

Transient Software Fault Tolerance Through Recovery

By Goutam Kumar Saha

A low-cost and effective solution for developing transient fault tolerant application software based on a single-version program

This article describes an unconventional software design technique of transient software fault tolerance through designed in recovery technique, without using conventional N-version software approach. Instead of using design diversity, it uses only single-version software approach with an affordable redundancy in both time and space.

Introduction

Fault tolerance means that a system can provide its services even in the presence of faults. A system *fails* when it cannot meet its promises. An *error* is a part of a system's state leading to a failure. The cause of an error is called a *fault*. Electrical short-duration noises or transients often cause multiple, random and independent bit errors in a processor's registers and memory.

There are many techniques available for checking validity of data or program code. Some of these use a parity bit, cyclic redundancy checks, checksums or error-correcting codes. Such techniques can detect multiple errors but cannot correct all of them. These hardware based error-correcting codes are capable of both detecting and correcting certain types of limited number of errors. Traditional hardware fixes cannot detect and correct all such random and multiple bit errors caused by potential transients. The conventional N-Version Programming (NVP) approach to fault-tolerant software systems involves the generation of functionally equivalent, yet independently developed and maintained software components, called N-Version Software (NVS) [1].

All these independent components are executed concurrently under a system, called N-version Executive (NVX). It uses a decision algorithm based on consensus to determine final output values. However, the *NVS* approach fails to provide service against such random bit errors because of the lack of consensus voting output. The objective of this article is to describe an unconventional software technique to detect and recover all such random bit errors caused by transients. It does not need N-Versions of the applications based on design diversity; rather it uses an enhanced version of the application.

Description of Works

The proposed technique needs three images of the application. Say, A_1, A_2, A_3 are the three images of the application program. The data codes reside on at three different locations beginning from, say, L_1, L_2, L_3 locations onwards. Each image is of, say, size of total B bytes only. Initially three pointers say, P_1, P_2, P_3 point to the starting addresses of the images i.e., L_1, L_2, L_3 respectively. The symbol $_$ denotes logical *AND*ing. The steps involved in designing this software fix namely, *Fault_Detection_Recovery* for generating a robust and an enhanced version of the application, are stated below.

```

Step 1.  $N = 0$  /* Initialize the variable N to count the byte number */
Step 2.  $P_1 = L_1$ 
         $P_2 = L_2$ 
         $P_3 = L_3$  /* Initialize the pointer variables to the starting addresses of three
                    images of the application */
Step 3.  $B = \text{Size\_of\_Application\_in\_bytes}$  /* Total number of bytes in each image
                                                is known */
Step 4. If ((  $[P_1] == [P_2]$  )  $_$  (  $[P_1] == [P_3]$  ) ), Then:
        Goto Step 5 /* Initially  $[P_1]$  denotes the content of byte at location  $L_1$  */
                /* If corresponding three bytes are not corrupted, then
                verify for the next byte and so on. */
Else If ((  $[P_1] == [P_2]$  )  $_$  (  $[P_1] \neq [P_3]$  )  $_$  (  $[P_2] \neq [P_3]$  ) ), Then:
         $[P_3] == [P_1]$  /* Corresponding corrupted byte of  $A_3$  is corrected
                        by  $P_1$ 's content */

Else If ((  $[P_1] == [P_3]$  )  $_$  (  $[P_1] \neq [P_2]$  )  $_$  (  $[P_2] \neq [P_3]$  ) ), Then:
         $[P_2] == [P_1]$  /* Corresponding corrupted byte of  $A_2$  is corrected */

Else If ((  $[P_2] == [P_3]$  )  $_$  (  $[P_1] \neq [P_2]$  )  $_$  (  $[P_1] \neq [P_3]$  ) ), Then:
         $[P_1] == [P_2]$  /* Corresponding corrupted byte of  $A_1$  is corrected */

```

```

Else If ( ([P1] ≠ [P2] ) _ ([P2] ≠ [P3] ) _ ([P1] ≠ [P3] ), Then:
    Goto Error_Recovery /* Corresponding bytes of all the three images
                        are corrupted to different values and then
                        recover them by copying from the master file
                        or backup, due to such total damage */

End If

```

```

Step 5. P1 ++ /* Pointers and byte counter N are incremented by one to point
            the next byte */

```

```

P2 ++
P3 ++
N ++

```

```

Step 6. If N <= B, Then: /* Verify for the end of application code*/
        Goto Step 4. /* Verify for sanity of next byte code */
Else:
    Return /* Return to main application A1 for continuing */
End If

```

{End of the Algorithm of *Fault_Detection_Recovery*}

Discussion

The proposed technique can detect and recover errors in one byte or eight bit random errors. In other words, if any one byte out of the three corresponding bytes, say, at locations L_1, L_2, L_3 at the three images of the application A_1, A_2, A_3 is corrupted, then the corrupted byte is recovered (as shown at Step 4). However, if all three corresponding bytes are corrupted to a similar bit-pattern to represent another value, then only this technique's effectiveness is reduced. The probability of one particular value stored at three different locations, getting inadvertently corrupted to the same bit-pattern representing an another value is 2^{-24} only.

The probability of such ambiguity causing failure of the proposed technique due to such independent faults at three images is

$$1 / 2^8 * 1 / 2^8 * 1 / 2^8 = 1 / 2^{24}$$

Interested readers may refer to [2,3,4,5,6] for related works on transient software fault tolerance. The procedure discussed above, namely, *Fault_Detection_Recovery*, can be easily invoked periodically from an

application system. However, the frequency of invoking this sanity procedure depends on how noisy (electrically) the industrial computing environment is. System engineers can easily modify this frequency in order to meet the transients' threats.

Conclusion

The proposed technique is a low cost and effective solution towards development of a transient fault tolerant application software based on a single-version program, with few additional codes for fault detection and recovery. However, system designers can easily afford such redundancy in both space and time, in this modern computing age of affordable, high speed computing machines, in order to gain higher reliability and high transient software fault tolerance. This technique is easier to implement and more economical in comparison to conventional NVP.

References

- [1] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance during Program Execution", in Proceedings *COMPSAC 77*, pp. 149-155.
- [2] Goutam Kumar Saha, "A Software Fix towards Fault Tolerant Computing", to be published in *ACM Ubiquity*, vol. 4, 2003.
- [3] Goutam Kumar Saha, "Mobile Computing & Fault Tolerance Issues", in Computer Society of India – *CSI Hard Copy*, Vol. 37, August, 2003.
- [4] Goutam Kumar Saha, "Transient Software Fault Tolerance in Mobile Computing", in *CSI 2003 - 38th National Con.*, New Delhi, Computer Society of India, 2003.
- [5] Goutam Kumar Saha, "Transient Fault Tolerant Processing in a RF Application", *International Journal - SAMS*, vol. 38, pp. 81-93, Gordon and Breach, 2000.
- [6] Goutam Kumar Saha, "Transient Fault Tolerance in Scientific Computing", Paper No. 330, *ACM Transactions on Computer Systems*, 2003.

Goutam Kumar Saha is with Centre for Development of Advanced Computing, Kolkata, India. Previously, he worked in LRDE, Defence R & D Organisation, Bangalore, India, as a scientist. His field of interest is transient software fault tolerance.