

## Short Paper

---

### A Software Tool for Fault Tolerance

GOUTAM KUMAR SAHA

*Centre for Development of Advanced Computing  
Kolkata, West Bengal 700091, India*

*E-mail: gksaha@rediffmail.com; gksaha@ieee.org*

This paper describes a software fix in order to tolerate multiple transient-faults in an application using code-redundancy of an application program that is enhanced with a new error-checking and switching technique. It is a low cost solution towards tolerating multiple bit-errors. This technique which is based on enhanced single-version scheme (ESVS), does not intend to correct errors. Rather it aims to mask various operational and environmental errors during the run-time of an application.

**Keywords:** transient bit-errors, fault tolerance, error masking, enhanced single-version scheme (ESVS), low-cost tool

#### 1. INTRODUCTION

We should accept that relying on software techniques for obtaining fault tolerance and dependability means accepting some overhead in terms of increased size of code and reduced performance. Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. Transient faults occur once and then disappear. If the operation is repeated, the fault may go away. An intermittent fault occurs, then vanishes of its own accord, then reappears and so on. A permanent fault is one that continues to exist until the faulty component is repaired. Electrical transients often cause random bit-errors of application code, data and faulty program control resulting in erroneous output. Software Fault Tolerance is the reliance on "Design Redundancy" to mask residual design faults present in software program. The proposed technique adopts the strategy of defensive programming based on redundancy. Software is an indispensable element in many aspects of our daily lives. In addition to the costs of developing software, penalty costs resulting from software failure are even more significant.

The vast majority of hardware failures in modern microprocessors (MP), especially for transient faults, are because of the limited hardware detection in them [1]. Transient faults are random events. They occur when various noise sources cause an incorrect result. Normally, alpha particles and other cosmic radiation alter the state of latches and dynamic logic, resulting in logic errors. The frequency of the transient faults is low now. However, devices getting smaller in size with increased transistor density, higher clock

---

Received May 18, 2004; revised July 26, 2004; accepted November 17, 2004.  
Communicated by Chin-Teng Lin.

frequency, and low power supply accompanied with deep sub-micron technology, do not only experience reduced noise margin and reliability but also experience increased impact of defects [7, 8].

In this proposed technique of enhanced single-version scheme (ESVS), each instruction of the basic processing logic is followed by a one-byte “NO-Operation (NOP)” instruction code. The memory locations or offsets of the inserted NOP-codes are known. Execution of NOP code does not alter the processor status word (PSW) and it provides a short-delay only. The immunity for the additional NOP code is checked before program control goes forward to execute adjacent application-instructions. As soon as one NOP code-corruption is detected, the checksum of the application code is computed and if this checksum does not match with the previously computed and stored checksum, then program control exits that corrupted image of the application code and data, and then starts execution from the beginning of the next image of the application already stored on memory. The required number of images of the application is  $(f + 1)$  in order to tolerate  $f$  number of sequential faults. Thus, transient bit-errors are tolerated. Again, a faulty program flow caused by erroneous codes, is also prevented and the program control is brought to the beginning of the next subsequent image of the application. Thus, this technique saves the cost of developing multiple versions of the application and the cost of multiple machines, as required in  $N$  modular redundancy (NMR e.g. Triple Modular Redundancy) technique. This ESVS technique needs only multiple copies or images of the same version running on one machine sequentially. As only one image is executed at any point of time, this ESVS technique has no overhead of synchronization also. The ESVS uses only one version and one machine. That is why, this approach is a low cost solution towards fault tolerant computing. In this ESVS technique, error processing is performed through error detection and possible switching of results. In this paper, fault tolerance against multiple and random bit-errors is carried out through on-line error-checking and switching code, for results. The proposed technique is suitable for various medium-scale computer controlled application systems or various embedded systems also because of no unaffordable overhead on space and time redundancy.

## 2. BACKGROUND AND PREVIOUS WORKS

In order to achieve ultra reliability in computing, it is necessary to adopt the strategy of defensive programming based on redundancy i.e. fault tolerant software e.g., Recovery Blocks (RB) [1],  $N$ -Version Programming (NVP) [2]. In RB, the acceptance test condition is expected to be met by the successful execution of either the primary module or the alternate modules. When an acceptance test detects a primary module failure, an alternate module executes. If all alternate modules are exhausted, the system crashes. In NVP,  $N$  number of variants or alternates run simultaneously on  $N$  different machines and at the end of program, the results are voted for a majority one that is considered as a correct result. If no consensus in results is found then the NVP system crashes. However, both the RB and NVP need multiple versions of software to be developed independently using different languages, tools etc. In reality, designing one version of reliable software is itself a very costly and challenging task. Again, designing multiple versions of software is found to be very expensive and beyond reach for many medium scale industries.

However, in critical systems with real-time deadlines, voting at program's end may not be acceptable. This scheme requires synchronization of the various software versions at comparison points. Errors that have no relation to each other are called random-bit errors. A burst error is a large error, disrupting numerous bits. In interleaving, data is interleaved or dispersed through the data stream prior to storage or transmission. With interleaving, the largest error that can occur in any block is limited to the size of the interleaved section. The field of error-correction codes is highly mathematical one. In general, two approaches are used: block codes using algebraic methods, and convolution codes using probabilistic methods. With interleaving, off-line correction of burst errors becomes easier with high time redundancy. A Self-Stabilizing system [3] cannot recover from random errors within a finite number of steps. It is suitable for recovering from systematic errors. Interested readers may refer to other works on fault tolerance-through recovery [4] and through a checksum over a row [5]. However, a typical fault tolerance technique may not need to perform error correction because of the need for quick results.

### 3. UNDERSTANDING THE WORK

This proposed Enhanced Single Version Scheme (ESVS) technique needs  $f$  number of copies or images of the enhanced application program and data, for tolerating  $(f - 1)$  number of sequential faults. The application is enhanced by inserting "NO-Operation" (NOP) instruction codes after every basic-application-instruction code (say,  $I_i$ ). During the run-time of an application, the error-processing code verifies for the possible NOP-code-corruption, as the locations of inserted NOP codes are known. If a NOP-code error is detected, a typical checksum is computed by XORing the application bytes and this checksum value is compared with the stored checksum value. If there is a mismatch, the program control switches to the next image of the application. The events of errors in NOP-codes and checksum, ascertain the possible errors in application-codes. The ESVS technique is explained in the following steps. The notation  $L_i^j$  denotes the byte-location of  $i$ -th NOP-code (i.e. inserted one) inside the  $j$ -th image of the application. The symbol  $[ ]$  denotes the byte-content. In the following steps of the ESVS scheme, we have used three copies of an application. Each copy has two error-processing codes. Each error-processing code covers a set of application-instructions along with extra NOP codes.

```

Image1
  /* 1st Error-processing code in the 1st image i.e., EPC11. The symbol ∨ denotes logical
  ORing */
  If ( $[L_1^1] == [L_2^1] == [L_3^1] \neq \text{NOP-code}$ ) ∨ (run-time-checksum  $\neq$  stored-checksum) then:
    Jump to the beginning of Image2 /* Image1 is faulty, so execute the next image */
  Endif

  I1 /* application-instruction-code */
L11 NOP
  I2

```

```

L21 NOP /* inserted NOP-code */
      I3
L31 NOP

/* 2nd Error-processing code in the 1st image i.e., EPC21 */
If ([L41] == [L51] == [L61] ≠ NOP-code) then:
    Jump to the beginning of Image2 /* Image1 is faulty, so execute the next image */
*/
Endif

L41 I4
L51 NOP
      I5
L61 NOP
      I6
      NOP
      .
      .
      .
END /* End of the Image1 */

```

### Image<sub>2</sub>

```

/* 1st Error-processing code in the 2nd image i.e., EPC12. The symbol ∨ denotes logical
ORing */
If ([L12] == [L22] == [L32] ≠ NOP-code) ∨ (run-time-checksum ≠ stored-checksum) then:
    Jump to the beginning of Image3 /*Image2 is faulty, so execute the next image */
Endif

I1 /* application-instruction-code */
L12 NOP
      I2
L22 NOP /* inserted NOP-code */
      I3
L32 NOP

/* 2nd Error-processing code in the 2nd image i.e., EPC22 */
If ([L42] == [L52] == [L62] ≠ NOP-code) then:
    Jump to the beginning of Image3 /* Image2 is faulty, so execute the next image */
Endif

I4
L42 NOP
      I5
L52 NOP
      I6
L62 NOP
      .
      .
      .
END /* End of the Image2 */

```

```

Image3
/* 1st Error-processing code in the 3rd image i.e., EPC13. The symbol ∨ denotes logical
ORing */
If ([L13] == [L23] == [L33] ≠ NOP-code) ∨ (run-time-checksum ≠ stored-checksum) then:
  If (Image4 does exist) Then:
    Jump to the beginning of Image4
  Else:
    System Crashes /* Reload and restart the application */
    /* Image3 is faulty, so execute the next image if exists, other wise system crashes
because there is no spare correct image to execute. */
  Endif

I1      /* application-instruction-code */
L13  NOP
I2
L23  NOP /* inserted NOP-code */
I3
L33  NOP

/* 2nd Error-processing code in the 3rd image i.e., EPC23 */
If ([L43] == [L53] == [L63] ≠ NOP-code) then:
  If (Image4 does exist) Then:
    Jump to the beginning of Image4
  Else:
    System Crashes /* Reload and restart the application */
    /* Image3 is faulty, so execute the next image if exists, other wise system crashes
because there is no spare image to execute. */
    /* Image3 is faulty, so execute the next image if exists, other wise system crashes
because there is no spare image to run. */
  Endif

I4
L43  NOP
I5
L53  NOP
I6
L63  NOP
.
.
END      /* End of the Image3 */

```

**Example:**

- Compute the Factorial of  $N$  using Intel 8086 MASM.
- We need to use error-processing code at the beginning of an image as well as at the beginning of a large application.
- We should put error-processing code at location prior to the branch instruction or to the procedure code also.
- One error-processing code is shown in this example.

- The number of instructions guarded by an error-processing code may vary.
- The higher this number is the longer the delay is for verification towards correct result.

**Image<sub>1</sub>**

```

/* 1st Error-processing code in the 1st image i.e., EPC11. The symbol ∨ denotes logical
   ORing. The symbol ∇ denotes logical XORing. */
If ([L11] ∇ [L21] ∇ [L31] ∇ [L41] ∇ [L51] ∇ [L61] ∇ [L71] ∇ [L81] ∇ [L91] ∇ [L101] ∇ [L111]
   ∇ [L121] ≠ 0) ∨ (run-time-checksum ∇ stored-checksum ≠ 0) then:
    Jump to the beginning of Image2 /*Image1 is faulty, so execute the next image */
Endif
MOV AH, 01 h      ; Read the number N from keyboard
L11  NOP
INT 21h          ; Request INT 21h service 1
L21  NOP
SUB AL, "0"      ; Input number in AL
L31  NOP
MOV DL, AL
L41  NOP
DEC DL
L51  NOP
MOV CX, DL      ; Set the counter register for iteration
L61  NOP
LVL1: MUL DL     ; AX ← AL * DL
L71  NOP
DEC DL
L81  NOP
LOOP LVL1       ; Repeat through the level LVL1, until the counter register CX is
                 decreased to zero
L91  NOP
MOV DL, AL
L101 NOP
MOV AH, 02h
L111 NOP
INT 21 h       ; Display factorial
L121 NOP
INT 20 h       ; Stop
END

```

**Image<sub>2</sub>**

```

/* 1st Error-processing code in the 2nd image i.e., EPC12. The symbol ∨ denotes logical
   ORing. The symbol ∇ denotes logical XORing. */
If ([L12] ∇ [L22] ∇ [L32] ∇ [L42] ∇ [L52] ∇ [L62] ∇ [L72] ∇ [L82] ∇ [L92] ∇ [L102] ∇ [L112]
   ∇ [L122] ≠ 0) ∨ (run-time-checksum ∇ stored-checksum ≠ 0) then:
    Jump to the beginning of Image3 /* Image2 is faulty, so execute the next image
   */
Endif
L12 MOV AH, 01 h      ; Read the number N from keyboard

```

```

L22   NOP
        INT 21h           ; Request INT 21h service 1
L32   NOP
        SUB AL, "0"       ; Input number in AL
L42   NOP
        MOV DL, AL
L52   NOP
        DEC DL
L62   NOP
        MOV CX, DL       ; Set the counter register for iteration
LVL1:  NOP
L72   NOP
        MUL DL           ; AX ← AL * DL
L82   NOP
        DEC DL
L92   NOP
        LOOP LVL1
L102  NOP
        MOV DL, AL
L112  NOP
        MOV AH, 02h
L122  NOP
        INT 21 h        ; Display factorial
        NOP
        INT 20 h        ; Stop
        END

```

**Image<sub>3</sub>**

/\* 1st Error-processing code in the 3rd image i.e., EPC<sub>1</sub><sup>3</sup>. The symbol ∨ denotes logical ORing. The symbol ∇ denotes logical XOR

ing. \*/

If ([L<sub>1</sub><sup>3</sup>] ∨ [L<sub>2</sub><sup>3</sup>] ∨ [L<sub>3</sub><sup>3</sup>] ∨ [L<sub>4</sub><sup>3</sup>] ∨ [L<sub>5</sub><sup>3</sup>] ∨ [L<sub>6</sub><sup>3</sup>] ∨ [L<sub>7</sub><sup>3</sup>] ∨ [L<sub>8</sub><sup>3</sup>] ∨ [L<sub>9</sub><sup>3</sup>] ∨ [L<sub>10</sub><sup>3</sup>] ∨ [L<sub>11</sub><sup>3</sup>] ∨ [L<sub>12</sub><sup>3</sup>] ≠ 0) ∨ (run-time-checksum ∇ stored-checksum ≠ 0), then:

If Image<sub>4</sub> exists, then:

Jump to the beginning of Image<sub>4</sub> /\*Image<sub>3</sub> is faulty, so execute the next image \*/

Else:

System crashes. /\* Reload and Restart the application. \*/

End if

End if

MOV AH, 01 h ; Read the number N from keyboard

```

L13  NOP
        INT 21h           ; Request INT 21h service 1
L23  NOP
        SUB AL, "0"0     ; Input number in AL
L33  NOP
        MOV DL, AL
L43  NOP
        DEC DL
L53  NOP

```

	MOV CX, DL	; Set the counter register for iteration
L <sub>6</sub> <sup>3</sup>	NOP	
LVL1:	MUL DL	; AX ← AL * DL
L <sub>7</sub> <sup>3</sup>	NOP	
	DEC DL	
L <sub>8</sub> <sup>3</sup>	NOP	
	LOOP LVL1	
L <sub>9</sub> <sup>3</sup>	NOP	
	MOV DL, AL	
L <sub>10</sub> <sup>3</sup>	NOP	
	MOV AH, 02h	
L <sub>11</sub> <sup>3</sup>	NOP	
	INT 21 h	; Display factorial
L <sub>12</sub> <sup>3</sup>	NOP	
	INT 20 h	; Stop
	END	

#### 4. BAYESIAN ANALYSIS AND CONTRIBUTION

Based on the basic steps as stated in the previous section and the Bayesian analysis, we get the following possible program flows for deriving the probability of uncertainty of this ESVS scheme. We have taken the ESVS scheme with three images of an application. Each image (as shown in the example) has one block of code. For each image (or a block of code), we have used one error-processing code here. Here, each block of code consists of twelve application-instructions and twelve extra NOP instructions. A larger program may have multiple block of codes and in such case we need to avoid inter block branching. The notation  $EPC_1^{1*}$  indicates the erroneous NOP code in the block  $B_1^1$ .  $I^1$  denotes the Image<sub>1</sub> code.

**Table 1. Performance analysis of ESVS.**

Block size (or one image size)	No Fault	Tolerating 1 Fault	Tolerating 2 Faults	Space Redundancy (Executable Code Size)	Probability of Uncertainty (on Fault Coverage)
12 application- instructions + 12 NOP instructions	O(2)	O(2.6)	O(3.3)	O(3.8)	0.2
30 application- instructions + 30 NOP instructions	O(1.93)	O(2.52)	O(3.14)	O(3.3)	0.21
40 application- instructions + 40 NOP instructions	O(1.91)	O(2.4)	O(2.9)	O(3.1)	0.23

The all possible program flows (the ESVS with three images only):

- (1)  $EPC_1^1 \rightarrow I^1$ , (No fault at Image<sub>1</sub> code is detected, so execute the Image<sub>1</sub> application code).

- (2)  $EPC_1^{1*}$  (fault is detected in  $I^1$ , so skip the  $I^1$ )  $\rightarrow EPC_1^2 \rightarrow I^2$ , ( $I^2$  is detected OK, so execute the  $I^2$  and thus one fault is tolerated).
- (3)  $EPC_1^{1*}$  (fault in  $I^1$ )  $\rightarrow EPC_1^{2*}$  (fault in  $I^2$ )  $\rightarrow EPC_1^3 \rightarrow I^2$ , (Two faults are tolerated).
- (4)  $EPC_1^{1*}$  (fault in  $I^1$ )  $\rightarrow EPC_1^{2*}$  (fault in  $I^2$ )  $\rightarrow EPC_1^{3*}$  (fault in  $I^3$ )  $\rightarrow$  CRASH, (No spare is there to be executed, so application needs to be restarted).

## 5. EXPERIMENTAL RESULTS

To evaluate the feasibility and effectiveness of this proposed ESVS approach, we have applied this approach on a set of simple C programs (used as benchmarks) by manually modifying their source code according to its rules. Motorola 68040 processor and the compiler-Single step 7.4 by SDS, Inc., have been used with disabling all compiler optimizations when compiling the ESVS based application with four images. It is observed that on average, the size of the executable code is increased by 2.58 times and the source code grows by 4.66 times. An average slow-down of about 2.5 times is observed for tolerating two sequential faults. The fault injection environment as described in [6], is built around an application board hosting a 25 MHz Motorola 68040 processor, 2 Mbytes of RAM memory, and some peripheral devices. Fault injection is performed exploiting an ad hoc hardware device which allows monitoring the program execution and triggering a fault injection procedure when a given point is reached. For experiments, the adopted fault model is the single-bit flip into memory locations. Faults are randomly generated. On injecting total 2000 randomly generated faults (500 in the memory area containing data, and 1500 in the memory area containing the code) in an application (based on ESVS with two images) of Matrix multiplication of two matrices composed of  $8 \times 8$  integer values, out of total 2000 errors the number of hardware detected (by Error Detection Mechanisms e.g., microprocessor exceptions) is 382 and the Software detected (by this ESVS approach in case of disagreement among the NOP code) is 774 and the fail silent (not producing any difference in the program behavior) is 841. The rest 3 are the fail silent violations (not detected by any error detection methods).

Again, on injecting total 2000 randomly generated faults (500 in the memory area containing data, and 1500 in the memory area containing the code) in an application (ESVS based with two images) of solving a series of twenty quadratic equations, out of total 2000 errors, the number of hardware detected (by Error Detection Mechanisms e.g., microprocessor exceptions) is 423 and the Software detected (by this ESVS approach in cases of disagreement among the extra NOP instructions) is 782 and the fail silent (not producing any difference in the program behavior) is 791. The rest 4 is for fail silent violations (not detected by any error detection methods).

On an average, out of total 2000 errors, the average number of hardware detected (by Error Detection Mechanisms e.g., microprocessor exceptions) is 403 and the average number of Software detected (by this ESVS approach in such case of disagreement among the extra NOP instructions) is 778 and the average number of fail silent (not producing any difference in the program behavior) is 816. The average number of fail silent violations (not detected by any error detection methods) is 3.

This work aims not to tolerate software design bugs, rather it aims to design a robust and a low cost solution for tolerating transient faults of multiple bit flips during the exe-

cution of an application through adopting the code redundancy along with self-checking code. It can tolerate two consecutive faults on using only two copies of an application. Wrong inter-block program flow inside an application is detected when a program control enters a block without executing its error-processing code, and such events can be caught by employing and examining the MR\_EC variable which holds the error-code number that got executed most recently. However, a wrong program flow within a block of code in an application cannot be detected in this ESVS approach. Again, the detection of a wrong program flow out of the application image but in between the two copies of an application is limited by the number (on average, it is half the number of bytes in a copy of an application) of extra NOP code, in between two copies only. Again, an error that may occur in a block after the execution of its error-processing code remains undetected until the re-usage of a block. In other words, the fault detection capability of this ESVS scheme is limited over the errors in a block that might have occurred before the execution of its error-code only and such errors get detected only at the subsequent reference of the erroneous block. We rely that such error which occurs just after the execution of an error processing code, will get detected by the microprocessor's exception handling mechanism. Thus, this ESVS along with processor's exceptions is an effective solution to attain non-fail-stop kind of fault tolerance through hardening a processor-based application at the cost of an affordable overhead on both the time and space.

## 6. COST, EFFECTIVENESS AND RELIABILITY

Depending on the degree of expected computation-reliability, the error-processing code can be inserted more frequently. At this example, after every twelve (say,  $N$ ) application-instruction-codes, the error-processing code is inserted. At the event of detected fault, the time for switching to next image for results, consists the execution time of the in between application-instruction-codes ( $N$ ) and  $N$  number of NOP-codes (i.e.  $N$  machine clocks), and the execution time of the error-processing-code. The switching time is proportional to the value of  $N$  for a typical sized application program. In an application where switching time is not so critical, the value of  $N$  can be higher. In other words, number of insertions of the error-processing-code is lower. For an application with few number codes, only one error-processing-code can be inserted at the beginning of an image. This technique covers the faults over the entire application program. The selection of NOP-code for insertion, is guided by the knowledge that it is one byte long and it provides a delay of machine clock only without changing the processor-status-word (PSW). It is assumed that if a NOP-code is corrupted then its adjacent application-instruction-codes are also corrupted because of the fault model of multiple byte-errors in random. In order to overcome the limitations of this checksum over multiple bit-errors, extra NOP-codes are inserted inside the application-code and the task for detecting possible corrupted NOP-codes is performed. Again, for an another fault model where NOP-codes are not corrupted but one application-instruction is corrupted, then the checksum checking method is able to detect such fault and then the program control switches to an another image of the enhanced application program. Again, if one of the error-processing-code is corrupted then such error is detected by a subsequent error-processing-code. When all the  $f$  images are corrupted then this SV technique crashes. In other words, this Single-Ver-

sion technique can tolerate total  $(f - 1)$  number of faults by keeping total  $f$  number of images of the enhanced application program. The proposed approach has been adopted in programs like bubble sort and average computation. An average slow-down of about 3 times is observed. Average number of fail silent faults is about 850, hardware detected faults (detected by a hardware Error Detection Mechanisms EDMs) is about 205 and the Software detected (i.e., detected by the proposed software approach) is about 743. Fault injection is performed exploiting an ad hoc hardware device.

*Effectiveness* of both the recovery block and the  $N$ -version programming schemes relies [10] on the fact that they are basically intended to detect software design bugs. Environmental faults cannot be tolerated by RB and NVP schemes. They employ design diversification on both the software and hardware. *Reliability* of both the recovery block scheme and the  $N$ -Version Programming (NVP) scheme with three design variants is 0.66. It is assumed that only one fault occurs. Whereas the proposed ESVS scheme gains the reliability of 0.79 (or uncertainty of 0.21 only). ESVS is not intended to tolerate software design bugs. *Effectiveness* of the proposed ESVS scheme in tolerating run-time operational faults is high. Because, it tolerates environmental transient faults during the run time of an application on using only a single version code of an application on a single reliable machine. ESVS assumes that software has been designed correctly on employing little extra design time and software has no design bug. ESVS is suitable for both small and medium size applications.

The average *cost ratio* (i.e., the cost of fault tolerant software/cost of non-fault tolerant software) [9] for three variant Recovery blocks is 2.37. Again, the three variant NVP scheme bears the average cost ratio of 2.25. Whereas the proposed ESVP scheme that does not use any design diversification bears the average cost ratio of 1.31 only. Extra enhanced code on error processing contribute only 0.31 at the average cost.

## 7. CONCLUSION

The proposed ESVS technique is a low cost and an effective solution towards designing a fault tolerant application system because it saves the cost of designing multiple independent versions of the application software and the cost of multiple machines, as required by  $N$ -modular system or NVP system. This ESVS technique has no overhead with the synchronization tasks as needed in NVP system. This ESVS technique needs only  $f$  number of images of the enhanced application software running sequentially on one machine only, in order to tolerate  $(f - 1)$  number of faults with a fail-stop failure model (i.e., detection of fault and stopping execution) and with multiple random byte-errors. Error correction is not incorporated in this technique because of very high time redundancy with multiple byte-error recovery codes. However, this ESVS technique can be tailored to fit the exact requirement of a typical application system with proper study and an additional time for designing such single-version fault-tolerant application system. The overhead of extra execution time and space redundancy can easily be afforded using an affordable and modern high speed processing system, in order to gain such reliable computation. This new ESVS technique is also useful for preventing erroneous results caused by corrupted codes. This technique is also a useful tool for designing various computer controlled application systems.

## REFERENCES

1. A. Avizienis, "The N-version approach to fault-tolerant systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, 1985, pp. 1491-1501.
2. K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Transactions on Computers*, Vol. 38, 1989, pp. 626-636.
3. S. Dolev, E. Schiller, and J. Welch, "Random walk for self-stabilizing group communication in ad-hoc networks," in *Proceedings of 21st IEEE Symposium on Reliable Distributed Systems*, 2002, pp. 70-79.
4. G. K. Saha, "Transient software fault tolerance through recovery," *ACM Ubiquity*, ACM Press, Vol. 4, 2003, pp. 1-5.
5. G. K. Saha, "Transient fault tolerant processing in a RF application," *International Journal of System Analysis Modelling Simulation*, Vol. 38, 2000, pp. 81-93.
6. A. Benso, P. L. Civera, M. Rebaudengo, and M. S. Reorda, "An integrated HW and SW fault injection environment for real-time systems," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 1998, pp. 117-122.
7. L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective," *IBM Journal of Research and Development*, Vol. 43, 1999, pp. 863-874.
8. T. Sato, "Analyzing overhead of reissued instructions on data speculative processors," in *Proceedings of Workshop on Performance Analysis and its Impact on Design*, held in conjunction with *25th International Symposium on Computer Architecture*, 1998, pp. 143-147.
9. J. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware and software-fault-tolerant architectures," *IEEE Computer*, Vol. 23, 1990, pp. 39-51.
10. Y. Zorian, D. Gizopoulos, C. Vandenberg, and P. Magarshack, "Guest editors' introduction: design for yield and reliability," *IEEE Transactions on Design and Test of Computers*, Vol. 21, 2004, pp. 177-182.

**Goutam Kumar Saha** in his last seventeen years' research and teaching experience, he has worked as a scientist in LRDE, Defence Research and Development Organization, Bangalore, and at Electronics Research and Development Centre of India, Calcutta. At present, he is with The Centre for Development of Advanced Computing (CDAC), Kolkata, India, as a Scientist-F. He has authored around eighty-five research papers including SAMS Journal, ACM, C&EE Journal, IEEE, CSI etc. He is a senior member in IEEE, Computer Society of India, ACM, Fellow in IETE etc. He has received various awards, scholarships and grants from national and international organizations. He is a reviewer for AMSE Journal (France), IJCSA and the Journal of the Computer Society of India (JCSI). His field of interest is on fault tolerant computing, natural language processing and dependable computing.