Software Based Fault Tolerant Computing Using Redundancy

Goutam Kumar Saha

CA –2 / 4B, CPM Party Office Road, Baguiati,
Deshbandhu Nagar, Kolkata 700059 WB India.
E-mail: gksaha@rediffmail.com

Abstract

This paper examines how software based fault tolerant computing approach through triplicate redundancy and recovery. This approach is not intended to tolerate the software design bugs. It is intended to tolerate various environmental faults during the execution time of a computer-controlled system. Application data corruption due to electrical transients is detected and recovered to control the system immediately. The proposed approach is a low cost tool towards designing a robust industrial application system that can tolerate errors due to electrical surges and transients. This approach does not rely on design diversification.

Keywords: Fault tolerant robust computing, recovery and application system.

1. Introduction

Electrical Noises, Electrical Transients (ETs), Electrostatic Discharge (ESD), Electromagnetic Pulses (EMP) are the example of short duration noises. Again short duration noises often cause random data corruption in the primary memory of a computing machine. Many scientific applications which need reference information tables are thus forced to miss their goals by using those corrupted data tables and program codes. Often we take it granted that our program code and data banks are absolutely safe and correct while designing software for an on-line application. But it is always not correct because the high speed processing units are often victimized by short duration noises as discussed in (Anderson and Lee 1981), (Dimitri 1989), (Wicker 1995). Electromagnetic Interference (EMI) is an unplanned, extraneous electrical signal that affects the performance of a computer system. It can cause memory errors and data file destruction during the run time of an application. Externally produced EMI or Noise enters the computer through the cabling or openings in the case. Sometimes it enters by static discharge through the case of the disk drive. Thus while designing software, the effects of noises should not be overlooked in a scientific application that uses look-up data during its execution time.

2. The Software Application

The look up table contains the angular field distribution for different regions of an aircraft. Algorithm 1., however shows the basic logic for determining if phase and amplitude of the received signal in a particular direction match a record in a predefined user look-up table [LT] for the angular distribution concerned. Depending on the result of this match, some action or function (say track) is performed.
So before initiating an action, the matching logic and processing should be of higher reliability and accuracy. If any transient causes data corruption or program code corruption then the whole system will be locked up leading to a complete mission failure.

Algorithm 1.

/* A predefined user data look up table [LT] contains records of information like PHI, PHASE, and AMPLITUDE. This algorithm shows the basic processing logic to find out a true match of PHI, PHASE and AMPLITUDE of received signal with predefined record in LT. Variables namely, FI, FASE, AMPL are to denote PHI, PHASE, AMPLITUDE respectively. */

Step 1. Read: Received signal FI, FASE, AMPL
Step 2. If FI .EQ. FI_LT, then:
/* Compare input parameters with the stored Parameters in the Look up Table */
   If FASE .EQ. FASE_LT .AND. AMPL .EQ. AMPL_LT, then:
      TRACK /*If matching then initiate Tracking */
   Else:
      GOTO Step 1. /* If not matching, then read inputs */
      [End of If structure]

   Else:
      GOTO Step 1. /* If not matching, then read inputs */
      [End of If Structure]

{End of Algorithm 1. showing basic processing logic}

The above mentioned basic processing logic is made robust with enhanced processing logic as shown in Algorithm 2.

Algorithm 2.
/* It shows the enhanced processing logic towards transient fault tolerance. Variables namely, PREVFI, PREVFASE, PREVAMPL are to store the earlier incoming signal’s PHI, PHASE and AMPLITUDE. A routine “VBYTFLT” is called from this enhanced logic for detecting transient errors in the program and data code and for the correction thereof periodically say, after every NRUN number of executions of the application system. */

Step 1. Set NRUN = 0 /* it keeps the number of application run or Iteration involved. */
Step 2. Read: Received Signal FI, FASE, AMPL
Step 3. Set PREVFI = FI, PREVFASE = FASE, PREVAMPL = AMPL
Step 4. Read: Received Signal FI, FASE, AMPL /* successive 2nd reading */
Step 5. If FI <> PREVFI .AND. FASE<> PREVFASE .AND.
   AMPL <> PREVAMPL, Then:
      Set NRUN = NRUN + 1
      If NRUN .EQ. 20, Then:
      /* Compare with the upper limit Value of NRUN i.e., say 20 */
      Call VBYTFLT
      /* Fault detection & recovery Routine is invoked every 20 runs */
      Set NRUN = 0
      GOTO Step 2.
   Else:
      If FI <> FI_LT, Then:
      GOTO Step 2.
Else If FASE .EQ. FASE_LTB .AND. AMPL .EQ. AMPL_LT, Then:
   TRACK /* Initiate track */
Else:
   GOTO Step 2.
[End of If Structure]
[End of If Structure]
{End of the Enhanced Processing Logic i.e., Algorithm 2.}

Algorithm 3

The following steps show how the VBYTFLT Algorithm works.
/* It verifies the corresponding three bytes at an offset say, k, of the three images or copies of the application and its data. Byte error is detected by comparing the k-th byte of the three images of the application and Look-up table. If any byte error has occurred, then the corrupted byte is repaired by overwriting the corrupted byte with the byte pattern in majority. Any disagreement among the corresponding bytes indicates the potential transient bit-errors. Starting addresses of the three images are known. */

Step 1. Initialize: Size of an image in bytes. /* size of an image in bytes is known */

Step 2. Compare the k-th byte of the three images to find a majority one.

Step 3. If there is a disagreement, Then rewrite or repair the odd byte of an image with the majority one found at step-2.
   Else If there is no disagreement, Then increment k by one and go to step-2 for checking the next byte until the end of an image.
   Else If no agreement among the three bytes (a crash), reload the application, Then go to ERROR.

/* if no majority is found then call an ERROR routine for restart or re-executing the application. */
End If

Step 4. Return to the enhanced tracking program (algorithm-2) for reliable tracking.
{End of the VBYTFLT algorithm-3.}

3. Discussion

Algorithm-1 describes the basic processing logic. The input parameters are compared with the stored records in the look up table. If there is a matching record inside the LT, then it does not track because of the fact that LT stores only the parameters of the friends’ aircraft. But a mismatch or a 'not found' in LT, indicates that the aircraft is of a foe and therefore it needs to be tracked and that is why the tracking function is initiated. However, the basic processing logic does not work properly in reality.

Algorithm-2 describes the steps involved in this application with an enhanced processing logic. This logic uses the input parameters of two immediate and successive readings in order to eliminate any ambiguity in input data arising due to potential transients etc. It also behaves like an input filter. The NRUN variable can be tuned in order to combat the transients’ effect. In this algorithm, after every twenty (say, NRUN’s upper limit is 20) runs of the application, the error detection and recovery routine namely, VBYTFLT is invoked. If the transient threat is more frequent, then upper limit of the NRUN variable may be reduced to a value say, 5. Thus depending on the real environment, the time interval between two successive calls of VBYTFLT routine during the execution of the application, can be reduced (to say 1) or increased by changing the upper limit of the NRUN variable, as shown at step-5 of the algorithm-
2. The variables namely, PREVFI, PREVFASE, PREVAMPL are to store the previous inputs, whereas FI, FASE, AMPL variables store the most recent input parameters.

Algorithm-3 shows the steps involved in order to detect and correct errors in the look up table namely, LT, as well as in the application code. There are three images of the application code along with the LT, stored in memory of the computing machine. If the starting addresses of the three images are say $I_1$, $I_2$ and $I_3$ respectively. When the offset $K$ is say 0 (initial value), then the address $I_1^0$ denotes the starting address of first image $I_1$ only, because $I_1^0$ has the value of $(I_1 + 0)$ i.e., starting address plus offset. In general, if $I_m$ be the starting address of the $m^{th}$ image then, the address of the $k^{th}$ byte (or at offset say, $k$) is shown by equation (1).

$$I_m^K = I_m + k \quad (1)$$

Again, if any one byte out of the three corresponding bytes of three images at an offset says, $k$ is corrupted, then this VBYTFLT routine repairs the corrupted byte by overwriting the erroneous byte with the byte in majority. The affected byte is detected by means of comparison of three bytes at the same offset, as shown at step 2, 3 of the Algorithm-3.

The possibility of getting inadvertently alteration (by transients) of two bytes at distant locations, into a similar corrupted value in order to dissatisfy the step-3 of algorithm-3 is almost nil. In other words the chances of byte error remaining undetected is

$$1 / (2^8) \times 1 / (2^8) = 2^{-16} \quad (2)$$

This method is capable of detecting even all 8 bit errors i.e., even an entirely corrupted byte is also detected.

Again, it can repair all the 8-bit errors. If say, $I_m^K$ byte is corrupted to $I_m^* K$, and say, $I_m^K$ and $I_m^o K$ byte contents are same at an offset $k$ of two images $I_m^m$ and $I_m^o$, and then by comparing three corresponding bytes of the three images, we can detect that $I_m^m K$ byte is corrupted (as shown at step-3 of the Algorithm-3). The corrupted byte is repaired by overwriting the wrong one with the majority one. This is applicable even for 8 bit errors in a byte.

If there is no error in program and data code then the following equation will be satisfied.

$$I_m^m K = I_m^o K = I_m^* K \quad (3)$$

The chances of satisfying the equation (3) by the corrupted three bytes of the three images at the same offset is negligibly small, because the transients’ effects on memory, registers are very random and independent in nature.

$$1/(2^8) \times 1/(2^8) \times 1/(2^8) = 2^{-24} \quad (4)$$

In other words, the chances of three bytes at different locations corresponding to a particular value with similar bit pattern, getting altered (due to random effects of transients) simultaneously to a similar value in order to satisfy equation (5) is negligibly small.

$$I_m^m K = I_m^o K = I_m^* K \quad (5)$$

Again, the chances of a particular value (of one byte size) stored at same offset in the three images, getting altered to three different values (of different bit pattern), is

$$1 / (2^{24})$$

The above disastrous effect indicates a possibility of memory hardware or permanent errors and the ERROR routine is invoked for necessary recovery thereof.
In other words, the possibility of invoking the error routine namely, ERROR (as shown at step-3) will be negligibly small.

The routine VBYTFLT verifies for byte errors for the entire application program along with the reference data table (LT), by increasing the value of offset K from 0 to \(N - 1\) (the size of an image say, of N bytes). This is very effective for online error detection and error recovery of the application during the life cycle of the application. After detecting and repairing the entire application, program control goes back to the main application (as shown in Algorithm-2). Even a totally corrupted image can be repaired by this proposed technique by repairing byte after byte.

Space redundancy of this proposed technique is about three. However, because of the lower economic trend on the hardware prices, this much space redundancy can be easily affordable. Little higher time redundancy can be affordable because of easily affordable high speed machine. This proposed technique is capable of detecting and repairing any number of soft errors (not reproducible) as well as permanent errors during the run time of the application. Fault detectibility is inversely proportional to the upper limit of the variable NRUN, i.e.

\[ F_D \propto \frac{1}{NRUN_{UL}} \]  

(6)

Thus depending on the threat of transients potential, the upper limit value of NRUN can be changed in order to meet the requirement.

There are several conventional error detection and correction schemes like Parity Checks, Hamming Codes, Cyclic Redundancy Checks (CRC), Checksums etc. They are not free from limitations as stated in Rhee (1985), Saha (1999). The single parity checks can detect only odd number of errors. Any even number of errors remain undetected. So it is inadequate for detecting any number of errors. Again for CRC, shift register based circuits are used. CRC is normally shift register based. The Shift register circuit is for dividing polynomials and finding the remainder. Modulo 2 adders, multiplier circuits are also used. In CRC, when errors actually occur, the receiver fails to detect the errors when the remainder is zero. Again, CRC is having high time redundancy and that is why they are normally hardware based. Software based CRC implementation is impractical in a real time application. 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 transients or in the case of even changes, the errors remain undetected by this typical Checksum. Interested reader may refer the works of (Huang and Abraham 1984), (Avizienis 1985), (Liestman 1986), (Saha 1995), (Saha 2000), (Saha 2003), (Papageorgiou and Kakolakis 2004). However, the conventional methods have limitations and there exists higher redundancy in both time and memory space. However, the proposed technique is promising enough to detect multiple errors and corrections thereof with an affordable redundancy in both time and memory space for designing a robust computer-controlled system.

4. Conclusion

The proposed software based technique is a very cost effective and an economically important tool in comparison to the traditional triple modular redundancy (TMR) or an N-Versions programming technique that are based on design diversification. It can detect and repair errors. However, the proposed approach does not address the
design bugs of an application. During the life cycle of an application where an ultra reliability in the computed result is essential, this proposed technique will be very effective at the cost of affordable redundancy in time and space, without any increase in the monetary budget. This proposed technique can be used as a very effective tool for the system engineers. It is easier to implement also. It provides high fault detection and reliability in the processing logic. It is also a very useful and a low cost tool for gaining dependable computation and higher maintainability of a computer controlled system with an affordable memory overhead of 3.2 and 3.25 time redundancy.

References