Mitigating Soft Error Risks through Protecting Critical Variables and Blocks

[1]Muhammad Shaikh Sadi, [1]Md. Nazim Uddin, [1]Md. Mizanur Rahman Khan, [2]Jan Jürjens

[1]Khulna University of Engineering and Technology (KUET), Khulna, Bangladesh,
[2]TU Dortmund, Germany
[1]sheikhsadi@gmail.com, [1]nazim.cse.kuet@gmail.com, [1]rk_mizan@yahoo.com,
[2]jan.jurjens@cs.tu-dortmund.de

Abstract: Down scaling of CMOS technologies has resulted in high clock frequencies, smaller features sizes and low power consumption. But it reduces the soft error tolerance of the VLSI circuits. Safety critical systems are very sensitive to soft errors. A bit flip due to soft error can change the value of critical variable and consequently the system control flow can completely be changed which may lead to system failure. To minimize the risks of soft error, this paper proposes a novel methodology to detect and recover from soft error considering only 'critical code block' and 'critical variable' rather than considering all variables and/or blocks in the whole program. The proposed method reduces space and time overhead in comparison to existing dominant approach.

Keywords: *Soft Errors, Safety Critical System, Critical Variable, Critical Block, Criticality analysis, Risk mitigation.*

1. Introduction

Temporary unintended change of states resulting from the latching of single-event transient create transient faults in circuits and when these faults are executed in system, the resultant error is defined as soft error. Soft error involves changes to data, i.e. charge in a storage circuit, as instance; but not physical damage of it [1], [2], [3]. The undesired change due to these errors to the control flow of the system software may be proved catastrophic for the desired functionalities of the system. Specially, Soft error is a matter of great concern in those systems where high reliability is necessary [4], [5], [6]. Space programs (where a system cannot malfunction while in flight), banking transaction (where a momentary failure may cause a huge difference in balance), automated railway system (where a single bit flip can cause a drastic accident) [7], mission critical embedded applications, and so forth, are a few examples where soft errors are severe.

Many static approaches [33] have been proposed so far to find soft errors in programs. These approaches are proven effective in finding errors of known types only. But there is still a large gap in providing high-coverage and low-latency (rapid) error detection to protect systems from soft error while the system is in operation. Soft errors mitigating techniques mostly focuses on post design phases *i.e.* circuit level solutions, logic level solutions, error correcting code, spatial redundancy, etc., and some software based solutions evolving duplication of the whole program or duplication of instructions [8], Critical variable re-computation in whole program [9], etc. are concerns of prior research. Duplication seems to provide high-coverage at runtime for soft errors; it makes a comparison after every instruction leaving high performance overhead to prevent error-propagation and resulting system crashes.

This paper proposes a novel approach for soft error detection and recovery technique. It works with critical blocks and/or critical variables rather than the whole program. Critical blocks and/or variables are defined as the coding segments which impacts the overall control flow of the program. Identification of these blocks and advancing with it is the key concept of the proposed method. Only critical blocks and variables are considered since other variables in program code may cause such error that is from benign faults and faults that are not severe for the system. The main contribution of this paper is that it detects and recovers from soft errors in less time and space overhead in comparison to existing dominant approaches.

The paper is organized as follows: Section 2 describes related works. Section 3 outlines the methodology. Experimental Analysis and conclusions are depicted in Sections 4 and 5, respectively.

2. Related Works

Three types of soft errors mitigation techniques are highlighted so far; (i). Software based approaches, (ii). Hardware based approaches and (iii). Hardware and software combined (hybrid) approaches.

Software based approaches to tolerate soft errors include redundant programs to detect and/or recover from the problem, duplicating instructions [12], [13], task duplication [14], dual use of super scalar data paths, and Error detection and Correction Codes (ECC) [15]. Chip level Redundant Threading (CRT) [11] used a load value queue such that redundant executions can always see an identical view of memory. Walcott et al. [29] used redundant multi threading to determine the architectural vulnerability factor, and Shye et al. used process level redundancy to detect soft errors. In redundant multi threading, two identical threads are executed independently over some period and the outputs of the threads are compared to verify the correctness. EDDI [28] and SWIFT [13] duplicated instructions and program data to detect soft errors. Both redundant programs and duplicating instructions create higher memory requirements and increase register pressure. Error detection and Correction Codes (ECC) [15] adds extra bits with the original bit sequence to detect error. Using ECC to combinational logic blocks is complicated, and requires additional logic and calculations with already timing critical paths.

Hardware solutions for soft errors mitigation mainly emphasize circuit level solutions, logic level solutions and architectural solutions. At the circuit level, gate sizing techniques [16], [17], [18] increasing capacitance [19], [20], resistive hardening [21] are commonly used to increase the critical charge (Q_{crit}) of the circuit node as high as possible. However, these techniques tend to increase power consumption and lower the speed of the circuit. Logic level solutions [30], [31] mainly propose detection and recovery in combinational circuits by using redundant or self-checking circuits. Architectural solutions mainly introduce redundant hardware in the system to make the whole system more robust against soft errors. They include dynamic implementation verification architecture (DIVA) [22].

Hardware and software combined approaches [23], [24], [25], [26] use the parallel processing capacity of chip multiprocessors (CMPs) and redundant multi threading to detect and recover the problem. Mohamed et al. [32] shows Chip Level Redundantly Threaded Multiprocessor with Recovery (CRTR), where the basic idea is to run each program twice, as two identical threads, on a simultaneous multithreaded processor. One of the more interesting matters in the CRTR scheme is that there are certain faults from which it cannot recover. If a register value is written prior to committing an instruction, and if a fault corrupts that register after the committing of the instruction, then CRTR fails to recover from that problem. In Simultaneously and Redundantly Threaded processors with Recovery (SRTR) scheme [26], there is a probability of fault corrupting both threads since the leading thread and trailing thread execute on the same processor. However, in all cases the system is vulnerable to soft error problems in key areas.

In contrast, the complex use of threads presents a difficult programming model in software-based approaches while in hardware-based approaches, duplication suffer not only from overhead due to synchronizing duplicate threads but also from inherent performance overhead due to additional hardware. Moreover, these post-functional design phase approaches can increase time delays and power overhead without offering any performance gain.

3. Methodology to Detect and Recover from Soft Errors

A novel methodology has been proposed in this paper to mitigate soft error risk. In this method, the major working phenomenon is a two-phase approach. During 1st phase, the proposed method detects soft errors at only critical blocks and critical variables. At 2nd phase, the recovery mechanism goes to action by replacing the erroneous variable or code block with originals. The program that is to be executed is split into blocks; among them, 'critical blocks' (*definition of critical block is detailed in section 3.2*) are identified. As Figure 1 illustrates, while the program is being executed, if critical blocks are encountered, they are treated specially; that is, a procedure is invoked; the critical variables within are computed twice to get two outcomes. If the comparing-mechanism finds the results identical, the ordinary execution flow of the program continues from the next block. Otherwise, variable is noticed to be erroneous by dint of soft error. The recovery process is to replace the erroneous critical block with the original program that is backed up earlier.

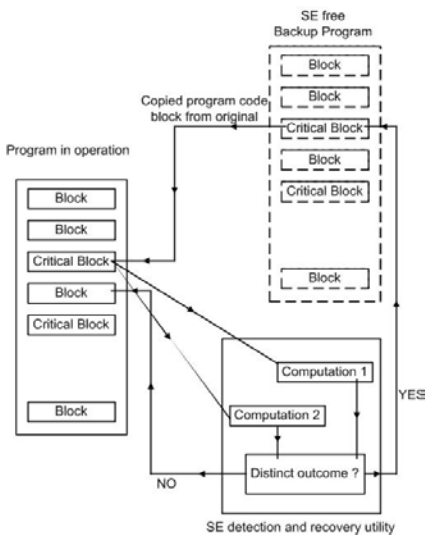


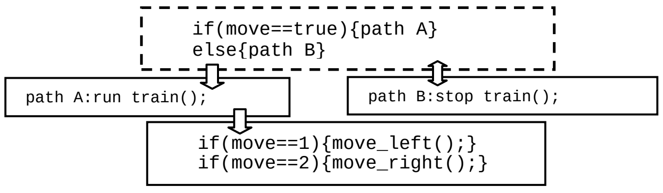
Figure 1. Soft Error Detection and Recovery

3.1 Backing up the program that is in operation

The backup of the functioning program is kept in memory for further working. This backup is assumed to be soft error free. Diverse technique (ECC, Parity, RAID, CRC etc.) survives to preserve consistency of the backup.

3.2 Identification of the Critical Variables and Blocks

Critical variables provide high coverage for data value errors. By dint of Critical variables, program execution flow is determined assuming erroneous values diversified from original, leads to erroneous outcome of program.



```
if(move==true){path A}
else{path B}
```

```
path A:run train();
```

```
path B:stop train();
```

```
if(move==1){move_left();}
if(move==2){move_right();}
```

Figure 2. Critical Block

Critical Blocks are defined as the programming segments which control the overall program flow. Criticality is determined analyzing diverse criteria (‘fan in’ and ‘fan out’, lifetime, functional dependencies, weight in conditional branches [34] etc.). Identification of these blocks and advancing with the critical blocks and/or variables noticed within it, is the key concept of the proposed method.

The code blocks responsible for branching program control flow are recognized to be critical-blocks. The dashed block in Figure 2 is example of Critical blocks within a program segment. Critical blocks decide which of the distinctive paths will be followed.

3.3 Computation of Critical blocks and Comparing Outcome

While executing a full program code, each critical blocks and/ or variables is computed twice and then compared two distinct outcomes to determine their consistency. The basic steps can be stated as follows:

Step 1: Each critical block is recomputed (executed twice and outcomes are stored).
Step 2: Recomputed results are compared to make sure that they are identical.
Step 3: If recomputed values show consistency, program will be continued from the next code block and no soft error will be reported.
Step 4: If recomputed values show inconsistency, program block will be identified as erroneous and soft error will be reported; and then the recovery procedure will be called for. Erroneous critical block will be replaced by the relevant original program’s critical block. And program execution will be continued from current block.

4. Experimental Analysis

The proposed method is experimented through a multi phases simulation process that detects the soft error occurred through the detection phase and duly recovers it in order to lead the program towards expected output with its counterpart; the recovery process. Backup of the operational program is kept in memory for soft error recovery process. A candidate program is checked through the simulator to detect soft error and duly reports it if there is any. Block wise execution of core program along with twin computation of critical one is helped by this backup to go to the desired end by supporting the blocks to be actual valued. A binary representation of the executable program is formed and lunched on the simulator editor as hexadecimal format. They are sequence of bit stream to negotiate with. Error is injected manually that is flipping a bit/ bits to change the original code sequence.

4.1 Fault Injection

Error is injected manually to change the original program. Error injection evolves bit flipping; this is to change a binary ‘0’ (zero) to a binary ‘1’ (one) and vice-versa at any bit position of a particular byte. Fault is injected at variables and/ or at any random position (instructions or variables) of the program’s binary file to change the value of the variables or the instructions. Suppose binary representation of a variable is 01000110, bit flipping may occur at any bit position due to soft errors. Suppose bit position 5 will be flipped, 0 to 1 then the value of the variable will be 01001110 that will cause a huge difference of the value.

4.2 Soft Error Detection

The critical blocks come into main focus among all other program-blocks. While executing, they are computed twice and compared by values they contain. Error is detected if the comparison is distinguished-valued. As soft error is transitory, it has neither repetitive occurrence nor long lasting effect. Hence, consecutive computation of variables results different result if any of them assumes erroneous value that should not to be. If no such case is encountered, ‘no soft error’ is reported.

4.3 Recovery from the effect of Soft Error

The previously noticed erroneous critical variable and its location is traced; and then, the backup is invoked to replace the erroneous critical code block with relevant originals. Recovery tool activates mechanisms to perform recovery tasks.

4.4 Result Analysis

The methodology is found to deal with critical blocks and/ or critical variables other than going through the whole program. This makes it less time consuming to perform computation (as shown in Figure 3).

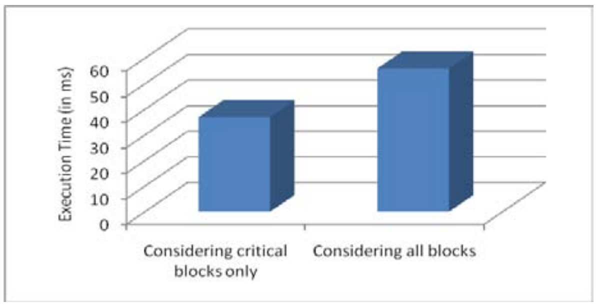


Figure 3. Comparison of Execution Time

Table 1 depicts a theoretical comparison among different soft-error tolerance techniques.

Table1. Review of different soft-error tolerance techniques

Approaches	***Diverse Data and Duplicated Instruction ED^{4}I [8]***	***Critical Variables Re-computation for Transient Error Detection [9]***	***Source Code Modification [10], [27]]***	***Proposed Method***
Adopted methodology	Original program and transformed program are both executed on same processor and results are compared.	Re-computes only critical variables (not the instructions) to detect and recovery from soft errors.	Based on modifications of source code. Protection methods are applied at intermediate representation of source code.	Detection is performed by critical blocks re-computation. Erroneous blocks are replaced by relevant backed up blocks.
Memory space overhead	At least double	Depends on no. of Critical variable.	Larger than usual based on modification scheme	Depends on no. of Critical Blocks
Execution time overhead	Longer than usual since comparison	Longer than usual since CV re-computation	High running time since source code modification	Relatively much lower
Number of variables to be executed	Double	Depends on no. of Critical variable.	Depends on modification techniques	Depends on no. of Critical Blocks
Drawback	Program may crash before reaching comparison point. Both the programs may be erroneous.	Unable to detect all severe errors since it works only with critical blocks. Instructions may also be erroneous.	System may crush before reaching the control flow checking point.	Improper identification of critical blocks leads to inefficiency.

Treating with fewer blocks/ variables requires lower memory space. Figure 4 illustrate the memory utilization of proposed method and whole program duplication methods with respect to time. Proposed approach consumes lesser memory space than existing leading whole program duplication approaches.

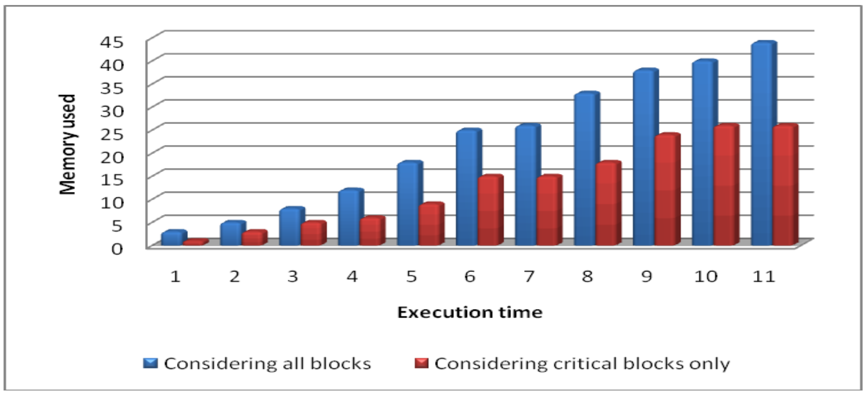


Figure 4. Comparison of memory utilization

5. Conclusions

This paper reflects the possibility of modification of existing methodologies to lower the criticalities of program blocks to minimize the risk of soft errors. The significant contribution of the paper is to lower soft error risks with a minimum time and space complexity since it works only with critical variables in critical blocks; hence, all the variables in program code are not considered to be recomputed or replicated though they also may cause such error, that is from benign faults and faults that are not severe for the system; which does no interference to system performance. It is seen that only critical variables induced soft error affects systems program flow in a great extent to be malfunctioned. Hence, leaving some ineffective errors un-pursued, the proposed method can achieve the goal in a cost effective (with respect to time and space) way.

Some possible steps could be adopted to enhance the performance of the method. Storing the recomputed outcomes of the critical blocks at cache memory will enhance memory access time, which can be a significant issue in case of memory latency. This can make the proposed method less time consuming by compensating the time killing to make double computation. The protection of backup of original program is a great concern to remain it soft error free. In support of storage, besides existing techniques such as Error-Correcting-Code (ECC), Redundant Array of Inexpensive Disks (RAID), enhancement can be explored. Another considerable concern is the critical blocks and critical variables. Obviously critical blocks and critical variables identification among numerous code blocks in operating program is a great challenge to make it optimum. Efficiency of the proposed method mostly depends on proper identification of critical blocks and critical variables. They can be grouped according to their criticality that is treated differently in different view-points. Several issues like "*fan out*", that is number of dependency/ branches exist; number of "*recursion*"- that is, looping or how many times a call repeated; "*severity of blocks*", that is block containing more weighted variables etc., are wide open to determine the criticality.

Hence, much more scopes are available in the field of critical block and variable identification. In contrast, proper identification of these critical blocks and variables can mitigate the soft error risks with optimal time and space requirement.

References

[1] A. Timor, A. Mendelson, Y. Birk, and N. Suri, "Using under utilized CPU resources to enhance its reliability," Dependable and Secure Computing, IEEE Transactions on, vol. 7, no. 1, pp. 94-109, 2010.

[2] 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.

[3] 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, 2005, pp. 243-7.

[4] R. K. Iyer, N. M. Nakka, Z. T. Kalbarczyk, and S. Mitra, "Recent advances and new avenues in hardware-level reliability support,"Micro, IEEE, vol. 25, pp. 18-29, 2005.

[5] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs,"Computer, vol. 39, pp. 118-120, 2006.

[6] S. Tosun, "Reliability-centric system design for embedded systems," Ph.D. Thesis, Syracuse University, United States --New York, 2005.

[7] Muhammad Sheikh Sadi, D. G. Myers, Cesar Ortega Sanchez, and Jan Jurjens, “Component Criticality Analysis to Minimizing Soft Errors Risk.” Comput Syst Sci & Eng (2010), vol 26 no 1 September 2010.

[8] Nahmsuk Oh, Subhasis Mitra, Edward j. McClusky, “ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions.” EEE Transactions on Computers Vol. 51 No. 2 February 2002

[9] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer, “Critical Variable Recomputation for Transient Error Detection”, 2008

[10] Adam Piotrowski, Dariusz Makowski, Grzegorz Jabło´nski, Andrzej, Napieralski, “The Automatic Implementation of Software Implemented Hardware Fault Tolerance Algorithms as a Radiation-Induced Soft Errors Mitigation Technique” Nuclear Science Symposium Conference Record, IEEE, 2008

[11] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multi-threading alternatives," in 29th Annual International Symposium on Computer Architecture, pp. 99-110, 2002, pp. 99-110.

[12] 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.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: software implemented fault tolerance," Los Alamitos, CA, USA, 2005, pp. 243-54.

[14] 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.

[15] 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.

[16] 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.

[17] N. Miskov-Zivanov and D. Marculescu, "MARS-C: modeling and reduction of soft errors in combinational circuits," Piscataway, NJ, USA, 2006, pp. 767-72.

[18] Z. Quming and K. Mohanram, "Cost-effective radiation hardening technique for combinational logic," Piscataway, NJ, USA, 2004, pp. 100-6.

[19] Oma, M. a, D. Rossi, and C. Metra, "Novel Transient Fault Hardened Static Latch," Charlotte, NC, United states, 2003, pp. 886-892.

[20] P. R. STMicroelectronics Release, "New chip technology from STmicroelectronics eliminates soft error threat to electronic systems," Available at www.st.com/stonline/press/news/year2003/t1394h.htm, 2003.

[21] L. R. Rockett, Jr., "Simulated SEU hardened scaled CMOS SRAM cell design using gated resistors," IEEE Transactions on Nuclear Science, vol. 39, pp. 1532-41, 1992.

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

[23] 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.

[24] S. Krishnamohan, "Efficient techniques for modeling and mitigation of soft errors in nanometer-scale static CMOS logic circuits," Ph.D. Thesis, Michigan State University, United States -- Michigan, 2005.

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

[26] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in 29th Annual International Symposium on Computer Architecture, 2002, pp. 87-98.

[27] Adam Piotrowski, Szymon Tarnowski, "Compiler-level Implementation of Single Event Upset Errors Mitigation Algorithms."

[28] 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.

[39] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," New York, NY 10016-5997, United States, 2007, pp. 516-527.

[30] M. Z. S. Mitra, N. Seifert, TM Mak and K. Kim. Soft and IFIP, "Soft Error Resilient System Design through Error Correction,"*VLSI-SoC,* January, 2006.

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

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

[33] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on Software Engineering, vol. 20, pp. 476-93, 1994.

[34] Salma Bergaoui, Pierre Vanhauwaert, and Regis Leveugle, "A New Critical Variable Analysis in Processor-Based Systems," IEEE Transactions, 2010