UOT 681.332

A Survey of Software Implemented Hardware Fault Tolerance Techniques*

T. H. KAZIMOV, J. M. SHAHRUKH

Azerbaijan National Academy of Sciences, Institute of Information on Technology, Baku
E-mail: depart9@iit.ab.az

This paper surveys various Software Implemented Hardware Fault Tolerance (SIHFT) techniques and methodologies. A major concern in digital electronics used in space is radiation-induced transient errors. Radiation hardening is an effective yet costly solution to this problem. Commercial off-the-shelf (COTS) components have been considered as a low-cost alternative to radiation-hardened parts. We assess the effectiveness of Software-Implemented Hardware Fault Tolerance (SIHFT) techniques in enhancing the reliability of COTS. This paper aims at providing designers and researchers with an overview of the available SIHFT technique.

Index Items: *SIHFT, transient errors, error detection mechanisms, fault detection, SEU.*

В данной работе рассматриваются программные обеспечения SIHFT (Software Implemented Hardware Fault Tolerance) различных методов и технологий, обеспечивающих устойчивость аппаратных средств к ошибкам. Временные ошибки, вызванные радиацией, занимают особое место в цифровой электронике. Радиационное укрепление аппаратных средств — эффективное, но все же дорогостоящее решение этой проблемы. Компоненты коммерческого стандарта COTS (Commercial off-the-shelf) являются дешевой альтернативой радиационному укреплению частей аппаратных средств. Мы оцениваем эффективность программных средств SIHFT, используемых с целью повышения надежности методов COTS. Приводится краткий обзор используемых SIHFT, предоставляющих проектировщикам и исследователям в этой области возможность выбора тех или иных решений.

1. Introduction

Radiation, such as alpha particles and cosmic rays, can cause transient faults in electronic systems. Such faults cause errors called Single-Event Upsets (SEUs). Transient faults occur once and then disappear. Often transient faults generate unpredictable random bit-errors or *soft errors* in an application system. A soft error is one that can be recovered by reprogramming. SEUs are a major cause of concern in a space environment and have also been observed at ground level. An example effect is a bit-flip, an undesired change of state in the content of a storage element. Radiation hardening is a well known technique used to reduce the sensitivity of the components to radiation.

Major drawback of the technique is that, the components are very expensive and lag behind today's commercial components in terms of performance. Therefore there is a strong motivation to build low-cost, high performance, fault tolerant systems that can perform in the space environment. Unhardened Commercial Off-The-Shelf (COTS) components, as in many other computing areas, are a challenging compotator against the specially designed components. The major drawback of COTS components is that they lack or have limited fault avoidance and fault tolerance features. Software-implemented hardware fault tolerance (SIHFT) techniques are proposed to provide low-cost solutions for enhancing the reliability of these systems without changing the hardware. First, we describe Error Detection Mechanisms which are a way for facing the consequences of hardware errors, in particular those originating from transient faults (Soft Errors) caused, for example by small particles hitting the circuit. These EDMs are extended to SIHFT techniques. Then SIHFT techniques will be considered.

2. Error Detection Mechanisms

The term *software fault tolerance* has been traditionally used for different purposes. We do not consider the issue of eliminating software bugs: we assume that the code is correct, and the faulty behavior is only due to transient faults affecting the system. It is possible to achieve a high degree of safe behavior in ordinary computers by complementing the intrinsic *Error Detection Mechanisms* (EDMs) of the system (exceptions, memory protection, etc.) with a set of carefully chosen software error detection techniques. These techniques include *Algorithm Based Fault Tolerance* (ABFT) [1], *Assertions* [2], and *Control Flow Checking, procedure duplication* [3] and *automatic transformations.*

2.1. ABFT. *Algorithm Based Fault Tolerance* is a very effective approach but lacks of generality. The

* Статья дана в авторском варианте.

technique focuses on matrix computations, and more specifically, on how we can detect and correct the probable faults in a matrix-vector or matrix-matrix multiplication. ABFT is distinguished by three characteristics: The encoding of the data used by the algorithm, the redesign of the algorithm to operate on the encoded data, and the distribution of the computation steps in the algorithm among computation units [2, 4].

2.2. Assertions. Assertions are a fairly common means of program validation and error detection. An executable assertion is a statement that checks whether a certain condition holds among various program variables, and, if that condition does not hold, takes some action. In essence, they check the current program state to determine if it is corrupt by testing for out-of-range variable values, the relationships between variables and inputs, and known corrupted states. These assertion conditions are derived from the specification, and the assertion can be made arbitrarily stringent in its checking. Assertions may be set up to only produce a warning upon detection of a corrupt state or they may take or initiate corrective action. For example, upon the detection of a corrupt state, the assertion may halt program execution or attempt to recover from the corrupt state [4, 13].

2.3. Control Flow Checking. The basic idea of *Control Flow checking* is to partition the application program in *basic blocks*, i.e., branch-free parts of code. For each block a deterministic signature is computed and faults can be detected by comparing the run-time signature with a pre-computed one. In most control-flow checking techniques one of the main problems is to tune the test granularity that should be used [4].

2.4. Procedure Duplication. Considering the *Procedure Duplication*, the programmer decides to duplicate the most critical procedures and to compare the obtained results. This approach requires that the programmer define a set of procedures to be duplicated and introduces the proper checks on the results. These code modifications can be executed only manually and may introduce errors [4].

2.5. Transformation Rules. This is an approach based on introducing data and code redundancy according to a set of transformations to be performed on the high-level source code. The transformed code is able to detect errors affecting both data and code: the former goal is achieved by duplicating each variable and adding consistency checks after every read operation. Other transformations focus on errors affecting the code, and correspond from one side to duplicating the code implementing each operation, and from the other to adding checks for verifying the consistency of the executed operations. This set of transformation rules applied to the high level code; these transformations introduce data and code redundancy, which allow the resulting program to detect possible errors affecting data and code. This method, although devised for transient faults, is also able to detect most permanent faults possibly existing in the system [4].

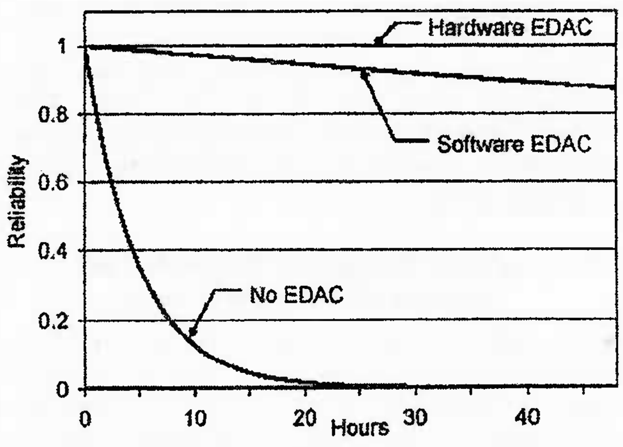
3. Software Implemented Hardware Fault Tolerance (SIHFT) Techniques

3.1. EDAC. Transient errors in memory chips are well-known, long considered reliability issues in computer systems. To prevent this problem *Error Detection and Correction* (EDAC) codes-also called *Error Correcting Codes* (ECCs) — are the dominating solution to this problem. The architecture requires extra hardware therefore introduces extra cost to the system. Usually COTS system does have no or very simplistic EDAC coding schemes. This limitation has to be solved by using another form of redundancy. Software EDAC techniques do not require extra hardware to implement EDAC but presents protection for code and data that resides in the main memory. SEUs in main memories usually manifest themselves as bit-flips. Software EDAC only addresses to find solution to transient errors, permanent errors are out of scope.

The objective is to devise a scheme to protect the data residing in main memory. The data that are protected by software EDAC are fetched and used by the processor in the same way as unprotected data are fetched and used. The EDAC program runs as a background task and is transparent to other programs running on the processor. Moreover, the protected data bits have to remain in their original form, to make the scheme transparent to the rest of the system. This requires the use of a systematic code such that data bits are not changed and are separable from the EDAC check bits.

If the same protection that is provided by hardware is to be provided by software, each read and write operation done by the processor has to be intercepted. However, this interception is infeasible because it imposes a large overhead in program execution time. Therefore, for software-implemented EDAC, only *periodic scrubbing* is done. In periodic scrubbing, the contents of memory are read periodically and all the correctable errors are corrected. When an error is detected, a scrub operation is enforced before the program is restarted.

The space used for check bits reduces the amount of memory available for programs and data. Therefore, the *check-bit overhead* (# check bits/ # data bits) must be as low as possible.



Software and hardware EDAC reliability comparison

It has proved to be very effective in enhancing the availability of the system. Without software EDAC, the system works an average of 2 days before it crashes due to SEU corruptions in programs and needs a reset. With software EDAC this average was increased to about 20 days which is still short but is an order of magnitude improvement. It can be seen from Figure that, hardware EDAC is outperforming software EDAC. This is due to the fact that hardware scheme performs the checking process on the fly for each data or code word read from the memory in oppose to the scrubbing process of software scheme [8, 9].

3.2. EDDI. *Error Detection by Duplicated Instructions* (EDDI) [10] is a software-only fault detection system that operates by duplicating program instructions and using this redundant execution to achieve fault tolerance. In EDDI, the instructions are duplicated during compilation and different registers and variables are used for the new instructions. This pure software technique is especially useful when designers cannot change the hardware, but they need dependability in the computer system.

The *Control Flow Checking by Software Signatures* (CFCSS) technique can be used with EDDI to increase the fault coverage. EDDI can be used to detect faults that are caused by bit-flips in memory, or that can be modeled as bit-flips in memory. For example in the operation of transfer from the memory to the data bus, a bit can be corrupted. This error can be modeled as a bit-flip in memory. Transient errors in control logic, address and date buses, functional units can cause the intermediate value of the computation to be incorrectly output. These errors can be detected by EDDI. The basic concept in time redundancy is to repeat computations and compare output results. If the errors are detected then the computations can be performed again to obtain the correct results. However time redundancy techniques suffer from execution time over head and performance loss in the system.

The idea of error detection by instruction duplication is to duplicate the instructions, variables and registers used in the control flow. *Master instructions* are the originals in the source code and the *shadow instructions* are the duplicated corresponding ones. General registers and memory are portioned into two segments to avoid overlapping store operations. After the master and shadow instructions are executed, the results are compared by the *comparison instruction.* In a correct execution the results are the same. If results are not the same, the computation is repeated to obtain the correct output. Below is an example of a simple duplicated addition operation:

```
ADD R3, R1, R2 ; master instruction
ADD R23, R21, R22 ; shadow instruction
BNE R3, R23, gotoError ; comparison instruction
```

To obtain the best performance out of the system, the comparison instruction should be performed right before register store operations or a branch or jump instruction. EDDI is a pure software technique for error detection and correction that achieves high error coverage with performance penalty due to the time redundancy introduced into the system. Algorithms to implement better performing EDDI and performance results can be found in [8, 10].

3.3. CFCSS. *Control Flow Checking by Software Signatures* (CFCSS) is a pure software method that checks the control flow of a program using assigned signatures. CFCSS uses an algorithm that assigns a unique signature to each node in the program graph and adds instructions for error detection. Signatures are embedded into the program during compilation time using the constant field of the instructions and compared with run-time signatures when the program is executed.

The basic idea of *Control Flow checking* is to partition the application program in basic blocks, i.e., branch-free parts of code. For each block a deterministic signature is computed and faults can be detected by comparing the run-time signature with a pre-computed one. There is also another algorithm used to reduce the code size and execution time overhead caused by checking instructions in CFCSS. Branching fault injection experiment is used with benchmark programs to determine the CFCSS performance. In the benchmark programs without CFCSS, an average of 33,7 % of the injected branching faults produced undetected incorrect outputs; however, in the programs with CFCSS, only 3,1 % of branching faults produced undetected incorrect outputs. CFCSS increases the error detection capability by an order of magnitude without the help of extra hardware added for error detection.

In CFCSS the program is divided into basic blocks. All nodes in the program graph are assigned different arbitrary numbers (signatures), which are embedded into the program during preprocessing or compile time. During program execution, a run-time signature G is stored in one of the general purpose registers called the *Global Signature Register* (GSR), and compared with the stored signature of the node whenever control is transferred to a new node. For multiple branching cases, a run-time adjusting signature D is combined with G [8, 11]. The complete algorithm and an example program are presented in [5]. Comparison of CFCSS with other software signature techniques can be found in [5].

3.4. SIED. *Software Implemented Error Detection* is a new error detection technique which is based on a new *control check flow* scheme combined with software redundancy. The distinctive advantage of the SIED approach over other fault tolerance techniques is the *fault coverage*. SIED is able to cope with faults affecting data and the program control flow. This technique is a new error detection approach combining software redundancy by duplicating instructions and the data segment, and a signature monitoring technique, which checks the inter-block and intra-block control flow. This method is able to cover both faults affecting the program execution flow and faults corrupting the program workspace.

The novelty of the proposed approach is the signature monitoring technique controlling the program flow that is combined with data and instruction duplication. The intra-block control flow is checked by introducing a *checkpass* flag between the original operation and the replicated one, while the inter-block control flow is checked by using dedicated control variables, which contain information about the current state of the program execution allowing to predict the next set of instructions that will be executed by the processor [6]. Since SIED presents high error detection abilities, it is suitable for safety-critical applications designed for mobile computing systems.

Another category of purely software approaches is *signature-monitoring techniques* in which a unique signature associated with a basic block is precompiled and saved somewhere in memory. During program execution, the same signature is computed and compared with the reference one. To implement the proposed technique, the program is split in basic blocks. A basic block is a finite number of ordered instructions to be executed sequentially. There is no branching instruction into a basic block, except possibly the last one [6].

Intra-Block Detection. The intra-block detection technique was designed to be combined with the instruction duplication approach introduced in [7]. The instruction duplication approach is designed to detect resulting errors in data, but this approach is inefficient when faults modify the content of a variable and its replica. The main idea of the proposed intra-block detection technique is to ensure that at least one of two instructions in a pair of related instructions (the original and its replica) is correctly executed. Each basic block has a fixed number of instructions. Intra-block detection is ensured, by introducing a *check pass* flag between the original instruction and its replica.

Inter-Block Detection. In order to detect on-line faults affecting inter-block transfer control, we developed an algorithm that assigns a unique signature to each basic block, and adds extra instructions in order to check the accuracy of the inter-block transfer control flow. The control flow is checked knowing in advance the next set of instructions that compose the basic block to be executed. Inter-block transfer control is checked by dedicated control variables containing information about the program execution current state. These variables are:

- *Identifier Block (IDB) — A unique identifier associated to each basic block.*
- *Status Condition Branch (SCB) — SCB is a variable updated each time a conditional instruction is executed.*
- *Execution Order (EO) — it decides the branching order when a block transfers the control to several different blocks*

The signature function used to detect illegal branches is:

$$X = f(EO_{ij}, SCB_{ij}, IDB_i) = EO + SCB + IDB_i.$$

There are different types of inter-block transition. They can be classified in three groups according to their dependency either on the current state of the program or on a certain condition to be satisfied (the case of conditional branching instructions) [6].

4. Conclusion

Table shows the total number of errors in each test program and the detection mechanism that detected these errors. Most of the errors were detected by EDDI. However, there are errors detected by CFCSS and watchdog timer, especially for tests with larger code sizes. Overall, the combination of EDDI, CFCSS and Watchdog Timer detected a total of 321 errors.

Errors detected by each SIHFT technique and undetected errors in the computation tests on the COTS board [8]

Program	Number of Errors	Error Detected by Each Technique			Undetected Errors
		EDDI	CFCSS	Watchdog	
Integer Sort	156	156	—	—	—
FP Sort	21	21	—	—	—
Quick Sort-Integer	43	31	5	6	1
FFT	102	99	1	2	—
Total	322	307	6	8	1

There is one case of undetected error. These numbers yield 99,7 % error detection coverage.

The results from the Hard board show that despite all the hardware fault tolerance techniques used in the board; there are cases of undetected errors. Even if single points of failure are eliminated by better design, additional fault tolerance techniques, perhaps in software, may still be required for high reliability. Software-implemented error detection and recovery techniques have been effective for the error rate observed in the COTS board. Software EDAC improves the availability of the COTS board by an order of magnitude and had only 3 % performance overhead. Even though hardware EDAC would be preferable for main memory, software EDAC provided acceptable reliability for the experiment. The experimental results show 99,7 % error detection coverage and 98.8 % error recovery coverage. Results show that COTS with SIHFT are viable techniques for low-radiation environments. In [12] optimizations applied to EDDI+ EDAC+ CFCSS will be described. These optimization comprise Software Implemented Fault Tolerance (SWIFT).

In this paper a new error detection technique called Software Implemented Error Detection (SIED) presented. The proposed method is based on a new control check flow scheme combined with software redundancy. The distinctive advantage of the SIED approach over other fault tolerance techniques is the *fault coverage*. SIED is able to cope with faults affecting data and the program control flow. Since SIED presents high error detection abilities, it is suitable for safety-critical applications designed for mobile computing systems.

REFERENCES

1. **Huang K. H., Abraham J. A.** Algorithm-Based Fault Tolerance for Matrix Operations // IEEE Trans. Computers. Vol. 33. Dec. 1984. P. 518—528.
2. **Zenha Rela M., Madeira H., Silva J. G.** Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks, Proc. FTCS-26, 1996. P. 394—403.
3. **Pradhan D. K.** Fault-Tolerant Computer System Design. Prentice Hall PTR, 1996.
4. **Stefanidis V. K. and Margaritis K. G.** Algorithm Based Fault Tolerance: Review and experimental study. Parallel and Distributed Processing Laboratory. Departement of Applied Informatics. 2003. WILEY-VCH Verlag GmbH & Co. KGaA. Weinheim.
5. **Nahmsuk Oh, Philip P. Shirvani and Edward J. McCluskey.** Control-Flow Checking by Software Signatures // IEEE Transaction on reliability. Vol. 51. N 2. March 2002.
6. **Nicolescu B., Savaria Y., Velazco R.** SIED: Software Implemented Error Detections // Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03) 1063-6722/03. 2003. IEEE.
7. **Nicolescu B., Velazco R.** Detecting soft errors by a purely software approach: method, tools and experimental results. Design Automation and Testing in Europe (DATE 2003). Munich: Germany, March 3—7. 2003.
8. **Ugur YENIER.** Fault Tolerant Computing In Space Environment And Software Implemented Hardware Fault Tolerance Techniques. Department of Computer Engineering Bosphorus University. Istanbul, 2000.
9. **Shirvani P. P., Oh N., McCluskey E. J., Wood D. L. and Lovellette M. N., Wood K. S.** Software-Implemented Hardware Fault Tolerance Experiments COTS in Space. 2000.
10. **Oh N., Shirvani P. P., and McCluskey E. J.** Error detection by duplicated instructions in super-scalar processors // IEEE Transactions on Reliability. 51 (1): 63—75. March 2002.
11. **Rebaudengo M., Sonza Reorda M., Torchiano M.** Massimo VIOLANTE "soft-error Detection through Software Fault-Tolerance Techniques".
12. **George A.** Reis Jonathan Chang Neil Vachharajani Ram Rangan David I. August. "SWIFT: Software Implemented Fault Tolerance", Departments of Electrical Engineering and Computer Science.
13. **Laura L.** Pullum, "Software Fault Tolerance Techniques and Implementation", 2001 Artech House British Library Cataloguing in Publication Data, Boston, London.

ООО "Наука и технологии"

Учредитель журнала ООО "Наука и технологии"

Журнал зарегистрирован в Комитете Российской Федерации по печати.

Свидетельство о регистрации № 018873 от 27 мая 1999 г.

Редактор *Абизова Т. В.*

Оригинал-макет и электронная версия изготовлены в ООО "Адвансед Солюшнз".

Сдано в набор 28.01.2010. Подписано в печать 11.03.2010.

Формат 60 × 88 1/8. Усл.-печ. л. 5,88. Уч.-изд. л. 6,43. Печать цифровая. Тираж 120 экз.

Отпечатано в ООО "СиД"