JOURNAL OF SOFTWARE ENGINEERING & INTELLIGENT SYSTEMS
ISSN 2518-8739
31st August 2017, Volume 2, Issue 2, JSEIS, CADMEI Copyright © 2016-2017
www.jseis.org

Towards tolerating faults by triple modular redundancy

[1]Muhammad Sheikh Sadi, [2]Shaheena Sultana, [3]Md. Moin Al Rashid, [4]Kazi Fahat Lateef
[1,2,3,4]Khulna University of Engineering and Technology (KUET), Khulna, Bangladesh
E-mail: [1]sheikhsadi@gmail.com, [2]shaheena223@cse.kuet.ac.bd, [3]mmoyeenshuvo@gmail.com,
[4]fahadlateef@gmail.com

ABSTRACT

Increased clock frequencies, higher transistor counts, lower voltage levels, and reduced noise margin have exponentially raised performance of modern microprocessors but made processors more vulnerable to soft errors. To detect soft errors, redundant software and hardware are introduced frequently by the designers but software techniques have shown their capability to protect against soft errors without any hardware overhead and are more beneficial for their flexibility and low cost as well as easier to deployment. The techniques of Software fault-tolerance enable application protection by constructing redundancy into the compiled code. A new methodology is proposed for tolerating soft errors through triple modular redundancy. Experimental studies show that this method has increased reliability and offer efficient protection via software modulation in comparison to existing duplication and triplication methods.

Keywords: soft errors; fault tolerance; triple modular redundancy; reliability; risk mitigation;

1. INTRODUCTION

"Glitches" were an acknowledged way of life in the beginning of Personal Computers (PC). From that point forward, as PCs have turned out to be more dependable, application fault versatility gets to be vital. The scaling of devices, operating voltages, and design margins for increasing performance and functionality raises concerns about the susceptibility of future-generation systems to soft errors. Soft errors are contributed by soft numerous sources, together with electromagnetic interference, capacitive coupling, and other sources of electrical noise [1]. Traditionally, such soft errors were for the most part of concern when designing high accessibility frameworks regularly utilized as a part of mission-or life-critical applications. Space programs where an arrangement can't bear the cost of a glitch while in flight, are defenseless against soft errors. Nuclear power observing systems, where a single failure may bring about extreme destruction, and real-time systems, where a missed deadline can establish an inaccurate activity and a possible system failure, are a couple of different samples where soft errors are a serious issue. Soft memory error is one of the supreme intractable sources of faults which is such an event that corrupts the value stored in a memory cell deprived of damaging the cell itself [2-4].

Because of the vulnerability grows exponentially, processor manufacturers are starting to contain reliability as a first order concern on part with performance, power, in addition to cost. Software fault-tolerance techniques which deliver whole processor core protection with least execution time overhead do not need any hardware for execution and support application protection by building redundancy ly into the compiled code. These are beneficial because they have fewer requirements and lower costs but they may not provide adequate protection in all situations.

In order to discovering soft errors in programs via software techniques [5], various methodologies have as of now been proposed. These methodologies are proven effective of acknowledged types only but absence in providing high-coverage as well as low latency error detection, while the system is in operation state. Soft errors mitigating techniques mostly focuses on post design phases i.e. circuit level solutions, logic level solutions, error correcting code, spatial redundancy, etc. Some software based explanations evolving duplication of the whole program or duplication of instructions [6], critical variable re-computation in whole program [7], etc. are concerns of prior research. These techniques obtain fault coverage whilst degrading performance and several sources.

This paper offers a methodology for soft error detection and recovery through triple modular redundancy working only with variables instead of the entire program. Minimizing the amount of comparisons is the key point because of checking with only variables. The core contribution of this paper is that it identifies as well as recovers from soft errors in less time and space overhead with increased reliability.

The paper is prepared as follows: Section 2 represents the related works. The proposed methodology is described in Section 3. Section 4 provides the experimental setup and analysis. Finally, in Section 5, conclusions are drawn.

2. RELATED WORKS

There has been a great deal of work for addressing soft errors in processor and memory resources with both hardware and software methods. Three categories of soft error mitigation methods are highlighted among them so far which are (i) software based approaches (ii) hardware based approaches and (iii) hybrid (hardware and software combined) approaches.

To fault detection and recovery, software based methodologies can meaningfully increase reliability and are beneficial as a consequence of without necessitating any hardware modifications. The procedure of changing from a vulnerable arrangement to a dependable system just needs recompilation of the performing application. For tolerating soft errors, it creates redundant programs to detect [8-11] as well as recover from soft errors [12], duplicating instructions [13, 14] and tasks [15], double utilization of super scalar information ways [16] and so forth. Chip level Redundant Threading (CRT) [8] utilized a load value queue such that repetitive executions can simply see an indistinguishable perspective of memory. Error detection and Correction Codes (ECC) [17] enhances additional bits with the original bit sequence for detecting error. TRUMP (Triple Redundancy Using Multiplication Protection) interweaves the original program with an AN-encoded form of the program [20]. Error Detection by Diverse Data and Duplicated Instructions (EDDI) [13], and Software Implemented Fault Tolerance (SWIFT) [14] copies instructions in addition to program data to detect soft errors but then arises some overhead for the reason that both of them used redundant programs and duplicating instructions that generate higher memory requirements and increase register load. N-version programming (NVP), the procedure of using N independent modules to ensure the similar assignment, was originally produced to decrease faults in the system design process, by using different teams and compilation tools to improve a software system [20]. The technique called MASK (The method proposed by Jonathan et. al [20] to mask soft errors), enthusiastically implements invariants that can be demonstrated true statically.

By the side of a hardware level, fault tolerance is generally accomplished by copying individually hardware component. Hardware based methodologies commonly consist of circuit level, logic level in addition to architectural solutions. At the circuit level, gate sizing procedures, resistive hardening, increasing capacitance are generally used to rise the critical charge (Qcrit) of the circuit node. Radiation solidifying keeps up a repetitive duplicate of information which give the right information after a particle strikes along with support to recover corrupted section from the upset. Nonetheless, these procedures lead to bring down the speed and expansion power utilization of the circuit. Logic level solutions mostly propose individually detection and recovery in combinational circuits by using redundant or self-checking circuits. Architectural solutions present redundant hardware in the system making the whole system more robust against soft errors.

Hybrid methods are the combination of the features of software as well as hardware. These methods utilize the chip multiprocessors (CMPs) which have parallel processing capability and redundant multi-threading for detecting and recovering errors. Mohamed et al. [20] proposed Chip Level Redundantly Threaded Multiprocessor with Recovery (CRTR), where running every program for two times by means of two duplicate threads, on an instantaneous multi-threaded processor is the primary idea. In simultaneously and redundantly threaded processors with Recovery (SRTR) scheme [11], it is possible to fault corruption in two threads from the time while both the leading and trailing thread perform on the similar processor. These strategies may perhaps maintain not just since inherent execution overhead in addition to can emerge power in addition to time delays without proposing any execution pick up.

3. THE METHODOLOGY TO DETECT AND RECOVER THROUGH TMR

A methodology has been proposed to give fault tolerance with the purpose of reducing space and time plus improved reliability. The entire operational method consists of two main phases. For the duration of the first phase, the proposed technique detects either any soft error occurs or not. If a soft error occurred, the second phase takes necessary actions which are the recovery mechanisms. At the time of recovery action, erroneous variables are replaced by the originals with the help of majority voting.

3.1 Triple modular redundancy

To construct dependable systems, redundancy methods are usually used to assure high reliability, availability as well as data integrity. Triple Modular Redundancy (TMR) is a widely used redundancy technique that has the ability to mask faults. In a TMR system, same logic function is implemented in three ways and a voter circuit is used for voting their outputs. For voting on the outputs of the individual modules, majority voting circuits are mostly used in TMR systems.

3.2 Majority voting

Majority rule is a decision rule that chooses alternatives that have a majority, that is, more than half the votes. In a few TMR systems, the voters are also triplicated. Figure 1 shows the majority voting technique.

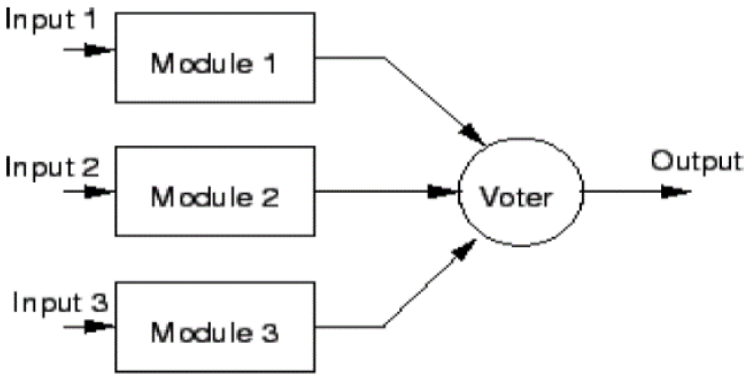

Figure. 1 Voting technique

In this technique, there is only a single voter—if the voter fails in such a system, then the complete system will fail. However, in a decent TMR system the voter is much more reliable than the other TMR components.

3.3 Error detection and correction methodology

The error detection and correction methodology is explained in Figure 2, First of all, program variables in the program are triplicated and then preceding variables are identified. Each operation has to be performed on triplicated copies of the variables. For detecting soft errors, the dependent variables are needed to be compared only. After error detection, majority voting is applied among the triplicated copies to replace the erroneous variable.

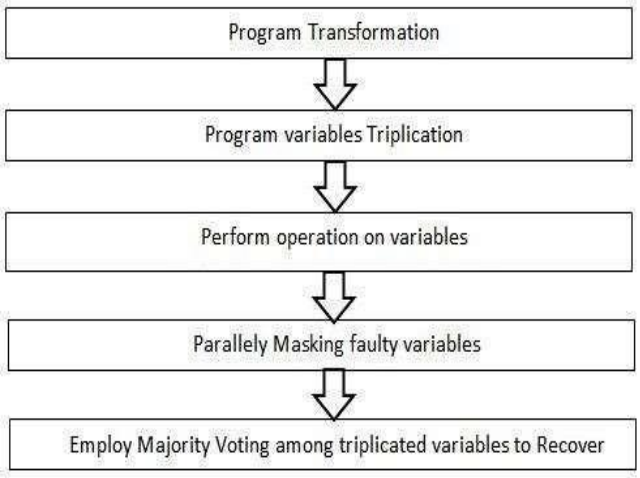

Figure. 2 The methodology to detect and correct errors

3.4 Soft error and recovery

First a simple .c file is uploaded just like Figure 3. Then, it is necessary to find the variables, assignment and arithmetic operators from that file and triplicate the variables as well as assign the original value of each variable into its redundant copies. Thus, we need to do some steps which procedures are shown below.

```
#include<stdio.h>
int main() {
int a, b, c;
a=10;
b=50;
c=12;
a=a+c;
b=a+b;
}
```

Figure. 3 Original code

a) Program variables triplication

It is necessary to find variables from our sample code and then does some operations to triplicate each variable. For doing that, we split the declaration line into some charactersand the do some string operations for triplicating each variable.

Procedure:

If we find a string which contains "int" (not "int main()") then

1. Split the string by "space" to get the list of variables.
2. Split the list of variables by "," to get the variables individually.
3. Spit the last element of the 2nd part by ";" to get exact variable.
4. After string operations, we get the two redundant copies of each variable.

Figure. 4 Procedure of finding variables and their triplication

Figure 4 shows an example to explain the whole procedure. In the original code, we find a line like that "**int a, b, c;**". Now using these steps, we retrieve our variables.

1st step, we split this string by "space" to get the list of variables. Thus, we find:

int a, b, c;

2nd step, we split the list of variables by ",". Thus, we find:

a	b	c;

3rd step, we spit the last element of the 2nd step by ";". Thus, we find:

a	b	c

4th step, we do some string operations to get the two redundant copies of each variable. So, we find

a1, a2	b1, b2	c1, c2

b) Perform operation on variables

Procedure:

If we find a string which contains "=" and not any arithmetic operator then
1. Split the string by "=".
2. Converting the string of the right of "="operator into integer.
3. Give the same value to the redundant copies of that variable.

Figure. 5 Operations on variables

Figure 5 shows an example to explain the whole procedure. In the original code, we find a line like that "**a=10;**" Now using these steps, we give the value into the redundant copies.

1st step, we split this string by "=". So we find

a	10

2nd step, we convert '10' which situates the right of "="operator into integer.

3rd step, we give the same value to the redundant copies of 'a'. So we find

a=10;
a1=10;
a2=10;

c) Perform operation on arithmetic operators

After triplicating each variable, now our task is to take the value of each variable and assign the value into its redundant copies.

Procedure:

If we find a string which contains "=" and any arithmetic operator then
1. Split the string by "=".
2. Split the string by of the right of arithmetic operator.
3. After string operations, we give the arithmetic operator into two redundant copies.

Figure. 6 Operation on arithmetic operators

Figure 6 shows an example to explain the whole procedure. In the original code, we find a line like that "**a=a+c;**" Now using this steps, we give the value into the redundant copies.

1st step, we split this string by "=". So we find

a	a+c

2nd step, we Split the string by of the right of "+" operator. So we find

a	c

3rd step, we do some string operations to give the arithmetic operator into two redundant copies. For 'a', we have already got two redundant versions 'a1' and 'a2' and for 'c', we have already got two redundant versions 'c1' and 'c2'. Thus, we get:

```
a=a+c;
a1=a1+c1;
a2=a1+c2;
```

3.5 Injecting fault

After triplicating all variables and performing operations on all assignment and arithmetic operators, it's necessary to inject fault into any variable's original or redundant copies. Then we apply our mechanism for detecting and correcting error. Figure 7(a) shows the transformed code of an assignment operator for variable 'b'. Figure 7(b) contains the same thing like Figure 7(a), but in Figure 7(b), we inject a fault into the triplicate copy of 'b2'.

```
int b, b1, b2;
b=50;
b1=50;
b2=50;
```

(a)

```
int b, b1, b2;
b=50;
b1=50;
b2=78;
```

(b)

Figure. 7 Injecting fault

3.6 Fault detection

For escaping storing inappropriate values in memory as well as storing values to incorrect addresses, the value of the original version of any operator must be equaled to their redundant copies. If any dissimilarity is identified between the original and redundant versions, then a fault has arisen and the appropriate handling code is executed, whether that be withdrawing or restarting the program or simply notifying another process. Figure 8 shows the algorithm for soft error detection.

```
Begin
    For each statement in program code
        Duplicate variables
        After operation compare the values of variables
          If result mismatch
            Then report Soft Error
    End Loop
End
```

Figure. 8 The algorithm for soft error detection

3.7 Soft error tolerance

For the soft error tolerance, instead of creating one redundant copy, the transformation creates two redundant copies. Three independent copies allow a fault to corrupt any one version's computation while the two other versions

can still have the correct computation. By using a simple majority voting scheme, any single-bit fault can be masked as two of the versions will still maintain the correct data.

Majority voting is the most important part of the proposed technique. In Figure 7(b), a fault is injected in 'b2' which is the redundant copy of variable 'b'. And Figure 9 shows the procedure of majority voting. It shows that the code first checks all the variables and their copies to find if any variable is faulty or not. If any faulty variable is detected, it is masked by comparing the values of redundant copies.

```
Suppose, Variable a, Duplicated a1, Triplicated a2
 If Error is detected, then
if a not equals a1, a not equals a2 and a1 equals a2 then
        Faulty variable = a and then a = a1 or a2
else if a1 not equals a, a1 not equals a2 and a equals a2 then
        Faulty variable = a1 and then a1 = a or a2
else
        Faulty variable = a2 and then a2 = a or a1
```

Figure. 9 Recover through majority voting

Now we apply this recovery technique in Figure 7(b) where a fault is injected in 'b2' which is the redundant copy of 'b'. Figure 10(a) is the same figure such like Figure 7(b). Figure 10(b) shows the recovery mechanism for Figure 10(a) and Figure 10(c) shows the code after recovery mechanism.

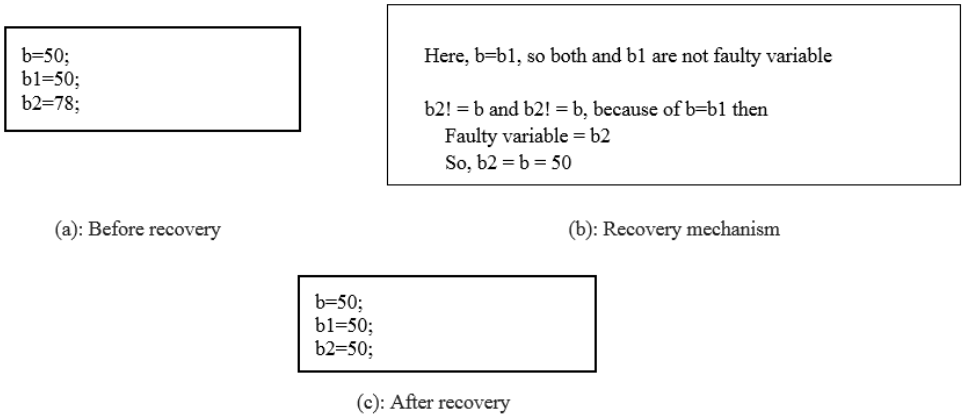

Figure. 10 Recovery process

The overall process for detection and recovery of soft error can be described through the following basic steps:

Step 1: A simple program is loaded and a new file is generated where all the variables are automatically triplicated.

Step 2: The values of all the original variables and redundant variables are obtained from the generated file.

Step 3: Error is injected randomly and manually to validate the methodology in all cases. In sense it is looked no error but actually an error was occurred but not noticed. But, for reliability it is checked in each time.

Step 4: Error is recognized through the checking of duplicated value with the original. If error, then recovery process is analyzed to get back to the original state.

Step 5: For the recovery purpose, majority voting is applied.

Step 6: After recovering, now all variables have their correct values.

4. EXPERIMENTAL ANALYSIS

For the purpose of evaluating the feasibility and effectiveness of the proposed technique, the method has faced through a number of tests. These tests are executed on some simple C Programs. The programs are analyzed and source code is modified according to the proposed technique. The task of simulation method is to detect soft error in the detection phase as well as to recover it in the recovery phase. To detect fault, variables are duplicated and then checked whether these are similar with the original one. After detection, a majority voting technique is applied to the triplicated variables with the intention of replacing the incorrect variable with the correct one.

The proposed methodology exhibits some optimistic results as it only deals with variables in a simple C program both in detection and recovery steps.

Figure 11(a) shows a simple C program with declaration and assignment operations. Then Figure 11(b) shows the transformed code of the original program which contains the redundant copies of each variable and redundant copies contain the value of its original copy. And the transformed code also contains the majority voting mechanism.

```
#include<stdio.h>
int main() {
   int a, b;
   a=10;
   b=20;
}
```

(a)

```
#include<iostream>
using namespace std;
int main() {
   int a, b;
   int a1, b1;
   int a2, b2;
   a=10;
   a1=10;
   a2=10;
   b=20;
   b1=20;
   b2=20;
   majority (a, a1, a2);
   majority (b, b1, b2);
}
```

(b)

Figure. 11 Simple C program and its output

Figure 12 (a) contains another C program as same as Figure 11 (a). But this time it contains some arithmetic operations. Before arithmetic operations, majority voting technique is applied because if any fault is injected, majority voting detects it and then recover it.

```
#include<stdio.h>
int main() {
   int c, d, e;
   c=5;
   d=12;
   e=7;
   c=c+d;
   d=c/e;
}
```

(a)

```
#include<iostream>
using namespace std;
int main() {
int c, d, e;
int c1, d1, e1;
int c2, d2, e2;
c=5;
c1=5;
c2=5;
 d=12;
d1=12;
d2=12;
 e=7;
 e1=7;
 e2=7;
 majority(c, c1, c2);
 majority(d, d1, d2);
 majority (e, e1, e2);
 c=c+d;
 c1=c1+d1;
 c2=c2+d2;
 d=c/e;
 d1=c1/e1;
 d2=c2/e2;
}
```

(b)

Figure. 12 Simple C program and its transformed form (with arithmetic operations)

For triplication, majority voting technique can detect error and recover from it. Thus, it needs lesser time. However, in duplication technique, it cannot recover from the error. So, a fresh copy is needed to be loaded again. For this reason, it needs more time than the proposed method. A. R. George et. al. [19] duplicated data that perform error detection only while the proposed method triplicated data to ensure error detection and correction as well. In order to recover from the error, by using the existing method (e.g., using the method proposed by A. R. George et. al. [19]), a fresh copy is needed to be loaded. Nevertheless, in the proposed method, if a soft error occurred, erroneous variables are replaced by the originals with the help of majority voting. The time taken by existing method for duplication process and the time taken by the proposed method for triplication process are shown in Figure 13.

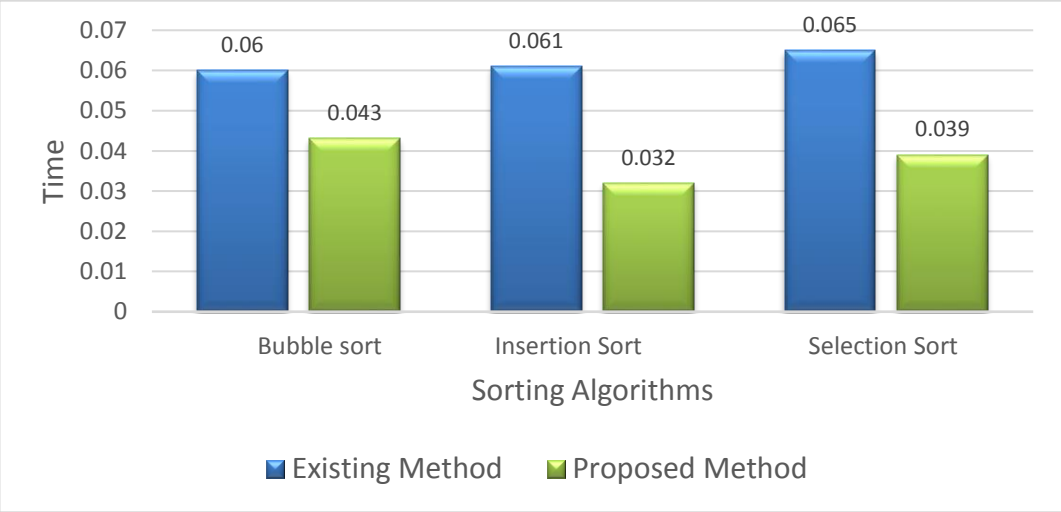

Figure. 13 The comparison between time overhead consumed by the existing method and the proposed method

From the above analysis, it is clear that the proposed methodology is time efficient with respect to existing method for soft errors tolerance. In the proposed method, error is masked by majority voting technique. However, in the existing technique, error is not recovered from the process. So, a fresh copy is needed to be loaded again which needs more time than the proposed method. Hence, the proposed method requires lesser time than the existing method to tolerate soft errors. The comparison between time overhead consumed by the existing method and that of the proposed method is shown in Figure 13 for different sorting algorithms.

5. CONCLUSIONS

It is noticeable that fault tolerance is a main consideration for the purpose of controlling software fault for the system, for that reason new approaches and variation of remaining techniques are required. The main contribution of the proposed method is to mitigate soft error threats with a minimum time and space complexity and enhance system performance and most essentially reliability since it works only with variables. Henceforth, all the variables in program code are not considered though they also may cause error that is from benign faults and faults that are not severe for the system; that does no interference to system performance.

From the time when it mechanisms with variable comparison, it is possible for certain opcode fault to change a regular instruction into a synchronizing instruction, in which case the program is susceptible to the control-flow issues. Even though protection presented by this technique is reasonably huge (with respect to time and space), there may perhaps have fault affects between the validation as well as use, nevertheless this case is quiet severe. However, variable comparison has the ability to escape large amount of comparisons and significantly decreases the execution time of the program along with memory utilization. TMR establishes software modulated fault tolerance in a cost effective technique that gives opportunity to the designers more flexibility in their application of fault detection and recovery techniques and inspires further into software modulated fault tolerance.

REFERENCES

1. J. Yang et al., "Radiation-Induced Soft Error Analysis of STT-MRAM: A Device to Circuit Approach," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 380-393, March 2016.
2. A. Timor, A. Mendelson, Y. Birk, and N. Suri, "Using underutilized CPU resources to enhance its reliability," *Dependable and Secure Computing, IEEE Transactions on*, vol. 7, no. 1, pp. 94-109, 2010.
3. E. L. Rhod, C. A. L. Lisboa, L. Carro, M. S. Reorda, and M. Violante, "Hardware and Software Transparency in the Protection of Programs Against SEUs and SETs," *Journal of Electronic Testing*, vol. 24, pp. 45-56, 2008.
4. S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*, San Francisco, CA, USA, pp. 243 - 247, pp. 243-7.
5. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin., "Dynamically discovering likely program invariants to support program evolution*", IEEE Transactions on Software Engineering*, 27(2):1-25, 2001, and 2005.
6. O. Nahmusk , M. Subhasis, E. j. McClusky, "EDD^{4}I: Error Detection b Diverse Data and Duplicated Instructions." *IEEE Transactions on Computers*, Vol. 51 No. 2 February 2002.
7. K. Pattabiraman, Z. Kalbarczyk, and R. K. lyer, "Critical Variable Precomputation for Transient Error Detection", 2008
8. S. S. Mukherjee, M. Kontz and S. K. Reinhardt "Detailed design and evaluation of redundant multi-threading alternatives" *In Proceeding of the 29th Annual International Symposium on Computer Architecture*, 2002, pp. 99-110.
9. S. K. Reinhardt and S. S. Mukherjee "Transient fault detection via simultaneous multithreading", *In Proceeding of the 27th International Symposium on Computer Architecture*, 2000, pp. 25-36.
10. E. Rotenberg "AR-SMT: A Micro Architectural Approach to Fault Tolerance in Microprocessors" *In Proceeding of the 29th Annual International Symposium on Fault-Tolerant Computing*, 1999, pp. 84-91.
11. J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk "Fingerprinting: Bounding soft-error detection latency and bandwidth" *In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, New York, NY 10036-5701, United States, 2004, pp. 224-234.
12. T. N. Vijaykumar, I. Pomeranz, and K. Cheng "Transient-fault recovery using simultaneous multithreading," *In Proceeding of the 29th Annual International Symposium on Computer Architecture*, pp. 87-98, 2002.
13. N. Oh, P. P. Shirvani, and E. J. McCluskey "Error detection by duplicated instructions in super-scalar processors" Reliability, *IEEE Transactions on*, vol. 51, pp. 63-75, 2002.
14. G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: software implemented fault tolerance" *In Proceeding of the International Symposium on Code Generation and Optimization*, Los Alamitos, CA, USA, 2005, pp. 243-54.
15. Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin "Reliability aware co-synthesis for embedded systems," *In Proceeding of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2004, pp. 41-50.
16. J. Ray, J. C. Hoe, and B. Falsafi "Dual use of superscalar data path for transient fault detection and recovery" *In Proceeding of the 34th ACM/IEEE International Symposium on Microarchitecture*, 2001, pp. 214-224.
17. C. L. Chen and M. Y. Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications: A State-Of-The-Art Review," *IBM Journal of Research and Development*, vol. 28, pp. 124-134, 1984.
18. K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," *in Proceedings of the International Symposium on Computer Architecture*, New York, NY 10016-5997, United States, 2007, pp. 516-527.
19. A. R. George, C. Jonathan , N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee "Software-Controlled Fault Tolerance." *ACM Transactions on Architecture and Code Optimization (TACO)*, December 2005.
20. C. Jonathan, A. R. George, and D. I. August "Automatic Instruction Level Software-Only Recovery." *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2006.

AUTHORS PROFILE

Muhammad Sheikh Sadi received B.Sc. Eng. in Electrical and Electronic Engineering from Khulna University of Engineering and Technology, Bangladesh in 2000, M.Sc. Eng. In Computer Science and Engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh in 2004, and completed PhD (Area: Dependable Embedded Systems) from Curtin University of Technology, Australia in 2010. He is currently Professor at the Department of Computer Science and Engineering, Khulna University of Engineering and Technology, Bangladesh. He teaches and supervises undergraduate and postgraduate theses in topics related to Fault Tolerant Computing, Embedded Systems, Digital System Design, Soft Errors Tolerance etc. He has published more than 50 papers and book chapters in his area of expertise. Muhammad Sheikh Sadi is a member of the IEEE since 2004.

Shaheena Sultana received B.Sc. Eng. in Electrical and Electronic Engineering from Khulna University of Engineering and Technology, Bangladesh in 2000, M.Sc. Eng. in Computer Science and Engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh in 2011. She is doing Ph. D. in Computer Science and Engineering at Bangladesh University of Engineering and Technology, Dhaka, Bangladesh. She is currently Assistant Professor at the Department of Computer Science and Engineering, Khulna University of Engineering and Technology, Bangladesh. Her research interest is in Graph Drawing, Graph Partitioning, VLSI Layout Algorithms, and Soft Computing. Shaheena Sultana is a member of the IEEE.

Md. Moin Al Rashid received B.Sc. Eng. in Computer Science and Engineering from Khulna University of Engineering and Technology, Bangladesh in 2016. He is currently working at Samsung Research & Development Institute Bangladesh LTD as a Software Engineer. He interests in contest programming and participates in various online programming contests.

Kazi Fahad Lateef received B.Sc. Eng. in Computer Science and Engineering from Khulna University of Engineering and Technology, Bangladesh in 2016. He is currently working at Samsung Research & Development Institute Bangladesh LTD as a Software Engineer