Performance Evaluation of Checksum-Based ABFT*

Ahmad A. Al-Yamani, Nahmsuk Oh and Edward J. McCluskey
Center for Reliable Computing – Stanford University, Stanford, California
{alyamani, nsoh, ejm@crc.stanford.edu}

Abstract

In Algorithm-based fault tolerance (ABFT), fault tolerance is tailored to the algorithm performed. Most of the previous studies that compared ABFT schemes considered only error detection and correction capabilities. Some previous studies looked at the overhead but no previous work –as far as we know– compared different recovery schemes for data processing applications considering throughput as the main metric. In this work, we compare the performance of two recovery schemes: recomputing and ABFT correction, for different error rates. We consider errors that occur during computation as well as those that occur during error detection, location and correction processes. A metric for performance evaluation of different design alternatives is defined. Results show that multiple error correction using ABFT has poorer performance than single error correction even at high error rates. We also present, implement and evaluate early detection in ABFT. In early detection, we try to detect the errors that occur in the checksum calculation before starting the actual computation. Early detection improves throughput in cases of intensive computations and cases of high error rates.

1. Introduction

A major concern in fault tolerant systems design is to provide the desired fault tolerance within the available cost, power consumption, performance constraints, etc. It has been shown that in certain matrix applications, low overhead fault tolerance can be achieved using ABFT [1] [2].

Errors occur with different rates depending on the environments where computing systems are operated. For example, satellites experience error rates based on their altitude and location. This variation makes different fault tolerance schemes more appropriate in different environments. The recent increase in computationally intensive data processing applications makes it essential to protect the computation units such as ALUs against transient errors [3]. Performance is a critical metric for these data processing applications. Errors influence the rate at which these applications run. The influence of errors on the performance of these systems depends on the recovery scheme applied. ABFT was invented for providing low-overhead recovery in data processing applications. In this paper we model checksum ABFT for matrix operations considering transient faults in hardware. We consider three matrix operations: matrix multiplication, LU decomposition and matrix inversion.

The main contributions of this work are: 1) A performance-based comparison of error recovery by recomputing and by ABFT correction, for various error rates. Such a performance comparison study is essential for data processing applications for which ABFT is a suitable fault tolerance scheme, 2) An evaluation of the effect of correction capability (code distance) on performance. In this evaluation, we show by simulation that multiple error correction has a negative impact on performance even for high error rates, 3) Definition of a metric for performance evaluation of fault tolerance schemes for data processing applications, and 4) An early detection scheme for ABFT is presented and its impact on performance is analyzed.

In Sec. 2, Checksum-based ABFT is explained. In Sec. 3, we review the related literature. Section 4 discusses fault injection. In Sec. 5 we present the experiments and results. Section 6 is the conclusion.

2. Checksum-Based ABFT

ABFT refers to a class of techniques that are based on tailoring fault tolerance to the algorithm performed. It can be tuned to provide the desired fault tolerance e.g., single error detection, single error correction, etc. For

* This work was supported by King Fahd University of Petroleum and Minerals through the Saudi Arabian Cultural Mission to USA. It was also supported in part by the National Aeronautics and Space Administration and administered through the Jet Propulsion Laboratory, California Institute of Technology under Contract No. 1216777.

some computations, ABFT can be implemented with low overhead as shown in [1] where the redundancy ratio was shown to decrease linearly as the matrix dimension was increased. In [2], it was shown that using ABFT for solving a linear system of equations, error detection required 13.1% memory overhead and 32% execution time overhead. Single error correction required 28% memory overhead and 73% execution time overhead.

Checksums for error control in matrix operations was presented for linear computations in [4]. ABFT was presented in [1] where checksums were given as an example. Jou et al. enhanced ABFT with the weighted checksums for better detection and correction capability [5]. Anfinson et al., explained the algebraic interpretation of ABFT for matrix operations with definitions for distance, code space, etc. [6].

In the weighted checksum approach, *d* rows (columns) are added to the original matrix [5]. For an n×n *A* matrix, define *d* linearly independent n×1 weight vectors $w^{(i)}$, *i = 1, 2, … d*, with elements $w^{(i)}_j$, *j = 1, 2, …, n*.

Definition 2.1: Define an n×(n+d) weights matrix $W = \begin{bmatrix} 1 & 0 & \cdots & 0 & w_1^{(1)} & w_1^{(2)} & \cdots & w_1^{(d)} \\ 0 & 1 & \cdots & 0 & w_2^{(1)} & w_2^{(2)} & \cdots & w_2^{(d)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & w_n^{(1)} & w_n^{(2)} & \cdots & w_n^{(d)} \end{bmatrix}$

The following terms are defined as follows: weighted row checksum matrix $A_{rw} = A \times W$, weighted column checksum matrix $A_{cw} = W^T \times A$ and weighted full checksum matrix $A_{fw} = W^T \times A \times W$

Definition 2.2: Define a d×(n+d) code matrix $H = \begin{bmatrix} w_1^{(1)} & w_2^{(1)} & \cdots & w_n^{(1)} & -1 & 0 & \cdots & 0 \\ w_1^{(2)} & w_2^{(2)} & \cdots & w_n^{(2)} & 0 & -1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_1^{(d)} & w_2^{(d)} & \cdots & w_n^{(d)} & 0 & 0 & \cdots & -1 \end{bmatrix}$

The null space of *H* is *N(H) = {x : Hx = 0}*. A column of A_{cw} (call it *x*) is error-free if it belongs to the null space of *H*. If every *d* columns of *H* are linearly independent, up to *d* errors can be detected and up to $\lfloor d/2 \rfloor$ errors can be corrected in every column (row) of A_{cw} (B_{rw}). Let the syndrome vector be $s = Hx$ ($x^T H^T$).

An error-free vector *x* generates a 0 syndrome while an erroneous vector $\hat{x}$ generates a non-zero syndrome. Let the correction vector be $c = \hat{x} - x$, $Hc = H(\hat{x} - x) = s$. By solving the linear system of equations, *c* can be obtained. This system cannot be used with more than $\lfloor d/2 \rfloor$ equations that correspond to the erroneous elements. A problem in this process is that it is unknown to the correction module which $\lfloor d/2 \rfloor$ elements are erroneous. This means that it might try as many as $C^{n+d}_{\lfloor d/2 \rfloor}$ possibilities for each erroneous vector in the worst case.

3. Previous Work

Hudak et al., compared several fault-tolerant software techniques including ABFT based on error coverage [7]. The techniques were ranked based on MTTF and cost. Prata et al. compared ABFT with result checking based on error coverage, overhead and ease of use [8]. In [2], ABFT for detection only was compared to detection and correction in terms of error coverage, memory size and execution time.

Beaudry defined computation capacity as a performance metric in order to study the interaction between performance and reliability [9]. She incorporated computation capacity in reliability modeling of redundant systems. She defined *computation capacity* as the amount of useful computation per unit time available on the system. Since she was evaluating gracefully degrading systems, she defined computation capacity as a function of the system state. For our evaluations, we define computation capacity as a function of the error rate.

In [10], Model-based evaluation of performability, which is a measure that combines performance and dependability, was used to improve the effectiveness of fault tolerant software (N-version programming). In their model, the program accepts inputs at the beginning of each iteration of its execution and provides output that is a function of the most recent input. A decision function determines if the output of an iteration is erroneous. They quantified performability by the number of successful iterations during some bounded time interval. Their aim was to demonstrate the feasibility of performability modeling.

The work presented here is unique in that it evaluates the performance of different recovery techniques executing in the presence of errors. The error rate and the recovery scheme used have a direct impact on the rate at which the algorithm produces correct results. We compare the performance of (1) Error recovery by recomputing and (2) ABFT correction considering *computation capacity*. Such a study is essential for data processing applications in environments that introduce transient errors like single event upsets (SEUs) [11]. An example for such applications is the REE project where science applications are supposed to run in space (a radiation environment) [3]. Besides fault tolerance, high performance is a critical metric for such a system. This study is important for the design of such systems since it compares recovery schemes based on performance.

4. Fault Injection

The fault model assumed in [1] and [5] was a malfunction of a processor in a processor array. The manifestation was that one or more elements of the resultant matrix could be erroneous. The ABFT experiments performed in [2] used FERRARI to emulate transient bit flips at execution time within the address space of a program. In [12], the fault model used was a bit flip in a data entry during the execution. Hudak et al. used FIAT to inject faults by changing a memory location (code or data) [7]. In [8], Xception was used for emulating transient faults in ALUs, data and address busses, general-purpose registers, condition registers and memory.

In this work the fault model assumed is a bit flip in the memory, the cache, a register, or in the ALU. Injecting faults in the code segment will cause one of the following cases: 1) the program will crash due to illegal operations, 2) the program will malfunction due to a control flow error or a status register error or 3) the program will perform an instruction different from the original one and so will produce an erroneous result. For cases 1 and 2, the schemes we are comparing will not work properly since ABFT does not detect these errors. For case 3, one or more entries will change in the result, which is equivalent to an error in the data segment.

The distribution of the number of errors generated is Poisson(λts), where t is the time elapsed and s is the target matrix size. In other words, the error rate (λ) is the mean number of errors per unit time per floating point number. After choosing the number of errors to be injected, data entries are chosen uniformly for fault injection. Fault injection is performed by making a random change in the chosen entry. For the ABFT schemes applied, there is no difference between a bit flip and a random change since they both will cause the syndromes to have non-zero values in the same positions, and the location and correction processes are similar in both cases.

Based on the classification given for FERRARI, our fault injection scheme covers transient faults in the data bus and the address bus while fetching and storing operands. It also covers computation unit faults as well as storage (GPRs, cache and memory) faults in the data portion of the program.

5. Experiments and Results

Three ABFT algorithms: matrix multiplication, LU decomposition and matrix inversion were implemented in C with matrices of floating point numbers. We used different values for d, which also decides the correction capability. For encoding we use the standard H matrix from [6], $w_j^{(i)} = \beta^{(i-1)(j-1)}$.

Since the errors are assumed to be transient, two recovery approaches are possible: 1) Recomputing upon error detection, and 2) Correction using the syndrome matrix (ABFT correction), see Sec. 2. Recomputing is a rollback scheme that requires only error detection, which costs less than correction in terms of memory overhead. On the other hand, it might cost more in terms of execution time depending on the application and the error rate. ABFT correction is a rollforward technique that costs more memory overhead. It might cost less in execution time depending on the application and the error rate. In case ABFT correction is unable to correct the errors, it also recomputes. Both schemes are implemented and their performance is compared.

LU decomposition is the most computationally intensive process in solving linear systems of equations. A square matrix is generated randomly using a uniform distribution and then decomposed. To verify correctness of the result, we check if it is a solution for the system of linear equations or not.

Matrix inversion is an example for a computationally intensive operation. In [12], Shultz method was used for ABFT matrix inversion to preserve checksum encoding. A square matrix A is generated randomly and

inverted. To verify the correctness of the result, we multiply A^{-1} by A and subtract the result from the identity matrix I. If the subtraction result is non-zero then the result is erroneous.

In our simulation experiments, each algorithm runs for 100 iterations with every combination of (matrix size, code distance and error rate). An *iteration* corresponds to running the algorithm once on input data. Every time an iteration produces an erroneous output the total number of iterations is incremented up to 200.

Definition 5.1: Success Ratio (*SR*) is defined as the fraction of iterations in which correct results were produced.

Definition 5.2: Completion Time (*CT*) is defined as the time it took the program to complete a fixed number of iterations. This time includes the time spent on those iterations that did not produce correct results.

Definition 5.3: Computation Capacity (*CC*) is defined as the number of successful iterations per unit time with a certain error rate. Based on this definition, the following formula can be used to compute computation capacity:

$$CC = \frac{SR \times Total\ Number\ of\ Iterations}{CT}$$

Definition 5.4: By *early detection (ED),* we mean checking for errors after computing the checksums and before starting the actual computation. This checking is performed using the same checksum detection matrix '*H*' used for ABFT detection. We implement early detection and evaluate its effect on *computation capacity*. Its rationale is to avoid running intensive computations in the presence of uncorrectable errors.

In these experiments we used matrix sizes (10×10 to 50×50) of floating point numbers. The error rates used were (10^{-9} to 10^{-3}) errors per entry per second. To avoid round off errors in floating point computations we used a range of 10^{-6} within which two numbers were considered equal. In the Stanford Advanced Research and Global Observations Satellite (ARGOS) project, in which Commercial Of-the-Shelf (COTS) components where tested in space using Software Implemented Hardware Fault Tolerance (SIHFT), the number of errors observed in the memory was around 10 errors per megabyte per day (≈9.25×10^{-10} errors per entry per second). We use this as well as higher error rates in our experiments. With all error rates we used, multiple error correction did not show good performance compared to single error correction or recomputing.

5.1 LU Decomposition (LUD)

Figure 1 shows a comparison between recomputing and ABFT correction for LU decomposition for various correction capabilities. The label (*i* EC) corresponds to correcting *i* errors. The error rate in the figure starts at 10^{-7} because the computation capacities for smaller error rates were the same as those for 10^{-7} i.e., the lines extend horizontally for smaller error rates. Recomputing had the highest computation capacity for all error rates. This is due to the complicated correction process with more than a single checksum row and column. Correction can't be applied with a single checksum row and column since L and U matrices are not full checksum matrices.

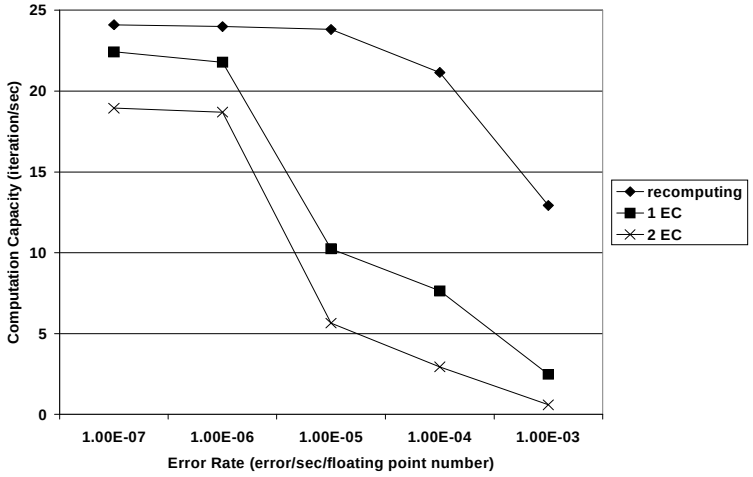

Figure 1: Computation capacity of recomputing and ABFT correction with different code distances (LUD, 40×40)

Figure 2 shows the effect of early detection on computation capacity. Under high error rates early detection improves the computation capacity for all schemes because it reduces the latency of detection. At low error rates the loss in computational capacity was small. An interesting observation from the figure is that early detection might make correction a favorable scheme. The figure shows that without early detection, the computation capacity of recomputing is higher than that of single error correction. By applying early detection, single error correction turns out to have higher computation capacity than recomputing at high error rates.

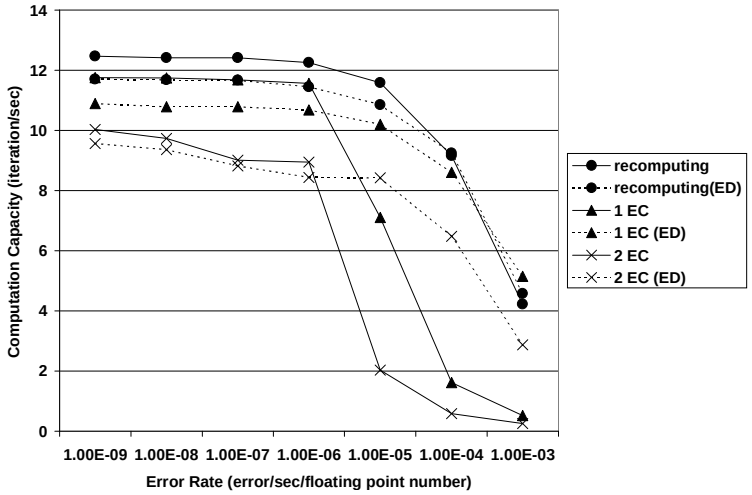

Figure 2: Effect of early detection on computation capacity (LUD, 50×50)

5.2 Matrix Inversion (MI)

Figure 3 shows the computation capacity of recomputing and ABFT correction for matrix inversion with various correction capabilities. It shows that at low error rates, single error correction had high computation capacity. At high error rates, recomputing had higher computation capacity.

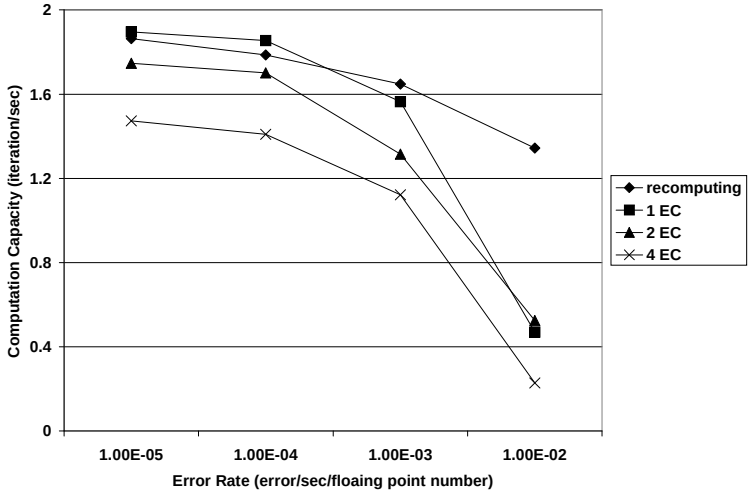

Figure 3: Computation capacity of recomputing and ABFT correction with different code distances (MI, 40×40)

Figure 4 shows the effect of early detection on computation capacity. Since matrix inversion is more computationally intensive than multiplication and decomposition, early detection shows improvement in computation capacity for all error rates.

The results for matrix multiplication are omitted for space limitations.

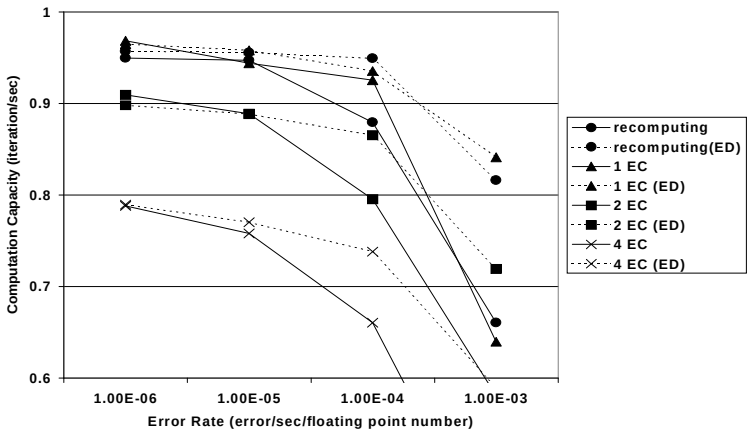

Figure 4: Effect of early detection on computation capacity (MI, 50×50)

6. Summary and Conclusions

Our simulation shows that multiple error correction in checksum-based ABFT is inefficient in terms of both completion time and success ratio for the applications we evaluated.

By comparing the schemes based on computation capacity, single error correction was the best scheme for matrix multiplication. Recomputing was the best scheme for LU decomposition. For matrix inversion, recomputing was better for high error rates and correction was better for low error rates. Also early detection was beneficial for multiplication and decomposition only at high error rates. However, it was beneficial for inversion under all error rates due to the computationally intensive nature of the algorithm.

In conclusion, error rate had a direct impact on performance. Recomputing and single error correction with checksum-based ABFT had close computation capacities. Multiple error correction had worse computation capacities than recomputing and single error correction even with high error rates. The results shown here are for the given matrix applications using checksum-based ABFT. For different matrix algorithms or different fault injection schemes, results might change and that is a subject for further investigation.

References

[1] K. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations", *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 518-528, June 1984.

[2] R. K. Acree, Nasr Ullah, A. Karia, J. T. Rahmeh and J. A. Abraham, "An Object-Oriented Approach for Implementing Algorithm-Based Fault Tolerance", *12th Annual International Phoenix Computers and Communications Conference* , pp. 210-216, Mar. 1993.

[3] R. R. Some and D. Ngo, "REE: A COTS-Based Fault Tolerant Parallel Processing Supercomputer", *IEEE Digital Avionics Systems Conference*, Oct 1999.

[4] Paul S. Dwyer, *Linear Computations,* John Wiley & Sons, 1951.

[5] J. Jou and J. Abraham, "Fault-Tolerant Matrix Arithmetic and Signal Processing on Highly Concurrent Computing Structures", *Proc. IEEE*, vol. 74, pp. 732-741, May 1986.

[6] C. Anfinson and F. Luk, "A Linear Algebraic Model of Algorithm Based Fault Tolerance", *IEEE Transactions on Computers,* Vol. 37, No. 12, pp. 1599-1604, Dec. 1988.

[7] J. Hudak, B. Suh, D. Siewiorek and Z. Segall, "Evaluation & Comparison of Fault-Tolerant Software Techniques", *IEEE Transactions on Reliability*, Vol. 42, No. 2, June 1993.

[8] P. Prata and J. Silva, "Algorithm Based Fault Tolerance Versus Result-Checking for Matrix Computations", *International Symposium on Fault Tolerant Computing FTCS-29*, pp. 4 – 11, June 1999.

[9] M. D. Beaudry, "*Performance Considerations for the Reliability Analysis of Computing Systems"*, PhD Thesis, Stanford Univerity, Electrical Engineering, 1978.

[10] A. T. Tai, J. F. Meyer and A. Avizienis, "Performability Enhancement of Fault-Tolerant Software", *IEEE Transactions on Reliability*, Vol. 42, No. 2, June 1993.[11] J. Beahan, L. Edmonds, R. Ferraro, A. Johnston, D. Katz and R. Some, "Detailed Radiation Fault Modeling of the Remote Exploration and Experimentation (REE) First Generation Testbed Architecture", *IEEE Aerospace Conference*, pp. 279-281, March 2000.

[12] E. I. Milovanovic, I. Z. Milovanovic, M. K. Stojcev and G. S. Jovanovic, "Fault-Tolerant Matrix Inversion on Processor Array", *Electronic Letters,* pp. 1206-1208, June 1992.