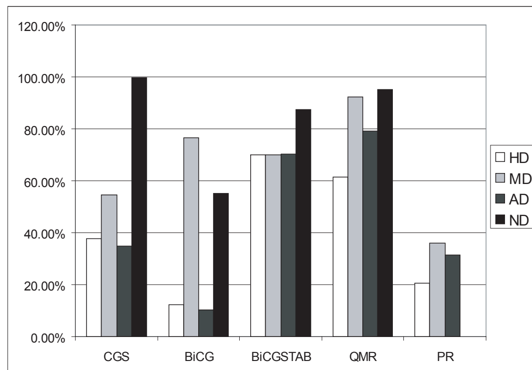


Figure 5: False negative rates type (ii) `OutFail` runs

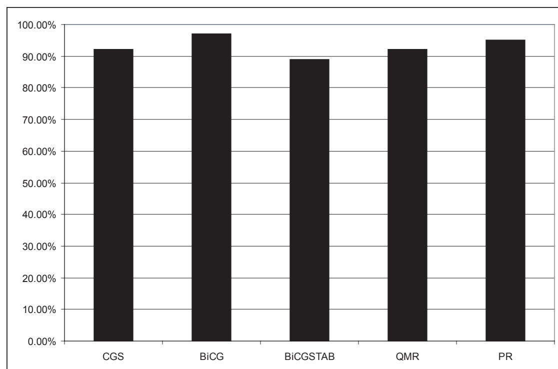


Figure 6: Rate of correct error estimations

mate. Figure 6 shows the probability that each method's computed output error is within 10% of the true error. While `BiCG` and `PR` show high correct estimation probabilities, the other methods are wrong as much as 10% of the time.

3.3 Soft Error Tolerance

We combine our soft error detection methods with a checkpoint-based recovery mechanisms to create full soft error tolerance solutions. Periodically checkpointing the entire application state is sufficient but can be expensive for applications with significant amounts of state. Thus, we evaluate these checkpointing options:

- `ChkptAllVars`: checkpoint all variables periodically;
- `ChkptWOnce`: checkpoint only the write-once variables (e.g. A and b) before the main iteration;
- `ChkptWOnceScalars`: checkpoint write-once variables before the main iteration and checkpoint all scalar variables periodically (all scalars are overwritten each iteration) .

We also combine these options with the perfect detector `PD`, which signals an error one iteration after a bit-flip is injected (the optimal protection for a convergence-based detector) to capture an upper bound on the protection they provide. Checkpoints are recorded in memory to provide a lower-bound on the cost. We use a checkpoint period of 1, 16, 256 or 4096 iterations for `ChkptAllVars` and `ChkptWOnceScalars`.

The checkpoint methods generally increase application run time slightly. `ChkptWOnce` is the cheapest scheme since it requires only one checkpoint. `ChkptWOnceScalars` captures some of the evolving iterative state at little additional expense. `ChkptAllVars` with a 1 iteration period is generally the most expensive. `ChkptAllVars` with `PD` and a checkpoint period of 4096 iterations was also observed to have a higher overhead due to longer rollbacks.

`ChkptWOnceScalars`, which restores scalar variables but not other multi-write data structures, can destabilize the computation, as evidenced by high hang rates with `ChkptWOnceScalars`+`PD` for `CGS` (up to 36%), `BiCG` (72%-79%) and `QMR` (94%) and increased convergence times for `CGS` and `BiCG`. This effect depends heavily on checkpoint period: some periods lead to a large increase in convergence time, while others show none at all. This destabilizing effect does not occur with other detection algorithms due to their lower error detection rates.

Overall, the convergence times of the `ChkptWOnce` and `ChkptWOnceScalars` are similar to those with no checkpointing although they do lower SDC and Hang rates. However, `ChkptAllVars` outperforms both: it has a low run time cost (checkpoints are written to RAM) and greatly reduces the rates of SDCs and Hangs, which are completely eliminated when combined with the `PD` detector. No technique reduces Abort rates because segmentation faults typically happen within one iteration of the error injection.

3.4 Differential Vulnerability of State

To more accurately understand the soft error vulnerability of iterative methods we examine the vulnerabilities of different types of application state. We have observed a difference between the effects of error injections into write-once variables ($WriteOnce$) and into variables that are overwritten during each iteration ($OverWrite$). One effect is the fact that while error injections into $WriteOnce$ variables can cause the iterative method to incorrectly estimate its final error (as shown in Figure 6), injections into $OverWrite$ variables do not have have this effect. The reason is that correct error estimation depends on the matrix A and array b having correct state. Since they are both $OverWrite$ variables, only $OverWrite$ error injections can have an effect on error estimation.

Another interesting effect, shown in Figure 7, is that errors injected into $OverWrite$ variables have 0% chance of resulting in an abort. This is due to the fact that injections into $WriteOnce$ variables mostly happen in the sparse matrix A. In our experiments this matrix was stored in compressed-column format, which maintains in-

dexes into an array of matrix data values. Bit-flips into these indexes can cause random memory to be accessed, resulting in segmentation violations. This effect is not seen with $OverWrite$ variables because they are all vectors and scalars and as such, use very little indirection. Figure 7 also shows a remarkably lower SDC rate in `CGS`, `BiCG` and `BiCGSTAB` for $WriteOnce$ injections relative to injections $OverWrite$. The source of this drop is unclear, particularly in light of the opposite behavior for `PR` and little difference for `QMR`.

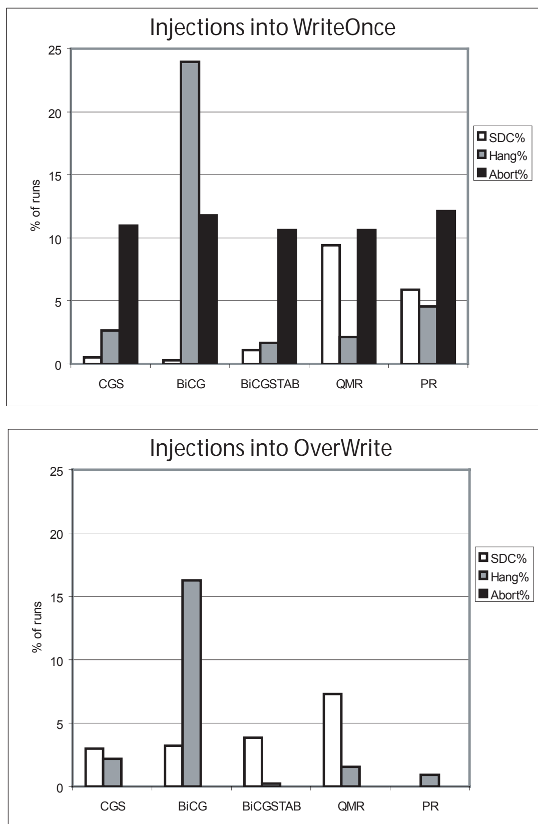


Figure 7: SDC, Hang and Abort rates for WriteOnce and OverWrite variables

4 Summary

We experimentally measured the soft error vulnerability of iterative linear methods. Contrary to common opinion, we demonstrate that they can show high rates of hangs, aborts and silent data corruptions. Further, we show that soft error detection requires more complex techniques than our three simple residual-based methods, despite the common belief that simple residual convergence would be sufficient. Finally, checkpoint-based fault tolerance techniques can be effective in making iterative methods less vulnerable to soft errors. However, the only fully effective technique is full checkpointing that uses the perfect detector, which may not be feasible in practice.

References

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

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

[3] 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.

[4] Iain Duff, Roger Grimes, and John Lewis. Single-event upset in evolving commercial silicon-on-insulator microprocessor technologies. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.