Detector Placement for Critical Variables Using Static Backward Slicing

Mehnuma Tabassum Omar, Seemanta Saha, Monika Gope and Ariful Islam Khandaker

Department of Computer Science and Engineering, Khulna University of Engineering & Technology (KUET), Khulna 9203, Bangladesh

Abstract- In this contemporaneous world, in order to fulfill consumers' limitless mandate for developing density, advancing functionality, and dominating power consumption, the dimensions and operating voltages of computer electronics are lessening day by day. This reduction in size raises sensitivity to radiation dramatically triggering soft error. If the radiation is large enough then even a single radiation may cause a stored data bit to be corrupted and flipped [1]. This deviation in the data values is termed as data errors. The consequence of the deviation is very vital on critical variables which consequently modify the system control flow prompting system failure [2]. Critical variables are characterized as those having high sensitivity to errors. This paper concentrates on critical variables for developing and employing error detectors to keep them away from data errors and provides high coverage for errors in any data value used in the program. The development is performed automatically through the support of backward slice of a program. The error detectors yield the method of checking expressions optimized in each control flow path surveyed at runtime [3].

Index Terms- Critical Variable, Detector, Program execution flow, Progam Slice, Backward Static Slice.

I. Introduction

Along with the enhancement of fabrication technology, the act of microprocessors has been amplified remarkably. The improved act of today's microprocessors is also being highlighted by reduced feature size, lesser operating voltage levels and condensed noise margins. On contemporary, it is evidenced that this progression on the integrated circuits with the advanced complexity of computer will accelerate chips more susceptible to transient faults which are temporary unintended changes of states arising from the latching of single event transient outcomes of external particle strikes or process-related parametric variation. Any system when affected by these transient faults generates soft error [2] leading towards a less reliable system. These kinds of errors are a vital threat to computer systems and becoming more significant with the development of technology. Soft errors are very critical as they do not arise consistently and cannot be expected as manufacturing faults or design faults. Soft errors affect both the control flow and data of a program. Recent researches verified that about 33% to 77% of transient faults are transformed to Control Flow Errors (CFE) and the rest transformed to data errors [4] [5]. Soft error interrupts the system's reliability and affects the data stored in memory. In current investigation it has been revealed that with a rate of about 5000 FIT per Mbit, the modern memory elements have been attacked by single-event upsets (SEU) [6]. Several software applications already infer fault tolerance moderately [7].

To deliver high-coverage at runtime for both the software and hardware errors, the Duplication technique has gained some popularity. The technique is executed by duplicating instruction at nominated program point such as stores or branches [8] and associating each instruction with the duplicated instruction. Although this employment may avert error propagation and block crashes, they are affected by high performance overhead. Along with this, it sacrifices coverage of program crashes before attaining the association point. Besides this, the native program and the reproduced program may subject to common mode errors, therefore proposing inadequate protection against software errors and permanent hardware faults.

For defending the system from common mode faults, a diverse execution method titled ED4I has been introduced in [9]. It is accomplished by implementing two different versions of the same program. ED4I is based on software and provides protection against transient and permanent hardware faults. The native program is multiplied by a constant factor ***k***. The two programs: the native program and the transmuted program; are then performed on the same processor and the outcomes of their performance are compared. The data operand set of the transmuted program is altered as compared to the native program hence it is capable of masking several types of hardware errors resides in processor functional units or in memory. ED4I separates the data values consumed by a program from the instructions processing that data values and employs diversity simply on the data values not on the instructions. As a consequence, ED4I cannot detect errors in instruction issue and decode logic [3].

To contract with the risk of SEUs and single-event transients (SETs), numerous hardware and software protection structures have been projected. These structures can handle the SEUs or SETs but employing these structures to each memory elements of a circuit or to entire portion of a software application, rises the costs extensively or diminishes the performance tremendously. Besides this, modern microprocessors cover a considerable amount of the transient faults completely that have no effect to the system termed as benign fault [10]. In this circumstance only those variables or registers having the highest dependency than the rest in the system are considered and delivers high coverage [11] notated as critical variable.

Fault detection and recovery technique can be classified as software-only technique, hardware-only technique and hybrid technique. Among them, as the software-only structures against fault detection and recovery do not involve any hardware alteration, they are beneficial in enlightening reliability of the system appreciably. Therefore, software-only structures like software redundancy techniques are considerably inexpensive, faster, and simpler to set up. In order to transmute the system from a susceptible state to a reliable state, they only entail recompilation of the running program. On the other hand, hardware-only techniques comprise a majority of fault protection schemes, demanding the acquisition and disposition of advanced machines. The techniques are also convenient for the application developer, distributor or end users in terms of making resolutions of when and how to raise reliability. On the contrary, in hardware-only organizations, reliability options are regulated by the hardware designer or manufacturer only. The absence of the necessities of hardware also permits software-only skills to be directed to organizations that are previously installed. Arrangement of redundancy structures to organizations that are previously installed may become significant as a result of inadequate approximation of the soft-error rate by designers or vagueness concerning the operational settings of the organization. Any modification in the functioning environment of a hardware results in perceptible consequences on reliability, requiring disposition of upgraded software redundancy structures [12].

This paper emphasizes on software-only technique. The methodology proposed by this paper is to develop runtime error detectors for a critical variable utilizing the idea behind static analysis of a computer program. The method employs detectors or checks into an application for defending it from crashes and delivering high-coverage to detect errors that consequence in application failures. The detector is derived by acting the backward slicing of a program.

The key point of this paper is to check the precedence variables of critical variables in a block of software program and putting checks or detectors. These extracted detectors estimate the values of critical variables in a different manner than the other previous methods, evaluated in the original program to avoid the occurrence of common mode errors.

The paper is organized in some sections. Section II will illustrate techniques related to this research. Section III will recite some terminology which will be used to analyze the underlying method. The proposed methodology will be illustrated with examples and techniques in section IV. Finally, section V will conclude this paper.

II. RELATED WORKS

In order to narrate the underlying methodology of this paper, certain research work has been discoursed here. First of all, some software, hardware and hybrid approaches have been revealed. Then, researches accomplished on critical variable have been recounted. After then, works on some slicing techniques has been stated. Some researches related to static analysis and dynamic analysis are also being studied in order to gain the basic idea behind them.

In order to lessen soft errors the some software centered methods like [13-16] for the recognition of soft errors including redundancy of given programs and/or [17], duplicating instructions [8] [18], task duplication [19], and Error detection and Correction Codes (ECC) [20] etc. for recuperating from soft errors. Other duplication techniques entitled SWIFT [18] and EDDI [8], replicate instructions along with the data of the program to identify soft errors. Nevertheless, they generate complex memory pre-requisite as well as escalate register load too. Alternative practice is Error detection and Correction Codes (ECC) [20] which attaches extra bits through the actual bit sequence to recognize error. But it necessitates further logic and calculations and becomes complicated while using into combinational logic blocks. In order to minimize soft error by means of hardware tactics, the most popular solutions are classified into circuit level, logic level and architectural solutions. The gate sizing techniques [21-23], increasing capacitance [24] [25] are certain circuit level solutions. The resistive hardening [26] is a circuit level solution that maintains the critical charge ($\mathbf{Q_{crit}}$) of the circuit node as favorable as possible. Nevertheless, these techniques lean towards low speed and high power consumption. [27] [28] are some of logic level solutions that practice redundant or self-checking circuits in order to offer recognition and repossession in combinational circuits. The architectural solutions like [8] [29] familiarize redundant hardware in the organization to become more vigorous over soft errors. [30] [31] [23] [17] remark some hybrid (Hardware and software combined) approaches. But, all of them are susceptible to soft errors in main areas.

The idea of preceding variables has been introduced [2] in order to mitigate soft error and generates critical variable. They illustrate critical variable and invent corresponding critical block which is the most vital area having highest dependency in a program. They also demonstrate that accumulate detection towards critical variable inside a critical block will provide high protection and high coverage from soft error. The preceding variable analysis technique exposed in [11] detects soft error with reduced time overhead. A FPGA-based system in [7] has been designated in order to spot the critical variable from a specified program. Investigation shows that the FPGA-based system can diminish the critical calculation errors needed in the assessed applications expressively. The paper projected in [32] designates an automated variable dependence analysis [33] [34] that can be acted as an analyzer of a program flow and functioned such as an error checker. The technique is also capable of tracing critical variables dynamically and automatically from the original program in presence of other variables.

Slicing of a given program [35] [36] involves recognizing the fragments of the program which may influence the values of an elected set of variables directly or indirectly. The slicing schema can be verified as inter-procedural program slicing [37], dynamic program slicing [38-40], forward and backward slicing [41], program slicing using dependence graph [42] [43]. As stated by the slicing procedures, the application programmer fix a point from the given program afterward the dependency is scrutinized. Therefore, the application programmers were in charge to catch on the precedence variables physically.

The static analysis methods [44-46] used predefined fault model to assess the program typically identified by common programming bugs. They trace errors through entire possible routes in a program. Typically it is very difficult to discover all possible routes of a program. Consequently, they direct towards inefficient recognitions owing to their estimation. The dynamic analysis technique stated in [47] [48] originates code invariants for example the linear associations between variables sets of a program, variable's steadiness and dissimilarities concerning two or more variables of the program. The key resolution of [47] is to exhibit the invariants establish to programmers. The exhibited invariants are generated with an illustrative input set that are not in this set established on the execution of the application. This could consequence in the invariants to become despoiled though the error is not present in the program [3].

III. BASIC TERMINOLOGY

Some terminologies which have been used to comprehend the proposed approach are as follows:

A. Critical Variable: The variables of a program are the most essential element for the program which affects the entire program execution flow. These variables are also responsible for keeping up the program processing. The variables are associated to other variables in a program implicitly or explicitly. A single variable of a program can be reliant on one or more than one variable of that program. The most reliant variable among the others is titled as the precedence to the others. The variables having consequence on the rest variables of the overall code are labeled as critical variables [2]. They demonstrate a prominent sensitivity for arbitrary data errors in the application. Any errors produced on these variables are expected to spread in various areas in the program triggering program malfunction. Therefore, they provide highest dynamic fan outs in the program and adding check for them provide high coverage to soft error [3]. In Fig. 1 q is a critical variable [32].

B. Program Slice: It is a program assessment technique. It confines the attention of analysis to definite sub sections of a program. The program slicing technique is capable of extracting statements from a given program that are pertinent to a particular computation.

```
int a = 4,c = 1,p = 6;
int x = 10,q,w = 40;
int y = x + c;
int z = y + a + 1;
int d = 5;
q = w + z;
```

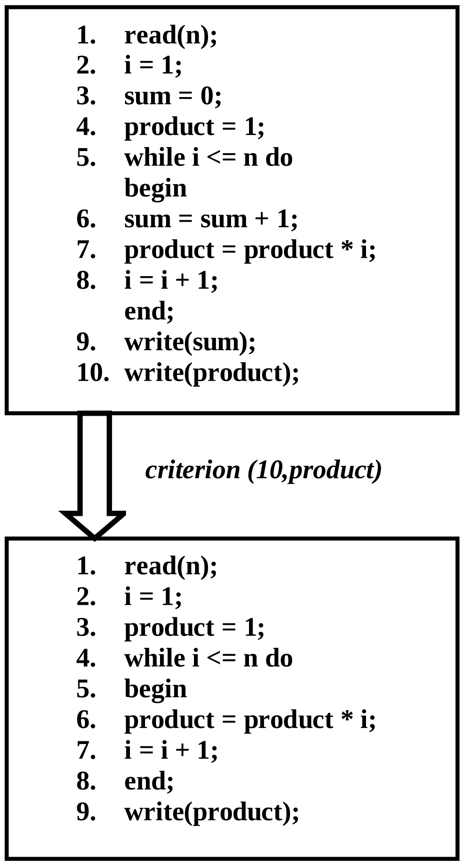
Fig 1: A Code Fragment

For this resolution, a point called slicing point is delivered by the programmer which may be before or after a precise statement. This statement contains the slicing variable. The slicing is performed either in forward direction or backward. The outcome of slicing may be corresponded to the program comprising some group of statements from the original program **[50]**.

C. Backward Static Slicing: Characteristically, a slicing principle comprising of a touple < S, V >, where S is the number of a particular statement and V is a slicing variable is acknowledged as static slicing. If the static slicing contains the input for the variable then it is transformed to dynamic slicing. The touple is termed as slicing criterion. If the slices are considered by accumulating statements and control predicates through a traversal of the program's control flow graph starting from the slicing criterion towards forward direction, then these slices are denoted as forward static slices. And if the traversal is achieved in backward direction it is referred as backward static slice [49]. Fig. 2 depicts the backward static slicing technique.

D. Checking Expression: An expression that will perform the verification of the value of a particular variable. The checking expression is computed by optimizing the expression related to a particular variable**.**

E. Detector: Expression that contains all the checking expression for a particular variable and placed across all possible path needed to access that particular variable


```
1.  read(n);
2.  i = 1;
3.  sum = 0;
4.  product = 1;
5.  while i <= n do
    begin
6.  sum = sum + 1;
7.  product = product * i;
8.  i = i + 1;
    end;
9.  write(sum);
10. write(product);
```

```
1.  read(n);
2.  i = 1;
3.  product = 1;
4.  while i <= n do
5.  begin
6.  product = product * i;
7.  i = i + 1;
8.  end;
9.  write(product);
```

Fig 2: Backward Static Slicing Schema

IV. DETECOR PLACEMENT USING BACKWARD STATIC SLING

The following steps are maintained to place detector for the critical variable in a program using the static analysis in backward direction.

A. ***Critical Variable Documentation:*** Numerous operations are already being performed to identify the critical variable. These operations are gaining more popularity as the identification of such variable can drastically reduce time to detect soft error. In variable dependency analysis, the variable having highest dependency is termed as critical variable. Utilizing critical variable to identify soft error in a program provides high detection coverage. Nevertheless, the proposed technique assumed that the critical variable identification is already performed.

B. ***Finding Checking Expression Using Backward Slice:*** As mentioned earlier, the checking expression for a critical variable will perform the verification of the data value of critical variable.

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.  int a , b ,c , d , e;
6.  b = 2,c = 4, d = 5;
7.  e = 3, g = 4 , t = 6;
8.  cout<<"\nValue for a : ";
9.  cin>>a;
10. f = b + e;
11. if(a == 0)
12. {
13. b = a + c;
14. e = b + g;
15. d = e + b;
16. f = d + b;
17. }
18. else if(a == 1)
19. {
20. d = b + c;
21. e = a + b;
22. f = d + e;
23. }
24. else
25. {
26. a = d - e;
27. c = a - d;
28. b = d + e;
29. e = g + t;
30. f = b - c;
31. }
32. cout<<f;
33. return 0;
34. }
```

Fig 3: Sample C++ Program

For this purpose, the backward slicing technique is performed. The slicing criterion comprising the critical variable along with its statement number is provided. The slicing is performed on each path that executes the critical variable directly or indirectly and provides the corresponding checking expression. Here each path inside a block is identified starting from the closing brackets '}' until an opening bracket '{' is found. For each path a stack is implemented to check the dependency of critical variable on the other variables of the program. Each expression related to critical variable is optimized using copy propagation or constant propagation.

Example: Let '***f***' is a critical variable in Fig. 3. The slicing criterion is ***V (31, f)***.

The 1st backward slice for ***V*** is the value of '***f***' coming from the '***else***' block. The checking expression for ***f*** is ***f = b – c***. Now the expression will be developed using the following optimization process:

$f = (b - c);$
$= (b - (a - d));$
$= (b - ((d - e) - d));$
$= (b - ((d - (g + t)) - d));$
$= ((d + e) - ((d - (g + t)) - d));$
$= ((d + (g + t) - ((d - (g + t)) - d))\ ;$

The 2nd backward slice for ***V*** is the value of '***f***' coming from the '***else if***' block. The current checking expression for the value ***f*** is ***f = d + e***. Now the expression will be developed as follows:

$f = (d + e);$
$= (d + (a + b));$
$= ((b + c) + (a + b));$

The 3rd backward slice for ***V*** is the value of '***f***' coming from the '***if***' block. Now the checking expression is of the form ***f = d + b.*** The expression will be settled as follows:

$f = (d + b);$
$= (d + (a + c));$
$= ((e + b) + (a + c));$
$= ((e + (a + c)) + (a + c));$
$= (((b + g) + (a + c)) + (a + c));$
$= ((((a + c) + g) + (a + c)) + (a + c));$

The 4th backward slice for ***V*** is the value of '***f***' coming from the '***main'*** block. The checking expression is ***f = b + e.*** As there is no dependency for this expression, the optimization is not needed here. Now the value for '***f***' will be renamed as '***f1***' and will be verified by their corresponding block.

C. ***Detector Insertion:*** After computing the checking expression along each path of the execution of critical variable, the detector is placed to the original program before the expression involving the critical variable is executed.

```
#include <iostream>
using namespace std;
int main()
{
int a , b ,c , d , e;
b = 2,c = 4, d = 5;
e = 3, g = 4 , t = 6;
cout<<"\nValue for a : ";
cin>>a;
//DETECTOR START
f1 = b + e;
if(a==0)
    f1=(((a+c)+g)+(a+c))+(a+c);
else if(a==1)
    f1=(b+c)+(a+b);
else
    f1=(d+(g+t))-((d-(g+t))-d);
//DETECTOR END
if(a == 0)
{
b = a + c;
e = b + g;
d = e + b;
f = d + b;
}
else if(a == 1)
{
d = b + c;
e = a + b;
f = d + e;
}
else
{
a = d - e;
c = a - d;
b = d + e;
e = g + t;
f = b - c;
}
if(f == f1)
{
cout<<f;
cout<<endl;
}
else
{
cout<<"\nError";
}
cout<<f;
return 0;
}
```

Fig 4: Program Output after Placing Detector

The detector for the critical variable ‘*f*’ is labeled with a new variable say ***‘f1’***. The situation when the value of the critical variable ***‘f’*** is different from the detector’s variable ***‘f1’***, it is determined as an error. Fig. 4 depicts the output of the program after placing detector. The value of ***‘f1’*** in the detector is checked before it can be used in any block. A change for to the value of a variable that affects the critical variable in any path or block can modify the value of the critical variable ***‘f’***. If ***‘f’*** is not equal to ***‘f1’*** then an error must be detected. For example the value of ***‘a’*** is ***‘0’*** and a single bit error within the ***‘if’*** block, causes the value of ***‘c’*** to be replaced by ***‘5’*** then the value of ***‘f’*** would be ***‘19’*** whereas the value for ***‘f’*** at the detector denoting by ***‘f1’*** within that block is ***‘16’***. So, the error is resolved.

V. CONCLUSION

In a program the variable that possesses most reliance on others is labeled as the precedence to the others. Among them, the variables consuming effect of the rest variables within the entire program are known as critical variables. They exhibit a conspicuous kindliness to data errors and deliver highest dynamic fan outs. Any faults formed by these variables are anticipated to extent in the entire coverage of the application triggering program malfunction. Therefore, adding detector for them is a wise decision providing high coverage to soft error. The instances that effect on critical may be contributed from different branch or path of a program. In this situation the detector should be a path-specific checker. The methodology portrayed in this paper contracts with this situation. To generate path-specific detector the structure performs a backward static slice. The slicing scheme is capable for analyzing dependency of the critical variable mentioned in the slicing criterion and thus delivers path-specific slicing which in terns generates the checking expression for the critical variable. Then the expression is optimized to create path-specific detector. The path-specific detector recompiles the critical variable among each path and associates the original value with the recompiled one. Any divergence of this association is detected by the detector. A single bit error in a variable resides in a specific branch that can modify the critical variable is detected by the path-specific detector.

REFERENCES

[1] Robert C. Baumann, Fellow, IEEE, “Radiation-Induced Soft Errors in Advanced Semiconductor Technologies”, IEEE TRANSACTIONS ON DEVICE AND MATERIALS RELIABILITY, VOL. 5, NO. 3, SEPTEMBER 2005.

[2] Muhammad Sheikh Sadi, Md. Mizanur Rahman Khan, Md. NazimUddin and Jan Jürjens. “An Efficient Approach towards Mitigating Soft Errors Risks,” Signal & Image Processing: An International Journal (SIPIJ) Vol.2, No.3, September 2011.

[3] Karthik Pattabiraman, Zbigniew Kalbarczyk and Ravishankar K. Iyer, “Automated Derivation of Application-aware Error Detectors Using Static Analysis: The Trusted Illiac Approach”, Dependable and Secure Computing, IEEE Transactions on (Volume:8 , Issue: 1), Page(44 – 57), ISSN : 1545-5971.

[4] N. Oh, P. P. Shirvani and E. J. McClusky, “Control Flow Checking by Software Signature,” IEEE Transaction on Reliability, Vol. 51, No. 2, 2002, pp. 111-122. doi:10.1109/24.994926.

[5] A. Mahmood, “Concurrent Error Detection Using Watch- dog Processors—A Survey,” IEEE Transaction on Computers, Vol. 37, No. 2, 1988, pp. 160-17 doi:10.1109/12.2145.

[6] Soft Errors in Electronic Memory - A White Paper. http://www.tezzaron.com/media/soft_errors_1_1_secure.pdf.

[7] Andreas Riefert, Jorg Muller, Matthias Sauer, Wolfram Burgard and Bernd Becker, “Identi cation of Critical Variables using an FPGA-based Fault Injection Framework”, VLSI Test Symposium (VTS), 2013 IEEE 31st,Page(1-6), ISSN : 1093-0167, DOI: 10.1109/VTS.2013.6548936.

[8] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability, 51(1):63--75, March 2002.

[9] N.S. Oh, S. Mitra and E.J. McCluskey, ED4I: Error Detection by diverse data and duplicated instructions in super-scalar processors. IEEE Transactions on Computers, 51(2):pp. 180-199, February 2002.

[10] N. Wang, J. Quek, T. Rafacz, and S. Patel, \Characterizing the effects of transient faults on a high-performance processor pipeline," in International Conference on Dependable Systems and Networks, pp. 61-70, 2004.

[11] Muhammad Sheikh Sadi, Md. Nazim Uddin, Bishnu Sarker andTanay Roy,” Minimizing Time Overhead to Detect Soft Errors through Preceding Variable Analysis”, 14th International Conference on Computer and Information Technology(ICCIT),Page(263–268),DOI: 10.1109/ICCITechn.2011.6164795.

[12] GEORGE A. REIS III, “SOFTWARE MODULATED FAULT TOLERANCE”, A DISSERTATION PRESENTED TO THE FACULTYOF PRINCETON UNIVERSITYIN CANDIDACY FOR THE DEGREEOF DOCTOR OF PHILOSOPHY, ADVISER: PROFESSOR DAVID I. AUGUST, June 2008.

[13] 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.

[14] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in 27th International Symposium on Computer Architecture, 2000, pp. 25-36.

[15] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," In Proceeding of the 29th Annual International Symposium on Fault-Tolerant Computing, 1999, pp. 84-91.

[16] 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.

[17] 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, 2002, pp. 87-98.

[18] 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.

[19] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reliability-aware co-synthesis for embedded systems," in 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004, pp. 41-50.

[20] 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.

[21] J. K. Park and J. T. Kim, "A Soft Error Mitigation Technique for Constrained Gate-level Designs," IEICE Electronics Express, vol. 5, pp. 698-704, 2008.

[22] N. Miskov-Zivanov and D. Marculescu, "MARS-C: modeling and reduction of soft errors in combinational circuits," in Proceedings of the Design Automation Conference, Piscataway, NJ, USA, 2006, pp. 767-772.

[23] Z. Quming and K. Mohanram, "Cost-effective radiation hardening technique for combinational logic," in Proceedings of the International Conference on Computer Aided Design, Piscataway, NJ, USA, 2004, pp. 100-106.

[24] Oma, M. a, D. Rossi, and C. Metra, "Novel Transient Fault Hardened Static Latch," in IEEE International Test Conference (TC), Charlotte, NC, United states, 2003, pp. 886-892.

[25] S. P. Release. New chip technology from STmicroelectronics eliminates soft error threat to electronic systems, Available at:http://www.st.com/stonline/press/news/year2003/t1394h.htm [Online].

[26] M. Zhang, "Analysis and design of soft-error tolerant circuits," Ph.D., Ph.D. Thesis, University of Illinois at Urbana-Champaign, United States -- Illinois, 2006.

[27] M. Zhang, S. Mitra, T. M. Mak, N. Seifert, N. J. Wang, Q. Shi, K. S. Kim, N. R. Shanbhag, and S. J. Patel, "Sequential Element Design With Built-In Soft Error Resilience," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 14, pp. 1368-1378, 2006.

[28] T. M. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in 32nd Annual International Symposium on Microarchitecture, 1999, pp. 196-207.

[29] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower, "IBM z990 soft error detection and recovery," Device and Materials Reliability, IEEE Transactions on, vol. 5, pp. 419-427, 2005.

[30] B. T. Gold, J. Kim, J. C. Smolens, E. S. Chung, V. Liaskovitis, E. Nurvitadhi, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "TRUSS: a reliable, scalable server architecture,"Micro, IEEE, vol. 25, pp. 51-59, 2005.

[31] A. G. Mohamed, S. Chad, T. N. Vijaykumar, and P. Irith, "Transient-fault recovery for chip multiprocessors," IEEE Micro, vol. 23, p. 76, 2003.

[32] Muhammad Sheikh Sadi, Linkan Halder and Seemanta Saha,” Analyzing Variable Dependency in Multi-bit Errors Detection”, 1st International Conference on Electrical Information and Communication Technology (EICT 2013), 978-1-4799-2299-4/13/$31.00 ©2013 IEEE.

[33] K. Muthukumar, M. Hermenegildo, “Compile-Time Derivation of Variable Dependence Using Abstract Interpretation,” Journal of Logic Programming, Vol. 13, N. 2-3.

[34] Chris Fox, Mark Harman, Youssef Hassoun. Variable Dependence Analysis Technical Report: TR-10-0, Elsevier 2010.

[35] Mark Weiser. “Program slicing,” IEEE Transactions on Software Engineering, 10(4):352–357, 1984.

[36] D. W. Binkley and K. B. Gallagher. “Program slicing,”. In M. Zelkowitz, editor, Advances in Computing, Volume 43, pp. 1–50.Academic Press, 1996.

[37] Susan Horwitz, Thomas Reps, and David Wendell Binkley. “Inter procedural slicing using dependence graphs,” ACM Transactions on Programming Languages and Systems, 12(1):26–61, 1990.

[38] H. Agrawal and J. R. Horgan “Dynamic program slicing,” In ACMSIGPLAN” Conference on Programming Language Design and Implementation, pp. 246–256, New York, June 1990.

[39] R. Gopal. “Dynamic program slicing based on dependence graphs,” In IEEE Conference on Software Maintenance, pp. 191–200, 1991.

[40] B. Korel and J. Laski.“Dynamic program slicing,” Information Processing Letters, 29(3):155–163, Oct. 1988.

[41] Dave Binkley, Sebastian Danicic, Tibor Gyim´ othy, Mark Harman, Akos Kiss Lahcen Ouarbya. “Formalizing Executable Dynamic and Forward Slicing,” Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM‟04).

[42] S. Horwitz, T. Reps, and D. W. Binkley. Inter procedural slicing using dependence graphs. In ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in SIGPLAN Notices, 23(7), pp.35–46, 1988.

[43] S. Horwitz, T. Reps, and D. W. Binkley. “Inter procedural slicing using dependence graphs,” ACM Transactions on Programming Languages and Systems, 12(1):26–61, 1990.

[44] Bush, W.R., J.D. Pincus, and D.J. Sielaff, A static analyzer for finding dynamic programming errors. Software Practice and Experience, 2000. 30(7): p. 775-802.

[45] Das, M., S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. in Proceedings of the ACM SIG-PLAN 2002 Conference on Programming language design and implemen-tation. 2002. Berlin, Germany: ACM Press.

[46] Evans, D., J. Guttag, J. Horning, and Y.-M. Tan. LCLint: a tool for using specifications to check code. in 2nd ACM SIGSOFT sympo-sium on Foundations of software engineering. 1994. New Orleans, Loui-siana, United States: ACM Press.

[47] Ernst, M.D., J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. in 21st international conference on Software engineering. 1999.

[48] Hangal, S. and M.S. Lam. Tracking down software bugs using automatic anomaly detection. in 24th International Conference on Software Engineering. 2002. Orlando, Florida: ACM Press.

[49] N.Sasirekha, A.Edwin Robert and Dr.M.Hemalatha, “PROGRAM SLICING TECHNIQUES AND ITS APPLICATIONS”, International Journal of Software Engineering & Applications (IJSEA), Vol.2, No.3, July 2011.

AUTHORS

First Author—Mehnuma Tabassum Omar, Khulna University of Engineering & Technology, misty2409@gmail.com

Second Author— Seemanta Saha,, Khulna University of Engineering & Technology, seemantasaha@gmail.com

Third Author— Monika Gope, Khulna University of Engineering & Technology, gopenath14@yahoo.com

Fourth Author— Ariful Islam Khandaker, Khulna University of Engineering & Technology, aikhandaker@yahoo.com