2011 17th IEEE Pacific Rim International Symposium on Dependable Computing

Numerical Defect Correction as an Algorithm-Based Fault Tolerance Technique for Iterative Solvers

Fabian Oboril, Mehdi B. Tahoori
Chair of Dependable Nano Computing (CDNC)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: {fabian.oboril, mehdi.tahoori}@kit.edu

Vincent Heuveline, Dimitar Lukarski*, Jan-Philipp Weiss*
Engineering Mathematics and Computing Lab (EMCL)
**Shared Research Group on New Frontiers in High Performance Computing Exploiting Multicore and Coprocessor Technology*
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: {vincent.heuveline, dimitar.lukarski, jan-philipp.weiss}@kit.edu

***Abstract*—As hardware devices like processor cores and memory sub-systems based on nano-scale technology nodes become more unreliable, the need for fault tolerant numerical computing engines, as used in many critical applications with long computation/mission times, is becoming pronounced. In this paper, we present an Algorithm-based Fault Tolerance (ABFT) scheme for an iterative linear solver engine based on the Conjugated Gradient method (CG) by taking the advantage of numerical defect correction. This method is "pay as you go", meaning that there is practically only a runtime overhead if errors occur and a correction is performed. Our experimental comparison with software-based Triple Modular Redundancy (TMR) clearly shows the runtime benefit of the proposed approach, good fault tolerance and no occurrence of silent data corruption.**

***Keywords*-algorithm-based fault tolerance, defect correction, conjugated gradient, triple modular redundancy, checkpointing**

I. Introduction

Nowadays, everyone expects that the computation result of a microprocessor is correct as long as the program, which is executed is correct. However, this attitude is wrong and can be fatal. Even if the underlying hardware is fault free and working in its specified parameter range, malfunctions can still happen. These are induced by transient or intermittent errors (so-called *Soft Errors*) due to cosmic radiation, temperature, signal noises as well as imperfect design. The susceptibility of memories or microprocessors to such errors is thereby increasing with shrinking CMOS feature sizes [1].

The failures can manifest themselves in bit flips in memory or during computation. Thereby, a recent study, that investigated the error rate in the main memory of servers discovered a much higher error rate than expected. Thereby, up to 10 bit flips per day for each memory module were detected [2]. Hence, modern servers use ECC-protected memory [3] to detect and correct single bit upsets and by this means increase the reliability. However, not only the main memory is susceptible to errors but also the microprocessors including their caches, registers and execution units. While some microprocessors also use ECC protection for some caches [4], the execution units and registers are more or less unprotected. In fact, research has also developed solutions for these domains to increase their fault tolerance (e.g. hardware-based Triple Modular Redundancy (TMR)) [5]), but these are still far away from an adoption in mass production. Hence, it is extremely important that also software developers are aware of the problem of unreliable hardware and adjust their programs accordingly to ensure correct results even if the computation is not fault free.

This is particularly important in the field of numerical methods and scientific computing, which includes the domain of *Computational Fluid Dynamics* (CFD) simulations, that are for instance used to calculate the air flow around cars or to forecast the weather. Due to the high complexity of the modeled problems the amount of processed data is extremely large and simulation runtime can easily exceed several days. Hence, the modeled problems in this field put extremely high demands on computing power as well as memory capacity. Furthermore, in order to ensure correct calculation through out the entire runtime, also the reliability requirements are very high. However, due to the performance demands in this field, always the latest hardware technologies are used, which are more susceptible to various failures such as Soft Errors. Hence, it is very important to not only optimize the software algorithms for a faster runtime, but also for a higher reliability in order to ensure correct results, even if some errors occur during the calculation.

Researchers have already proposed some *Algorithm-based Fault Tolerance* (ABFT) techniques to overcome the problem of unreliable hardware by means of software/algorithm level techniques as discussed in detail in Section II. Roughly speaking, most of the work focuses thereby on basic kernels like matrix-matrix or matrix-vector multiplications and tries to detect errors by adding special checksum techniques. However, such techniques are often not applicable to real world problems due to their calculation or data overhead. Real world problems modeled by linear partial differential equations are typically transformed into a linear system of equations $Ax = b$. The problem solution x is then calculated

978-0-7695-4590-5/11 $26.00 © 2011 IEEE
DOI 10.1109/PRDC.2011.26

with the help of software algorithms, called *solvers*. Many (iterative) solvers have intrinsic smoothing properties, that can correct errors without any assistance by (external) error detection and correction schemes, so that there is no need to make every operation step fault tolerant. Hence, to keep the overhead, to ensure reliable operation, as low as possible, the entire solver as one holistic entity has to be taken into account.

For this reason, we present an approach that uses the *numerical defect correction method* [6], [7] extended by *dynamic checkpointing*, which can correct errors (also named defects) independent of any (external) error detection and correction scheme. The defect correction method is an iterative solving algorithm for a system of linear equations $Ax = b$, that converts the original problem into a defect problem $Ad = r := b - Ax$, which is then solved by another method (inner solver). As inner solver we use in this work the *Conjugated Gradient method* (CG), due to its favorable convergence properties. The great advantage of this approach over others is, that it combines the intrinsic error correction properties of the defect correction method with the fast convergence of CG, which as a standalone solver is very vulnerable towards errors (it possibly calculates wrong solution, or does not converge at all). Hence, the result is a very fast solver with a high fault tolerance, that can implicitly correct errors without the need of any explicit error detection technique.

The results of our fault injection experiments clearly show that our defect correction approach with dynamic checkpointing can ensure a correct solution even for a 1000 times higher fault rate than the standalone CG solver. Thereby, the runtime overhead for low fault rates is negligible. Hence, this approach is much better suited than a CG solver extended with software-based TMR, that has a higher runtime overhead for low fault rates.

The rest of the work is organized as follows. In Section II some related work is presented. Our model problem is introduced in Section III and the Conjugated Gradient method is explained in the following Section IV. Afterwards, our methodologies including the numerical defect correction method and the dynamic checkpointing approach to achieve fault tolerance are introduced in Section V. Finally, the empirical results for these techniques can be found in Section VI, before we conclude with Section VII

II. Related Work

Algorithm-based Fault Tolerance (ABFT) techniques have been proposed as a means of low-cost error protection in numerical computations by incorporating error protection in the data representation as well as in the algorithm at the software level. With the prevalence of many- and multi-processor systems (such as multi-core, multi-socket, computer clusters, etc.), researchers have taken benefit of excessive (and at that time mostly unused) computation power to hide the performance overhead associated with ABFT.

Checkpointing is one fault tolerance scheme that can be combined with ABFT, in which all process states of the application are saved into a stable storage periodically [8]. In case an error occurs during calculation, the actual state is thrown away and instead the last backup is used. Such techniques have been further improved to deal with many-processor systems in which the failure of one processor may result in unnecessary restarts of other processors [9]. Checkpointing techniques in massive parallel systems have also been investigated [10]. However, saving checkpoints still means a high storage overhead and can only indirectly correct errors but cannot detect them, so that additional techniques for error detection are necessary. Moreover, in applications that are memory bandwidth bound, as it often happens in the field of numerical simulation and scientific computing, checkpointing can dramatically increase the application runtime. In order to minimize the overhead associated with checkpointing, an algorithm-based checkpoint-free fault tolerance method for parallel matrix computations has been presented in [9].

To detect errors during calculation *Result Checking* (RC) can be used. Thereby, the results are checked without knowledge of the particular algorithm used to calculate them. An RC for matrix-matrix multiplication $C = AB$ with input matrices A and B works based on the observation, that the product of C with a random vector r is equal to the product of matrix A with vector Br, if no error occurs, i.e. $Cr =?A(Br)$ [11]. However, also RC can only detect but not correct errors.

Another ABFT error detection method is a *checksum scheme* for matrix operations, which was introduced in [12]. The input matrices are augmented with an additional checksum row and an additional checksum column. Each element of the checksum column/row is the sum of the elements of the original matrix that are in the same column/row. The augmented matrices are then multiplied using an unmodified multiplication algorithm – in the end, the additional row and column of the result matrix should still be the sum of the elements of the same row or column. If that is not the case, an error occurred. A linear algebraic model for checksum-based ABFT has been developed in [13]. ABFT for matrix inversion with maximum pivoting using checksum methods was proposed in [14]. A series of row and column operations were defined in this work which satisfy the checksum property. ABFT for floating point matrix operations using backward error assertions has been presented [15]. The use of the floating-point arithmetic coding approach to build fault survivable high performance computing applications has been explored in [16].

In [17] a recovery mechanism is proposed for software using several processes running in parallel. However, additional error detection schemes are necessary to detect

errors before the recovery mechanism can correct them. Furthermore, the presented methodology only works, if there is a known close relation E between the different states of the processes, i.e. $E = S_{p_1} + S_{p_2} + \cdots + S_{p_n}$, since the recovery scheme is based on this relationship.

ABFT techniques have also benefited from many-processor systems to hide the overhead in high performance numerical systems [18]–[21].

An ABFT approach for iterative solvers for partial differential equations has been presented in [22]. The used technique is based on checksums, that are added to a red-black successive over-relaxation (RB-SOR) solver. However, this approach does not take any advantage out of the intrinsic error correcting properties of this solver. In addition RB-SOR is very inefficient in terms of convergence speed, which makes it uninteresting for many real world problems.

In summary, all existing ABFT techniques for matrix operations are somehow based on adding checksum rows and columns, and performing extra computations for computing and checking them. This requires additional memory and runtime overhead even if no error occurs. In contrast, our proposed approach does not add any memory overhead, in the form of checksums, to the matrices and vectors, and takes advantage of inherent numerical defect correction to achieve fault tolerance. Therefore, almost no computation overhead is incurred when there are no errors. Moreover, our approach can correct errors without any explicit detection scheme, which makes it very efficient.

III. Model Problem

For the sake of simplicity the model problem under consideration in this work is a two-dimensional Poisson problem $-\Delta u = f$ for an unknown function u in the 2D unit square with homogeneous Dirichlet boundary conditions and a given right hand side f [23]. A typical discretization by means of finite difference or finite element methods on equidistant grids with grid size $h = 1/(n+1)$ for a large integer n results in a linear system of equations (LSE) $Au = b$, where A is a matrix of size n^2-by-n^2 and the vector b with length $N := n^2$ represents the discrete values of the right hand side. This system is characterized by the classical 5-point Laplacian matrix shown in Equation 1.

$$A = \begin{pmatrix} B & -I & & & \\ -I & B & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & B & -I \\ & & & -I & B \end{pmatrix} \in \mathrm{R}^{n^2 \times n^2}, \quad (1)$$

with $B = \begin{pmatrix} 4 & -1 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -1 & 4 \end{pmatrix} \in \mathrm{R}^{n \times n}$, and the identity matrix $I \in \mathrm{R}^{n \times n}$.

Beside this particular model problem our proposed fault tolerant solver can handle many other and much more complex problems as well. The only prerequisite is that the discretization of the problem results in a symmetric matrix A. This is due to the requirements of the applied Conjugated Gradient method, that is introduced in the following Section IV.

IV. Conjugated Gradient Method

A. Basics

One of the most powerful methods for solving symmetric and positive definite linear systems $Ax = b$ is the Conjugated Gradient method (CG) [24]. There are two main advantages of this algorithm. First, it provides a favorable convergence rate for most problems and second, it can be used as an out-of-the-box solver without any information of the spectrum of the matrix A (i.e. the eigenvalues of A) or the right hand side b. The idea of the algorithm is to update the current approximation of the solution by a new vector with respect to the A-orthogonal projection of the residual. With respect to the memory consumption this algorithm is optimal due to the fact that only three vectors need to be stored.

A pseudo code of the CG method is presented in Algorithm 1, where A is the input matrix and b is the right hand side of the system, the initial guess is given by vector $x^{(0)}$ and the residual is denoted by r. Here, (p, q) is the scalar product of two vectors p and q and the discrete L_2-norm of a vector r is given by $\|r\|_{L_2} := \sqrt{h(r, r)}$.

Each iteration of the Conjugate Gradient method gives a new approximate solution $x^{(k)}$ where the stopping criterion is evaluated by means of the corresponding residual $r^{(k)} = b - Ax^{(k)}$ which is implicitly calculated in the step $r = r - \alpha q$. Hence, in the fault free case and without floating point rounding errors the exact residual in step k is given by the recursion for r in Algorithm 1. The convergence of the method, i.e. $x^{(k)} \to x^{(*)}$ with the exact solution $x^{(*)}$, is determined by the condition number of the matrix A given

Algorithm 1 Conjugated Gradient Method

$x = x^{(0)}$ initial guess vector
$R = r = b - Ax$, $\rho = (r, r)$, $\beta = 0$, $p = 0$
for $k = 1$ to MAX_{iter} and $\|r\|_{L_2} < \epsilon \|R\|_{L_2}$ **do**
  $p = r + \beta p$
  $q = Ap$
  $\alpha = \rho / (p, q)$
  $x = x + \alpha p$
  $r = r - \alpha q$
  $\rho_{old} = \rho$, $\rho = (r, r)$
  $\beta = \rho / \rho_{old}$
end for

by

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the maximal and minimal (positive) eigenvalues of the matrix A. For our model problem we find $\kappa(A) = 4n^2/\pi^2 = 4N/\pi^2$.

Then it can be shown that the error $x^{(k)} - x^{(*)}$ in iteration k can be estimated by

$$||x^{(k)} - x^{(*)}||_A \leq 2 \left[\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right]^k ||x^{(*)} - x^{(0)}||_A, \quad (2)$$

where $||x||_A := \sqrt{(x, Ax)}$ is the energy norm, $x^{(*)}$ is the true solution, $x^{(0)}$ is the initial guess, $x^{(k)}$ is the k-th approximation of the solution.

B. Computation and Storage Costs

From (2) one can conclude that the error in the energy norm always decreases from one iteration to another. Furthermore, an upper bound for the maximum number of iterations k such that $\|x^{(*)} - x^{(k)}\|_A \leq \epsilon \|x^{(*)} - x^{(0)}\|$ can be derived by

$$k \leq \frac{1}{2}\sqrt{\kappa(A)} \ln\left(\frac{2}{\epsilon}\right) + 1$$

For our model problem, which is presented in Section III with $N = n^2$ unknowns, it follows that

$$k = O(N^{1/2}). \quad (3)$$

Due to the sparsity of the matrix A in this scenario, we have $5N$ additions and $5N$ multiplications in the matrix-vector multiplication, $2N$ operations per scalar product, and $2N$ operations per scaled vector update. In total, in each iteration step $20N$ floating point computations are necessary, such that the overall computational costs equals $O(N^{3/2})$.

The storage costs can easily be derived by looking at Algorithm 1 of the CG method. As one can see, five vectors (x, b, r, q, p) and the matrix A are used. Since A is sparse (most of the elements are zero), it is stored in a special format consuming less space than storing each element of the matrix (here, e.g. compressed sparse row format). That means that for a problem with N unknowns a storage capacity for $12N$ elements, each with a size of 64 bit (due to double precision), is necessary.

C. Fault Tolerance Behavior

Without explicit control of the behavior of the solution procedure the algorithm itself cannot "recognize" errors occurring during the computation. This can lead to an unacceptable accuracy loss for the solution. In addition, the CG method uses a 3-term short recursion and has an intrinsic memory effect with respect to the A-conjugated search directions [23], [24]. Once this memory is disturbed, e.g. by induced Soft Errors, the solution possibly cannot be found and the iteration does not complete with success. If the error appears as data corruption in the solution vector, this error cannot be determined, leading to a crucial reliability problem called *silent data corruption*. In order to avoid this, an explicit computation of the residual vector in the CG algorithm can be introduced. To this end, the residual update $r = r - \alpha p$ needs to be replaced by $r = b - Ax$. However, this step would significantly (more than 50%) increase the computational costs in each iteration by additional $11N$ calculations. Since this overhead also exists, if no errors occur during runtime, this approach is not applicable for real world problems. As a remedy, one can explicitly calculate the exact residual only once after the solver has computed the final solution, in order to detect a faulty solution, which does not satisfy the accuracy requirements.

V. Fault Tolerance Methodologies

In the following section we will explain our approaches to maintain reliability for numerical computations. As mentioned in the introduction, modern servers and processors use ECC protected memory and caches [3], [4]. Hence, we use the reasonable assumption that all data, which belongs to the control flow or is read-only is always correct. Especially this means, that conditional branches, loops and the executed operations (e.g. Ra=Ra+Rb) are always correct. Only the operands and results of calculations can be faulty. For the CG method presented in Algorithm 1 this means that the stopping criterion is always evaluated correctly and also the loop indices, which are often used as array indices as well, are assumed to be correct. Hence, it is only the computation part, which has to be protected.

A naive approach to make calculations less vulnerable is the usage of *Software Implemented Hardware Fault Tolerance* (SIHFT) techniques in the form of *Triple Modular Redundancy* (TMR). TMR triplicates the data, executes the operation once on each of the three data sets and afterwards chooses the correct result by majority voting. Hence, this approach not only detects errors but also has the ability to correct them. However, TMR is a brute force technique, that does not take specific properties of the algorithm into account. Furthermore, the triplication of data and operations leads to a high overhead in terms of data but also in terms of computing time (if not run fully parallel). Thereby, the overhead with respect to runtime is that huge for low fault rates, that this technique is in the most cases not usable (see the results in Section VI).

A. Numerical Defect Correction Method

We propose to use a solver with good intrinsic error correction properties. Our goal is to obtain the correct solution without huge additional costs in terms of performance and data storage. Therefore, we have chosen the *defect correction method*. As one can see in the pseudo code illustrated in Algorithm 2 the defect correction method consists of two steps,

Algorithm 2 Defect Correction Scheme

$x = x^{(0)}$ initial guess vector
$R = r = b - Ax$
while $\|r\|_{L_2} > \epsilon \|R\|_{L_2}$ **do**
  Solve (e.g. with CG) $Ad = r$
  $x = x + d$
  $r = b - Ax$
end while

which are mathematically strictly equivalent to the original linear system, if exact arithmetic is assumed. First of all, there is an outer iterative loop in which the original problem $Ax = b$ is transformed into a defect problem $Ad = r$, where the residual r is defined by $r := b - Ax$ (for an approximate solution x). This new problem is then solved in a second step by an inner solver. Afterwards, the solution x is updated by the computed defect $x = x + d$. In case the solution does not satisfy a certain accuracy, these steps are repeated. By this means the defect correction method ensures convergence to the correct solution even if (hardware) computation faults during the computation happened, which means that the proposed defect correction scheme is robust. This is in contrast to the standalone CG solver, where the convergence cannot be ensured if computation faults occur (see Section IV-C), which is a knock-out criterion. However, multiple iterations of the outer loop of the defect correction method can be necessary. Hence, the defect correction method can correct (hardware) computation faults without the usage of any explicit error detection technique (i.e. hardware failures are treated as numerical defects). Furthermore, the defect correction method is also "immune" against silent data corruption, which is a big advantage of this approach. As inner solver we use the already introduced CG method due to its high convergence speed.

B. Computation and Storage Costs

Since we use the CG method as inner solver, the considerations on the computation and storage costs for the inner solver of our model problem (see Section III) can be found in Section IV-B. For the calculation of the norm and the two other operations in the outer loop of the defect correction method only $15N$ floating point computations are necessary, which means that in the fault free case the costs of the outer loop are almost negligible compared to the costs of the inner solver and a stand-alone CG solver. For example in our experiments in the fault free case 52 CG-Iterations are executed, which means that the overhead of the defect correction method compared to the standard CG method is:

$$\frac{52 \cdot 20N + 1 \cdot 15N}{52 \cdot 20N} = 1.01,$$

i.e. 1 %. However, if faults occur during the computation, the outer loop is typically performed up to a few times. Thereby, the corresponding performance overhead is based on two effects. First, the number of outer loops may increase with error injection rate. Second, the number of inner CG steps may increase as well. But as observed in the experiments, the additional overhead compared to a fault free execution is kept in a moderate range (see the performance analysis in Section VI-D). In terms of storage an additional capacity of N elements ($64N$ bits) is necessary compared to a normal CG solver, since the defect vector d has to be stored additionally.

C. Flexible Checkpointing

A further improvement can be achieved by adding a flexible checkpointing technique to the inner solver. In case of high fault rates, it is very probable that the inner solver is stopped because faults lead to `nan`- or `inf`-values. In that case, the intermediate results for the defect vector are thrown away, which means that the inner solver is restarted another time for exactly the same problem. In case the fault rate is too high, there will be too many restarts which can adversely affect runtime (and also convergence). For this reason, we have developed a special flexible checkpointing technique to overcome this problem. After every m-th iteration the computed approximation (in that case the defect) is stored in a backup vector. In case the inner solver stops due to `nan`- or `inf`-values, the defect vector is restored from the backup (checkpoint). Afterwards, the defect correction method continues as normal. Since it can happen that the inner solver is stopped before m iterations are done, m can be decreased dynamically according to the fault rate. In other words, the checkpointing rate is adjusted based on the history of the restart rate for the inner solver.

However, this feature takes another $64N$ bits of storage space for saving the backup vector. If m is large enough (compared to the number of iterations of the inner solver), the additional runtime overhead for backup storage and loading is negligible. For a better overview and comparison, the costs for all methods is put side-by-side in Table I for the 2D Poisson problem with homogeneous boundary conditions.

Method	Computation Costs	Storage Costs
Pure CG	#CG-Iterations $\times$ $20N$	$768N$ bits
CG with TMR	#CG-Iterations $\times$ $60N$	$1408N$ bits
Defect Correction	#CG-Iterations $\times$ $20N$ + #Outer-Loops $\times$ $15N$	$832N$ bits
Defect Correction & dyn. Checkpointing	#CG-Iterations $\times$ $20N$ + #Outer-Loops $\times$ $15N$	$896N$ bits

Table I
ESTIMATION OF THE COMPUTATION AND STORAGE COSTS FOR DIFFERENT SOLVERS WITH DIFFERENT FAULT TOLERANCE FOR THE 2D POISSON PROBLEM WITH N UNKNOWNS (FAULT FREE CASE)

VI. Results

In this section we analyze the proposed techniques for "fault-tolerant" iterative solvers based on the CG method in a practical situation. We compare the results of our approach with those of the original CG method and those of a version of the CG method combined with software-based Triple Modular Redundancy (TMR). We perform a comprehensive analysis with different fault rates, where faults are induced by fault injection during runtime.

A. Fault Injection

Fault injection is a widespread technique to test the reliability of different hardware or software. Since this work is completely software-based, fault injection at algorithmic level is used to investigate the reliability improvements of the different techniques presented in Section V. Every time an error-vulnerable data item (see classification in Section V) is written, a fault injection routine is called. This routine will then introduce a fault into the data item with a given probability. Therefore, the data is transfered from a decimal number representation (floating point) to a binary representation according to the IEEE 754 specifications [25]. Afterwards, the fault injection routine randomly flips bits and saves the data again with a decimal representation (floating point). The entire process is illustrated in Figure 1. Thereby, each bit has the same probability to be faulty. By this means also multiple bit faults per data element can be injected. The runtime for fault injection is not considered in our timing measurements.

Our model assumption that no errors occur in datapaths and in control logic is motivated by our conceptual approach. These errors would result in unpredictable program behavior and corrupted results. The corresponding errors and impact on results can not be detected or corrected at an algorithmic level. Since our proposed concepts are algorithm based

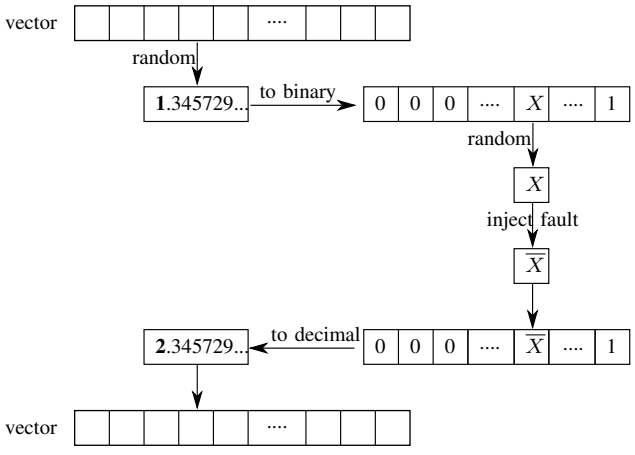

Figure 1. Fault injection routine

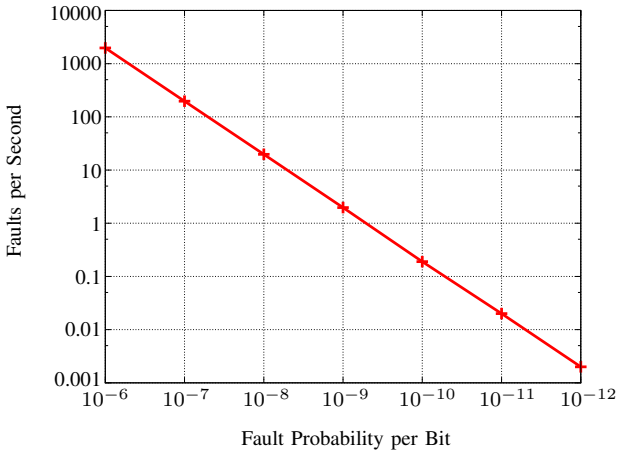

Figure 2. Number of faults per second for each fault rate.

fault tolerance schemes, the underlying assumptions are reasonable in our setting.

In our experiments a fault rate (= probability that a bit is faulty) between 10^{-12} and 10^{-6} has been used. The average number of (injected) faults for each fault rate is illustrated in Figure 2. Please note that these fault rates are extremely high compared to those observed in current hardware. However, they indicate what can happen with future hardware technologies or in special application areas (e.g. aerospace with high radiation intensity).

B. Setup

In order to investigate the methodologies presented in Section V, the standard test case for partial differential equations – the 2D Poisson equation on the unit square with homogeneous boundary conditions discretized by the Laplacian matrix – has been used (see Section III). The number of unknowns is 2 millions, which corresponds to a vector size of $128 \cdot 10^6$ bits using the double precision floating point format. All experiments have been performed on a system with four 12-core AMD Opteron-6174 processors with 2.2 GHz clock rate and with a total system memory of 128 GByte of DDR3-RAM. The operating system is RHEL 6 and for compilation of the different solvers the build-in gcc-4.4.4 has been used.

A solution x_{comp} calculated by the iterative solvers is considered to be a correct solution (within prescribed error tolerance), if the difference to the exact solution x^* satisfies $\|x^* - x_{comp}\| < 10^{-10}$. Since fault injection randomly selects bits to be faulty, all experiments for different settings have been performed 50 times in order to get reasonable results.

C. Vulnerability of CG, CG with TMR and Defect Correction

As a first aspect of the analysis, the study of the calculated solutions by the three different methods (CG, CG with

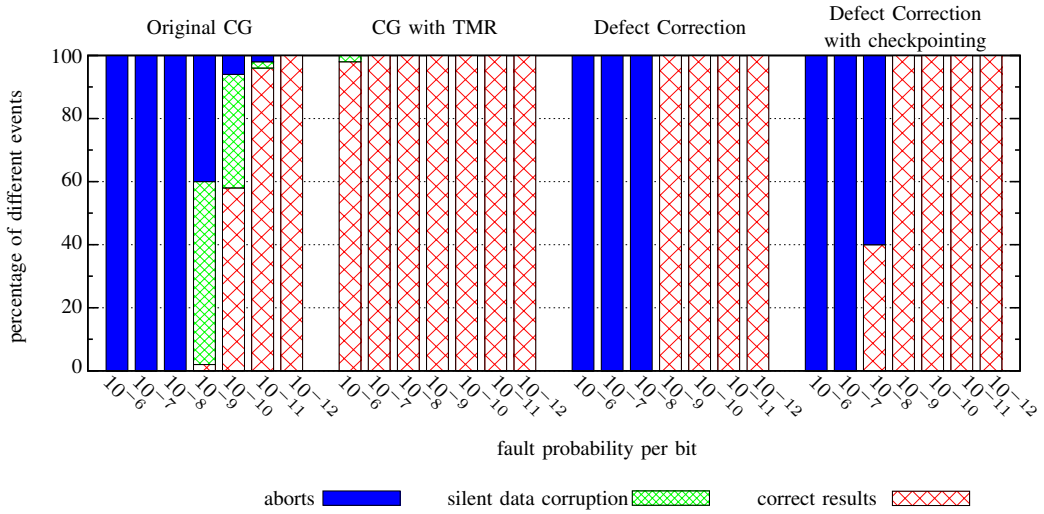

Figure 3. Percentage of runs where the correct result has been obtained, silent data corruption has occurred, and runs that have been aborted.

TMR, and CG with defect correction) is a good choice. Therefore, we introduce three different categories for the computed solution: 1.) correct results, 2.) aborted runs with no valid solution (i.e. error detection but no correction) and 3.) completed runs with incorrect results. While the first two categories preserve data integrity, the third one corresponds to silent data corruption, which is a crucial reliability problem. This means, that data is erroneous, but the application and thereby the user do not recognize it. Hence, it is extremely important to verify the calculated solution x_{comp} by checking if $\|A \cdot x_{comp} - b\| < \epsilon$ is satisfied for a given ϵ.

In Figure 3 the percentages of the three categories are depicted. As one can easily see, the original CG method without any fault tolerance techniques is in our case inappropriate if fault rates are higher than 10^{-11}. For bigger problem sizes, even lower fault rates are problematic, since the probability of a faulty bit is increasing with increasing problem size. With software-based TMR as fault tolerance technique the situation is much better. However, also the fault protection of TMR is limited. Since TMR just triplicates the data, executes the same operation on the three data sets, and afterwards chooses the correct result by majority voting, high fault rates where two out of three bits are faulty but both are 0 or 1 will lead to wrong results as it is illustrated in Figure 4. This can manifest in silent data corruption, as it can be seen in Figure 3 for a fault rate of 10^{-6}.

To overcome such a problem a higher redundancy may be a solution. However, already TMR comes along with a high overhead for detection – even for low fault rates leading to higher runtimes. Methods applying higher redundancy techniques are hence even more time consuming.

Another option is the defect correction method, which we use here. Based on the obtained results defect correction with CG as inner solver delivers similar fault tolerance as with CG with TMR for medium fault rates. In addition, due to its mathematical properties the defect correction method is "immune" against silent data corruption (undetected errors). In contrast the original CG and CG with TMR may result in silent data corruption.

For the original CG, silent data corruption is not a major problem for high fault rates. In this case the fault rate is that huge, that the CG method never satisfies its stopping criterion. Hence no silent data corruption occurs, no matter which solver is applied. However, if the fault rate is in a medium range the stopping criterion of the CG method is fulfilled in most of the cases, but the calculated solutions can be faulty. If there is no additional check for the solution as explained before, the result data can contain unrecognized faults. Thereby, the measured peak value was an occurrence rate of 58 % for silent data corruption for a fault rate of 10^{-9} for the original CG method. With TMR the vulnerability for silent data corruption is heavily reduced.

However, also defect correction is not the "holy grail". As illustrated in Figure 3, for some very high fault rates CG with TMR delivers more often a correct solution than the defect correction method does (in our case for a fault rate of 10^{-6} to 10^{-8}). This is due to the fact, that TMR can detect and correct the occurring errors on-the-fly, while in

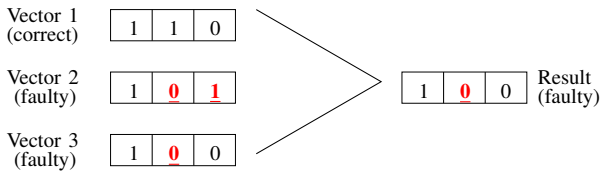

Figure 4. TMR can not correct always correct faults

the defect correction method the faults are only implicitly corrected by updating the solution and afterwards the residual r_{new}, and then start the inner solver (here: CG) again to solve $Ad = r_{new}$ (see Algorithm 2). If bit flips lead to an abortion of the inner solver the solution is not updated in the classical defect correction method. By this means, the residual remains the same and the following call of the inner solver tries to solve the same problem as before. Hence, this can lead to endless loops, which are counted as aborts in our categorization (runtime $> 20\times$ runtime of pure CG).

One possible solution to reduce the number of aborts due to endless loops is a dynamic backup of data (flexible checkpointing) during the run of the inner solver as proposed in Section V-C. While in the classic defect correction scheme, an abort of the inner solver leads to the problem that the computed data is thrown away, in the proposed enhanced version only the data of the last iterations of the inner solver is thrown away and an intermediate backup is used to update the solution. As the results in Figure 3 clearly show, the advantage compared to the classic defect correction scheme is huge. In our scenario the enhanced version could still compute the correct solution in a reasonable amount of time for a fault rate of 10^{-8}, for which the classic version was struggling. Admittedly, also this approach is not feasible for huge fault rates (here: more than 10^{-8}), under the used runtime constraints.

D. Runtime of CG, CG with TMR and Defect Correction

Beside the pure fault tolerance of the different solvers, their practical runtime is a very important property. In Figure 5 the average runtime for the computation of a correct solution for the three different approaches is depicted. Please note that since the standard CG solver and the defect correction scheme can not provide a correct solution within 120 seconds (i.e. 20 times the runtime in the fault free case) for a fault rate of 10^{-8} their runtime results are not presented for this fault rate.

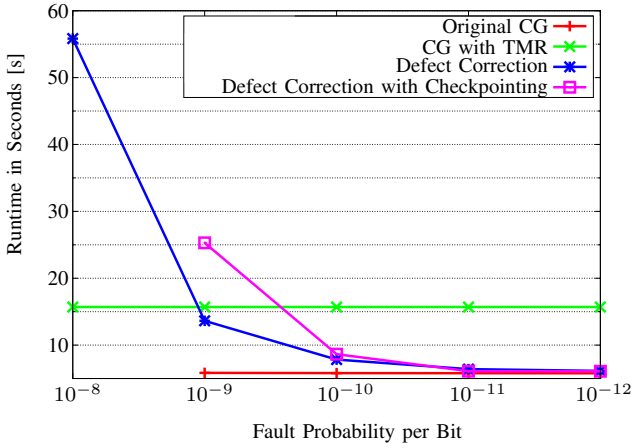

Figure 5. Average runtime for the different fault tolerance approaches for correct computed solutions

Due to the triplication of the data and the computations, the TMR approach comes along with a huge and constant runtime overhead. Figure 5 shows a performance overhead of a factor 2.7 which underlines the theoretical factor 3 deduced from the costs analyzed in Table I. Especially for low and medium fault rates this technique is hence not the best solution. Here, the defect correction method (with dynamic checkpointing) is clearly the better choice. Furthermore, the results prove, that the runtime overhead of the defect correction method for fault rates up to 10^{-11} compared to a fault free CG is negligible. However, if the fault rates exceed a certain value, the tide is turning. Since TMR can detect and correct faults on-the-fly no additional iterations are necessary to compute the correct solution. In contrast, the defect correction method needs additional steps, which means that the inner solver is called multiple times, and hence needs more time to calculate the correct solution. In this case, checkpointing helps to further reduce the average runtime. However, please note that TMR is only better (in terms of runtime and fault tolerance) than defect correction for extremely high fault rates, which might be unrealistic for current technologies. Therefore, one can conclude that a combination of the defect correction method with checkpointing and TMR is optimal, whereby TMR is only activated for a very high fault rate.

As already mentioned in Section V-A, the defect correction scheme contains two parts, that contribute to the runtime: first the number of outer loops and second the number for CG iterations. The number of outer iterations in the correction scheme depends on the error injection rate. For 0 injected faults (error probability = 10^{-12}), the outer loop is performed a single time and the inner CG loop has 52 iterations for our model problem. In this scenario, there is no overhead from error correction. For an error probability of 10^{-11} in average 0.2 faults are injected. In that case, the outer loop is still run just a single time, whereas for an error probability of 10^{-10} (i.e. 4.12 injected faults in average) the outer loop is performed 1.58 times in average. The total number of CG loops in this scenario is 75.66 in average leading to an overhead compared to the fault free case with the standard CG of a factor of 1.47. If the error rate is increased to 10^{-9} the average number of injected faults raises to 114, leading to 3.7 executions in average of the outer loop and 223 CG loops. So the error correction overhead is a factor of 4.3 in terms of additional work. This overhead can be seen in the runtime figure presented in Figure 5.

VII. Conclusion

In this paper we presented an Algorithm-based Fault Tolerance (ABFT) scheme for an iterative linear solver based on the Conjugated Gradient method (CG) by taking advantage of numerical defect correction. In our proposed method,

errors, due to hardware failures or external disturbances, are treated as numerical defects and by that means handled by the numerical defect correction method, which uses CG as its solving engine.

Furthermore, we enhanced our method with dynamic checkpointing when the range of numerical defects goes into infinity. Thereby, the inner solver does not throw away its computed results but instead just goes back to the previous checkpoint. The checkpointing steps are set dynamically based on recent history of retries.

Our experimental results based on fault injection on various fault rates and comparison with software implemented hardware fault tolerance (SIHFT) using Triple Modular Redundancy (TMR) at instruction level clearly shows the benefits of the proposed method compared to such brute-force SIHFT-TMR method. The runtime overhead of our proposed method is "pay as you go", meaning that there is practically only a runtime overhead when errors have to be corrected with additional iterations, which is only the case for very high fault rates. However, the runtime overhead gets larger with increasing fault rate. This is in contrast with TMR in which there is a "prepaid cost" even if no errors occur. Finally, our proposed approach always guarantees data integrity (i.e. there is no silent data corruption) combined with good fault tolerance, unlike TMR.

ACKNOWLEDGEMENTS

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and by the industrial collaboration partner Hewlett-Packard.

REFERENCES

[1] International Technology Roadmap for Semiconductors, http://www.itrs.net, 2009.

[2] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," *Commun. ACM*, vol. 54, pp. 100–107, February 2011.

[3] C. Chen and M. Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications," *IBM Journal of Research and Development*, vol. 28, no. 3, pp. 124–134, March 1984.

[4] N. Kurd, S. Bhamidipati, C. Mozak, J. Miller, T. Wilson, M. Nemani, and M. Chowdhury, "Westmere: A family of 32nm IA processors," in *IEEE International Solid-State Circuits Conference*, February 2010, pp. 96–97.

[5] D. Siewiorek and R. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3rd ed. A. K. Peters Ltd, December 1998, ISBN: 9781568810928.

[6] K. Böhmer, *Defect Correction Methods: Theory and Applications*, ser. Computing: Supplementum, 5. Wien [u.a.]: Springer, December 1984, ISBN: 9780387818320.

[7] O. Axelsson, *Iterative solution methods*, 1st ed. Cambridge [u.a.]: Cambridge University Press, March 1996, ISBN: 9780521555692.

[8] J. S. Plank, K. Li, and M. A. Puening, "Diskless Checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, pp. 972–986, October 1998.

[9] Z. Chen and J. Dongarra, "Algorithm-Based Checkpoint-Free Fault Tolerance for Parallel Matrix Computations on Volatile Resources," in *Proceedings of the International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, April 2006, pp. 97–97.

[10] T.-C. Chiueh and P. Deng, "Evaluation of Checkpoint Mechanisms for Massively Parallel Machines," in *Proceedings of the International Symposium on Fault-Tolerant Computing*, ser. FTCS '96. Washington, DC, USA: IEEE Computer Society, June 1996, pp. 370–379.

[11] R. A. Rubinfeld, "A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs," Ph.D. dissertation, Berkeley, CA, USA, 1991, UMI Order No. GAX91-26752.

[12] K.-H. Huang and J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. 33, pp. 518–528, June 1984.

[13] J. Anfinson and F. T. Luk, "A Linear Algebraic Model of Algorithm-Based Fault Tolerance," *IEEE Transactions on Computers*, vol. 37, pp. 1599–1604, December 1988.

[14] Y.-M. Yeh and T.-Y. Feng, "Algorithm-Based Fault Tolerance for Matrix Inversion with Maximum Pivoting," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 373–389, April 1992.

[15] D. Boley, G. H. Golub, S. Makar, N. Saxena, and E. J. McCluskey, "Floating Point Fault Tolerance with Backward Error Assertions," *IEEE Transactions on Computers*, vol. 44, pp. 302–311, February 1995.

[16] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault Tolerant High Performance Computing by a Coding Approach," in *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, June 2005, pp. 213–223.

[17] E. Yao, M. Chen, R. Wang, W. Zhang, and G. Tan, "A New and Efficient Algorithm-Based Fault Tolerance Scheme for A Million Way Parallelism," *Computing Research Repository*, vol. abs/1106.4213, June 2011.

[18] R. Banerjee and J. A. Abraham, "Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems," *IEEE Transactions on Computers*, vol. 35, pp. 296–306, April 1986.

[19] 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," *IEEE Transactions on Computers*, vol. 39, pp. 1132–1145, September 1990.

[20] V. Balasubramanian and P. Banerjee, "Compiler-Assisted Synthesis of Algorithm-Based Checking in Multiprocessors," *IEEE Transactions on Computers*, vol. 39, pp. 436–446, April 1990.

[21] G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. Dongarra, "Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems," in *Proceedings of the International Supercomputer Conference (ICS)*, 2004.

[22] A. Roy-Chowdhury, N. Bellas, and P. Banerjee, "Algorithm-Based Error-Detection Schemes for Iterative Solution of Partial Differential Equations," *IEEE Transactions on Computers*, vol. 45, pp. 394–407, April 1996.

[23] J. W. Demmel, *Applied Numerical Linear Algebra*. Philadelphia: SIAM, August 1997, ISBN: 9780898713893.

[24] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, April 2003, ISBN: 9780898715347.

[25] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, 1985.