Fault Tolerance Using Algorithm-Based Data-Word Recovery

Goutam Kumar Saha
Centre for Development of Advanced Computing, Kolkata, India
CA – 2 / 4 B, CPM Party Office Road, Baguiati, Deshbandhu Nagar,
Kolkata- 700059, West Bengal, INDIA
E-mail: gksaha@rediffmail.com

Abstract

This paper examines how to design a low-cost and algorithm-based approach that recovers random multiple bit – errors in an application's data-words on memory during the execution time of an application. This is a low cost and an effective software technique in order to detect and recover an application's sensitive data elements using an affordably lower redundancy in both time and space. This is a practical approach towards gaining fault tolerance and dependable computing through recovery or corrections of multiple bit-errors in memory variables as well.

Key Words: Data–word error detection and recovery, fault tolerance, transient bit errors.

1. Introduction

Transient or soft faults in processor main memory and registers are becoming higher and higher due to increased complexity, higher clock speeds, lower power consumption requirements and smaller transistor sizes. In the low-cost schemes proposed in this paper, the error detection and recovery (of a data-word) checks for faults periodically instead of every time step, each time detecting and identifying errors due to transient state-transition faults that may have occurred since the last check. Fault Tolerance is the ability of a system to perform its specified function correctly even in the presence of internal faults or errors. The purpose of fault tolerance is to increase the dependability of an application system. Faults can be classified as transient or permanent faults. A permanent fault will remain unless it is removed by some external agency. While it may seem that permanent faults are more severe, they are much easier to diagnose and handle from an engineering perspective. A transient fault will eventually disappear without any apparent intervention. Particularly a problematic type of transient fault is the intermittent fault that recurs, often unpredictably. Many real-life scientific and engineering applications use stored information in the form of many memory variables, and those data need to be safe for the life-time of the applications. The correctness of computed answers depends on how much the stored data are safe and correct. The electrical transients often corrupt the stored information. There is a need to detect the errors continually in various data elements that are being referred by a running computing application. This is to ensure the correctness of computed results. This is one of the primary objectives of a real time system. Potential transients often corrupt the information bit patterns that reside on the primary memory. Many online real-life application programs use such memory multi-word variables to store their reference information that are referred by intermediate processing steps for computing answers. For example, a computer controlled missile tracking system initiates a tracking function that is based on true matching of the received online signal frequency, pulse width and pulse repetition interval with its
corresponding stored information on memory. Transients often alter the stored bit-patterns randomly and such corruptions are independent to each other. Therefore, an online multi-words faults detection and recovery thereof must be carried out within a very short duration of time for gaining higher system safety. This is a very challenging task. The objective of this proposed approach is to gain a faster fault detection and faster recovery thereof, without too many rereading from stable storage. Conventional error codes have limitations over faster corrections of multiple and random bit errors on multiple words. There are several conventional error detection and error correction schemes like Parity Checks, Hamming Codes, Cyclic Redundancy Checks, Checksums etc. They are not free from limitations (Rhee, 1991; Wicker, 1995). The single parity checks can detect only odd number of errors. Any even number of errors is undetected. So it is inadequate for reliable detection of errors; in many situations, it only detects errors in about half of the encoded strings where errors occur. Again for Cyclic Redundancy Checks (CRC), shift register based circuits are used. CRC is normally implemented by hardware. The shift register circuit is for dividing polynomials and finding the remainder. Modulo 2 adders, Multiplier circuits are also needed. In CRC, when errors actually occur, the receiver fails to detect the errors when the remainder is 0. This occurs if the error sequence represented as a polynomial e(D), is itself some code word. Apart from this limitation, CRC are having high time redundancy and that is why they are normally hardware based. Software based CRC implementation is impractical in a real time application because of its higher time redundancy. Again, a Hamming code is to provide only the single-error correction and double-error detection. In a typical Checksum where n bytes are XORed and the result is stored in (n+1) th byte. Now if this byte itself is corrupted due to noises or in the case of even changes, the errors remain undetected by this typical Checksum. Thus this Checksum does not detect even changes. Thus all the above mentioned methods are not free from limitations and there exists higher redundancy in both time and memory space. Interested readers should refer to (Bertsekas and Gallayer, 1989; Huang and Abraham, 1984) for further related information. Traditionally, a single erroneous element can be corrected by using a full checksum matrix (Huang and Abraham, 1984). The procedure of the full checksum (for error detection) can be described as below.

Step 1. Add the information elements in each row.

Step 2. Add the information elements in each column.

Step 3. Compare each calculated sum to the corresponding checksum in the summation-vectors namely row checksum vector and column checksum vector.

Step 4. Find the inconsistent row, which is detected by an inequality in step-3.

Step 5. Find the inconsistent column, which is detected by an inequality in step-3.

In this full checksum technique an error is located at the intersection of the inconsistent row and inconsistent column. The erroneous element can be corrected (a) by adding the difference of the computed sum of the row or column data elements and the checksum to the erroneous element in the information part, (b) or by replacing the checksum by the computed sum of the information elements in the summation vector, in the case where the checksum is incorrect. Thus this method is capable of detecting, locating and correcting only a single erroneous element in a full checksum matrix. When there are multiple erroneous elements in a row or column of a matrix, this full checksum method is an invalid one. The full checksum matrix procedure is used to detect, locate, and correct a single erroneous element only. It is effective only for
the case of one erroneous element in a full checksum matrix. Here, it is valid for exactly one row and one column have an incorrect checksum. The intersection of this row and column locates the erroneous matrix element.

Again a method of full data duplication (Saha, 2000) is capable of detecting multiple errors only. It cannot correct the erroneous data without a reread from an immutable or stable storage. In this technique errors are detected by comparing the corresponding bytes or words of both the original and duplicated data.

A failure is said to occur when a system does not meet its specifications (Anderson and LEE, 1981). A valid system is one which the system operates in accordance with the specifications. An erroneous state of a system is an internal state, which could lead to failure despite a sequence of valid transitions. An error triggers a transition from normal state to an erroneous. An error in hardware or software component of the system is referred to as a fault in the system. A transient fault is one, which is not reproducible under controlled conditions. A permanent fault is one that requires corrective action for its removal, and, restoration of normal system operation. The proposed technique is capable of detecting erroneous multi-word values or data elements as soon as possible by means of verification and the erroneous task is halted in order to take recovery actions. Thus it prevents error propagation. After the corrections of erroneous look-up values, the application program execution continues. This approach of fault tolerance is often termed as fail-fast (McGarry and Rajerski, 1990). Interested readers may refer to (Green and Souce, 1984; Saha, 2003; 2004; 2005) also for further related information. This technique does not need multiple versions of software based on design diversification. Thus this proposed software technique is also a very useful tool for the system engineers in fixing fault tolerance while designing an application that uses many memory variables.

2. The Algorithm

The proposed approach recovers the multiple errors through multiple-bit error corrections. Here, error corrections are carried out using various safe guard variables. This approach does not use replicas of variables that often get randomly corrupted by potential electrical transients. In this approach various safe guard variables are used for guarding the sensitive look-up variables. For simplicity, if an application uses say, two sensitive look-up variables namely, \(A\) and \(B\), then their corresponding safe guard variables will be say, \(S_{AB}\) and \(D_{AB}\). The values of \(A\) and \(B\) are initialized on reading from a stable or immutable storage. The value of \(S_{AB}\) is initialized to \((A + B)\) and \(D_{AB}\) is initialized to a value of \((A - B)\). All these variables are declared to be global or public ones. The symbols “/* */” are to enclose the remarks. The following steps are to describe the processing logic of the proposed technique that guards two variables.

```plaintext
/* INITIALIZATION */
global: A, B, S_AB, D_AB /* variables are declared as global */
PROCIN
Read A, B from stable storage. /* initialize the look-up variables */
Set S_AB = A + B /* initialize the safe guard variables */
Set D_AB = A – B
Return
End of PROCIN
/* The routine PROCEDR is for data-words-bit-errors-detection and corrections */
PROCEDR
If ((A ≠ ( S_AB + D_AB ) / 2.0) .and. (B = ( S_AB – D_AB ) / 2.0)), Then:
    Reset A = ( S_AB + D_AB ) / 2.0 /* corrupted A is recovered */
Else If ((A = ( S_AB + D_AB ) / 2.0)
Then:
Reset $B = (S_{AB} - D_{AB}) / 2.0$
/* corrupted $B$ is recovered */
Else If $(A + B \neq S_{AB}) \text{ and } (A - B \neq D_{AB})$, Then:
Reset $S_{AB} = A + B$
/* corrupted $S_{AB}$ is recovered */
Else If $(A + B = S_{AB}) \text{ and } (A - B = D_{AB})$, Then:
Reset $D_{AB} = A - B$
/* corrupted $D_{AB}$ is recovered */
Else If $(A + B \neq S_{AB}) \text{ and } (A - B \neq D_{AB})$, Then:
/* User's variables as well as the safe guard variables are all corrupted */
Call PROCIN /* four faults at the crash, so reinitialize the variables */
End If
Return
End of PROCEDR

3. Discussion on the Proposed Approach

From the above mentioned steps, it is certain that this proposed scheme will detect and correct all single – bit and all multiple–bit errors in $A$, $B$, $S_{AB}$, and $D_{AB}$ with high probability. Even at the worst case of crashing when all these four values have been randomly altered or corrupted unintentionally, the possibility of satisfying the equalities in the PROCEDR, with these changed values is almost nil. For example, if each of these variables is of say 16 bits long and, because the transient bit- errors are independent in nature, the possibility of failure to detect errors in this proposed model is $1/2^{48}$ or $2^{-48}$. This scheme demands only two reads from stable storage: one for initializing the variables and another one for reinitializing them at the time of disastrous consequence of crashing when all the look-up variables of an application program and their corresponding safe guard variables are all corrupted. The procedures namely PROCIN, and PROCEDR can be called or invoked for executing these codes from an application program during its run time with an appropriate modification of program code. At the beginning of an application program the PROCIN needs to be executed in order to initialize various look-up variables that need to be guarded. Before processing with the look-up variables, the procedure PROCEDR is invoked from an application program in order to check for possible errors in look-up variables and for recoveries thereof. This proposed model could be fixed easily to an application program with proper modifications in the program codes of the application, PROCIN, and PROCEDR using an additional design time. Again for an example, if an application program uses five variables say, $A,B,C,D,E$ then their corresponding safe guard variables will be say, $S_{AB}, D_{AB}, S_{CD}, D_{CD}, S_E$, and $D_E$, with initialized values: $S_{AB}=A+B$, $D_{AB}=A-B$, $S_{CD}=C+D$, $D_{CD}=C-D$, $S_E=E$, and $D_E=E$. Values for $E, S_E, D_E$ are corrected on resetting the corrupted one with the value in majority on comparison. If all these three variables are corrupted and they do not match with each other, then at such crash these variables can be reinitialized through a reread from an immutable storage. In this case both the procedures PROCIN and PROCEDR need to be modified accordingly. Instead of procedures, macros can be incorporated for faster errors corrections with an increased overhead with space redundancy because of the macro – expansion for each macro – reference, placed at various locations in the application code.

A trivial scheme using replica of various look-up variables can detect error on comparing the value of a variable with its replica's value but cannot recover the error without a reread from an immutable storage. Alternatively, the following model based on conventional Hamming code may satisfy the proposed scheme:

- Use a separate Hamming code for each bit position (i.e., the i–th code guards
the i-th bits of the elements).
- Compute the syndrome for each bit position.
- If all of the nonzero syndromes are equal (indicating that the errors are all in the same element) then correct those incorrect bits; otherwise reread all of the values from an immutable storage.

However, the above approach suffers from an unaffordable high time redundancy due to longer overhead with syndrome computation for each bit position and due to the requirement of more rereads from immutable storage. It is obvious that reading from immutable storage is a slower process. But this Hamming code based model requires less additional storage. Whereas the proposed model does not require a reread from immutable storage for correcting the values except for an event of crashing. Thus, this proposed approach or model is a faster method for an online detection and recovery of multi-word, multi-bit, and random bit errors using an affordable overhead with time and space. Again, the possible bit-errors, in the codes of these procedures and in the application program code, can be detected through a comparison of the computed checksum with a pre-stored checksum and when such mismatch occurs between these checksums, then the program control can be switched to another replica of the application, fixed with another replica of both the procedures namely PROCIN and PROCEDR. Thus bit errors in the program codes can also be tolerated. We have evaluated the algorithm on injecting random multiple-bit-errors at the elements of an array with 100 integer elements. Our success in error detection and recovery has been observed to be 97.2%. The rest has been found to be limited because of memory-overflow and truncation errors.

4. Conclusion

This paper has examined an algorithm-based technique that is useful for online detection and corrections of multiple and random byte-errors in data words. This technique uses an affordable redundancy with time and memory space. This is a low cost solution towards safeguarding various sensitive data elements during the life cycle of an application. However, system engineers must take care of possible overflow and round off errors while coding an application program based on this software model. This model demands an additional time in developing such model. However, multi-word error detection and correction thereof become faster because of not many rereads from an immutable storage, during the run time of an application. This model can be extended for recovery of an array also. Testing for such transient errors that are not reproducible is out of the scope of this paper. This model is also useful for designing a dependable computer controlled system that often looks up much stored information on primary memory during its life cycle for producing reliable answers.

References


