A Software-Based Error Detection Technique Using Encoded Signatures

Yasser Sedaghat, Seyed Ghassem Miremadi, Mahdi Fazeli
Dependable Systems Laboratory (DSL)
Sharif University of Technology, Tehran, Iran
y_sedaghat@ce.sharif.edu, miremadi@sharif.edu, m_fazeli@ce.sharif.edu

Abstract

In this Paper, a software-based control flow checking technique called SWTES (Software-based error detection Technique using Encoded Signatures) is presented and evaluated. This technique is processor independent and can be applied to any kind of processors and microcontrollers. To implement this technique, the program is partitioned to a set of blocks and the encoded signatures are assigned during the compile time. In the run-time, the signatures are compared with the expected ones by a monitoring routine. The proposed technique is experimentally evaluated on an ATMEL MCS51 microcontroller using Software Implemented Fault Injection (SWIFI). The results show that this technique detects about 90% of the injected errors. The memory overhead is about 135% on average, and the performance overhead varies between 11% and 191% depending on the workload used.

1. Introduction

Today, the use of general purpose processors (from modern processors to simple microcontrollers) has become a common trend in many safety-critical systems such as space, automotive and industrial applications. Since these systems usually operate in harsh environments and the presence of faults in such applications may cause catastrophic consequences, the faults must be detected as early as possible [8]. It has been reported that the occurrence of transient faults is more probable (10 to 30 times [18]) than the permanent and intermittent faults in such systems [11, 16]. The permanent or transient faults in hardware components such as the program counter, the address circuitry and the memory elements or the software bugs such as compiler and operating system bugs also may result in CFEs [9, 15]. [16, 20] reported that more than half (up to 70% [2]) of the transient faults lead to the control flow errors (CFEs) in the program execution. Therefore, it seems that control flow checking is a viable solution to the systems reliability requirements.

Control flow checking techniques are mainly based on signature monitoring principle. An abstract of program execution is extracted and some signatures that represent the chosen abstract are assigned for the correct program execution before the system's execution (run-time). Signatures are assigned arbitrarily (assigned signatures), e.g. SIC [9], CFCSS [15], CCA [12], ECCA [1], and SEIS [11, 16], or derived from the binary code or the address of the instructions (derived signatures), e.g. PSA [14], SIS [19], ASIS [3], CSM [21], OSLC [10], and TTA [13]. During the run-time, the signatures are again generated in real-time and are compared to the stored, expected signatures. If a disagreement occurs, the occurrence of an

Proceedings of the 21st IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'06)
0-7695-2706-X/06 $20.00 © 2006 IEEE

error is detected and reported. The signature monitoring is performed by a watchdog processor in hardware and a monitoring routine [4], in software techniques. Since in modern computer architectures, the observability of the system bus is drastically reduced, e.g. by the use of on-chip caches and instruction prefetch queues, derived signature based techniques no longer can be used. Therefore, the employment of assigned signatures based techniques are almost exclusively used in COTS processors.

Many control flow checking techniques which are based on assigned signature principle are proposed in the literature. These techniques have some disadvantages which are as follows:

- The memory and performance overhead of these techniques are relatively high.
- The algorithm of assigning the signatures is complex in some techniques (e.g., SEIS [11, 16] and ECCA [1]).
- The Software-Based techniques, e.g. ECCA [1], are unable to detect the program crashes which are caused by a CFE.
- The error detection coverage of these kinds of techniques is rather low.

In this paper, a software based technique, called SWTES is presented and evaluated by means of the SWIFI [17] method. In the SWTES, the signatures are assigned to the program by a comparatively simple algorithm and compared to the expected ones in the run-time. This technique tries to mitigate the mentioned existing problems for the assigned signature-based techniques and provides high error detection coverage as well as having acceptable overheads.

The structure of this paper is as follows: Following the introduction, the error models are presented in section 2. The SWTES technique will be presented in details in the third section. In section 4, the capabilities of the proposed technique are explained. The experimental results are introduced in section 5 and finally the paper is concluded in the last section.

2. CFE models

The SWTES technique primarily detects CFEs caused by transient faults. The following definitions are presented for the sake of clarity.

Definition 1: A CFE is an illegal branch which can be caused by transient faults in hardware such as the program counter, address circuit or memory system [9].

Definition 2: A Basic Block (BB) is a maximal set of ordered non-branching instructions (except in the last instruction) or branch destinations (except in the first instruction) in which the execution always enters at the first instruction and leaves via the last instruction [22]. These sets should have a minimal length.

Definition 3: A Partition Block (PB) is a set of instructions between two physically consequent BB.

Definition 4: A program crash occurs when the execution illegally continues outside the program (unused memory space)

The SWTES technique assumes that the program is partitioned into BBs and PBs. In this model, seven types of CFEs are considered which are shown in Figure 1.

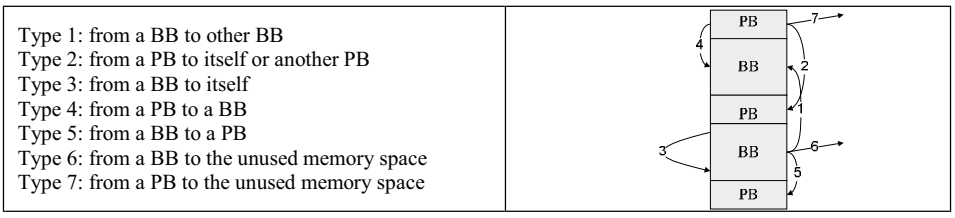


Figure 1. Program partitions and the seven types of CFEs

In fact, a CFE is the occurrence of one of the seven types of the CFEs. Types 6 and 7 usually lead to a program crash in our experiments. Therefore, these two types can be merged.

3. The SWTES technique

The structure of a program can be represented by a directed graph, where nodes represent the Basic Blocks and the arcs represent the relations between the Basic Blocks. This directed graph is called the *control flow graph* (CFG). In Figure 2 (b), the CFG of a typical program, in figure 2 (a), has been shown.

A	L1:
	MOV DPTR,#ARRAY
	MOV R1,#30
	MOV R4,#00

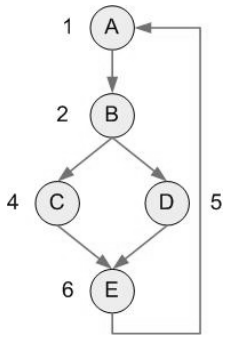
B	CLR A
	MOVC A,@A+DPTR
	INC DPTR
	MOV R0,A
	CLR C
	SUBB A,R1
	JC L2

C	MOV A,R0
	ADD A,R3
	MOV R3,A
	SJMP L3

D	L2:
	MOV A,R0
	CLR C
	SUBB A,R3
	MOV R3,A

E	L3:
	MOV R0,R4
	MOV R3,A
	MOV @R0,R3
	SJMP L1

(a)



(b)

Figure2. A typical source code and its CFG

The SWTES technique has the following steps:

1. Extracting the Basic and Partition Blocks and generating CFG of the program.
2. Labeling the Graph nodes by a specific algorithm.
3. Producing the signatures for each labeled node and assigning them to the Basic Blocks.
4. Inserting the appropriate instructions to the end of each Basic Block in order to sending the signature to the monitoring system in run-time phase.
5. In the SWTES, a supplementary mechanism called, Entry-Exit Checking is exploited to increase the error detection coverage. So in this step, a flag is assigned for the Basic Blocks and one for the parts of Partition Blocks which have not any branch instructions and branch destinations inside except in the last or the first instruction. Actually these parts are the Basic Blocks which have a short length, but it is not efficient to consider them as the Basic Blocks due to the large amount of imposed overhead.
6. In fact, there are two major parts that should be explained more for clarifying the proposed technique: The *labeling algorithm* and the *signature generating step*.

3.1. The labeling algorithm

In the labeling algorithm, the program CFG is taken as the input and a unique label is assigned to each node. It should be noticed that in each CFG, the length of labels of the nodes is the same. This labeling is exploited in the signature generating step which will be discussed in the following section. The labeling algorithm is presented below:

In this algorithm, a unique label is assigned to each of the CFG nodes and two supportive lists, namely *Black list* and *Valid list*, which are empty at first, are used. The algorithm works as follows: at first, the label 1 is assigned to the first node which is the root of the CFG. Then, for labeling each of the unlabeled nodes, one of the following cases happens:

1. If the node with *X* label (*X* node) has two unlabeled successor nodes then labels *2X* and *2X+1* are assigned to these nodes.
2. If *X* node has only one unlabeled successor node then label *2X* is assigned to this node and label *2X+1* is added to the Black list.
3. If *X* node hasn't any successor nodes or all of its successors be labeled before, labels *2X* and *2X+1* are added to the Black list.

In this algorithm, a numeric variable, *Limit*, is also used for limiting the growth of length of labels. Its value is initially as follows:

$$Limit = \lfloor \log_2(total\,number of\ nodes) \rfloor$$

If in any of the previously described steps of the algorithm, the label that is to be assigned to a node be greater than 2^{Limit}, one of the following cases is done:

1. If the *Valid list* is not empty and its first element is smaller than 2^{Limit} then this first element is removed from the list and is used as the label for the current node.
2. If the Black list is not empty then its first element (e.g. *A*) is removed from this list and labels *2A* and *2A+1* are inserted in the valid list then:
   a. If *2A* is smaller than 2^{Limit} this element is removed from valid list and is assigned to the current label.
   b. If *2A* is not smaller than 2^{Limit} then Num is increased by one unit and label *2X* is assigned to the current node.
3. In case none of the above cases is true *Limit* is increased by one unit and label *2X* is assigned to the current node.

The result of the above cases is that whenever it is not possible to assign a label to the successors of a node, this label will not be assigned to any other nodes of the CFG. Therefore by using this algorithm, all of the graph nodes are uniquely labeled, that is, the label which is assigned to each node is unique in the entire CFG.

Since the application programs are implemented in assembly level, the nodes have at most two successors. But it should be noted that this technique can be slightly modified to support more than two successors. Figure 2 (b), also shows the assigned labels in a typical CFG according to the preceding labeling algorithm.

3.2. The signature generating step

The generated signature for each node has four fields that three of them have a constant length and the fourth field may not exist regarding the node labeling relations (see figure 3). The length of the most significant field is two bits, called *Status field*. These two bits specify the status of the label of this node which is related to its successor node labels. The next field is the *Entry/Exit bits*. These bits indicate the last status of entry and exit flags. The third field (ID) specifies the label of the current block (the block which has already sent the signature). Finally, based on the last variable field of signature, named *successors field* and the status field, possible successors of current block are uniquely determined.

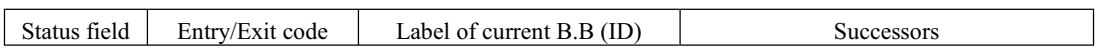


Figure 3. The structure of a Signature

The status field of the signature is determined based on the relations between the label of this node and labels of its successors. The rule of extracting the status field is listed bellow:

1. If the labels of successors of a block (with X label) are 2X and 2X+1 (or only 2X, if there is only one successor), status field of assigned signature of the block is equal to '00'. These kinds of successors are called *systematic successors* (see the figure 4).

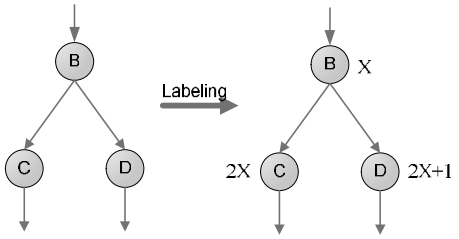


Figure 4. The systematic successors

2. Since in the CFG program it is probable for a node to have more than one predecessors, it can not be guaranteed that its label is in the form of 2x or 2x+1 (x is the label of the node's predecessor). If at least one of the node successors have a label which is not in the form of 2x or 2x+1, the status field of its signature is equal to '01'. The successor nodes that their labels are not in the form of 2x or 2x+1 are classified as *unsystematic successors* from their predecessors point of view. For example, in figure 5(a), the node C has two successors, named B and D. Since the node B has another predecessor except node C and due to the graph topology and labeling algorithm, node B is labeled by using this predecessor label and not the one of node C. In this case, node B is named an unsystematic successor. In contrast, node D has only one predecessor (node C) and is labeled after its predecessor, i.e. it is in a normal situation and is called a systematic successor. The status field of node C signature is '01'. In figure 5(b), since label of node C is X and the labels of its successor nodes are in the form of 2X and 2X+1, the status field of its signature is '00'. Label of node D is 2X but both of its successors are labeled by their previous predecessors and are unsystematic successors. Therefore, the status field of node D signature is '01'.

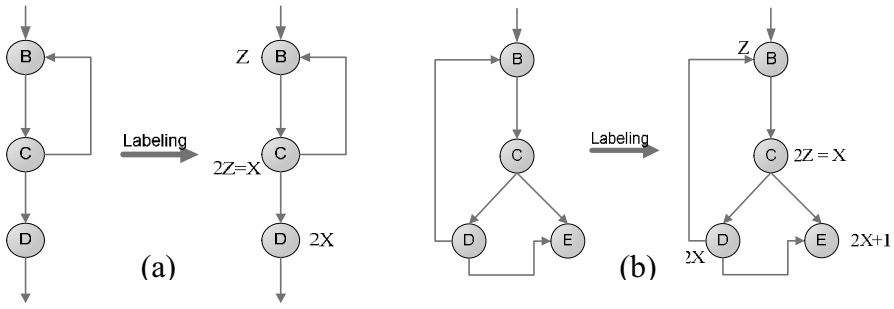


Figure 5. The unsystematic successors

3. Finally, the value '11' is reserved in the status field for sending a specific or control message to the monitoring system, e.g. sending an initial control message that informs the length of labels to the monitoring system.

The length of the second field is two bits, and is determined at the time of signature transmitting to the monitoring system. At the run-time, initially two flags (F1 and F2) are set to zero. In the entrance and exit parts of each Basic Block the F1 flag is complemented. Since the size of the sending signature part is relatively long, a flag is assigned to the first and to the end of this part to detect the possible illegal CFEs inside it. It should be noted that only one byte instruction is imposed for setting each flag which is a little overhead compared to the whole

block size. Whenever the execution enters to the sending signature part or exits from it, this flag is complemented. In the monitoring system when a signature is received and compared with the expected ones, the content of the flags is checked to verify if each exit follows an entry (see the Figure 6).

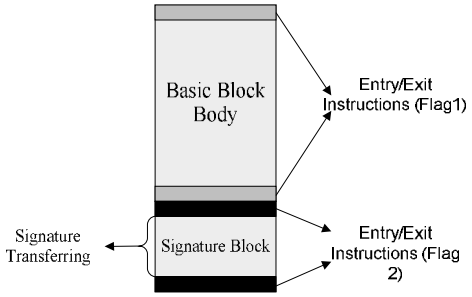


Figure 6. The location of inserting of Entry-Exit instructions

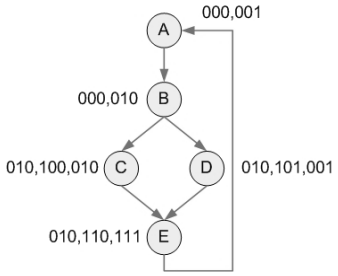


Figure 7. The CFG of the figure 2 with assigned signatures

The content of the fourth field that specifies the successors of the current block has a variable length depending on the status field. This field is used in the form of the following stages:

1. If the status field is equal to '00', the successor field will be removed. In this situation, the length of this field would be zero.
2. If the status field is equal to '01' the successor field will contain the XOR-difference between the current block label and labels of unsystematic successors. In this situation, the length of this field would be at most twice the length of labels.
3. And finally, if the status field is equal to '11', this field will be removed.

Figure 7 shows signatures that are assigned to CFG nodes of figure 2.

3.3. The control flow checking scenario

Whenever the program execution reaches the signature sending part, the signatures of each block is sent to a monitoring system. The monitoring system specifies the successors of the current Basic Block based on its status bits, ID fields and the possible successor field. Then in the monitoring system, the current signature ID is just compared with the label of successors of previous block which are specified by the previous signature. If the ID of the current block be equal to one of the labels of successors of the previous block, the flow of the program is correct and no CFE has been occurred.

In the proposed technique, an on-chip processor timer is exploited as a workload timer for detecting the program crash errors. In this mechanism, the timer is restarted by the monitored program in a special part of the program for preventing the overflow of the workload timer. In case of a program crash, the program flow will never reach a restarting instruction and an overflow interrupt will occur. In this case, a signature with status field of '11' is sent to the monitoring system.

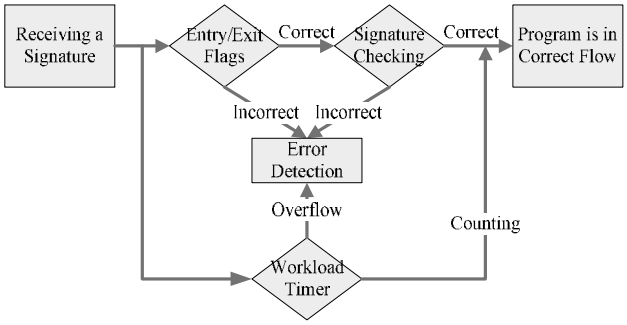


Figure 8. The flow of CFE detection in SWTES technique

As mentioned before, two flags are used to implement the Entry-Exit Mechanism. In the monitoring system, since these two flags are checked in the middle of the signature block the content of the flags should be always '01' in case of correct execution. Otherwise, a CFE has been occurred and is detected. Figure 8, shows the flow of CFE detection in the SWTES technique.

4. Capabilities of the SWTES technique

Regarding figure 1, it is obvious that type 1 and 2 errors are detected by monitoring system, while the next signature is received and checked. Also, type 4 and 5 errors disturb the program execution. Therefore, the flags are incorrectly complemented and a signature with invalid Entry/Exit field is sent to the monitoring system. Similar to other techniques which are based on the assigned signatures, the third type of errors can not be detected by this technique, unless in special situations (for example, if the result of these CFEs be jumping to the operand part of an instruction, this may be interpreted as an opcode of a branch instruction and as mentioned above can be detected). Finally, the incorrect jumps to unused memory spaces are detected by using the system internal workload timer.

Another feature of SWTES is its ability to checking the program control flow in multi processor systems. To reach this purpose, each processor just attaches its identification to the signatures and then sends it to the monitoring system. In this approach, the monitoring system can distinguish program control flows of different monitored systems from each others.

Due to the short signature length, the SWTES has a relatively low communication overhead in comparison with other similar techniques such as SEIS [11, 16] and ECCA [1]. In the proposed technique, unlike some approaches like ECCA [1], a simple labeling mechanism has been used that noticeably results in preventing long label lengths.

5. Experimental results

As mentioned before, since the aim of this research work is to introduce a general control flow checking technique, i.e. it dose not depend on a specific processor or platform, the proposed technique is applied to the 8051 microcontroller family which is widely used as the core of modern microcontrollers. In order to evaluate the SWTES technique, a tool named uVision 3.0 [7] is used to compile the assembly code of the benchmarks which are protected by the SWTES technique. The mentioned tool has also the capability of simulating the 8051 microcontroller family which is exploited for implementing the whole system that consists of the protected code, monitoring and fault injection system. The protected code saves the signatures to a specific location of system memory and then the monitor program picks up them from the mentioned location.

Table 1. The SWTES's control flow error detection coverage and memory overhead

Benchmarks	Number of injected faults	Number of detected errors	Error detection Coverage	Memory Overhead	Performance Overhead
Bubble Sort	2000	1964	98.2%	174.8%	191.36%
Linked List	2000	1625	81.25%	90.9%	10.95%
Binary Search Tree	2000	1803	90.15%	139.84%	98.86%

Three workload programs, all written in MCS51 assembly language, were used in the experiment: a bubble sort, a linked list and a binary search tree. A total of 2000 faults were injected into the evaluation system while running each workload and the results are shown in table 1. The used fault injection method in this work is SWIFI, in which a bit of the program counter register is randomly flipped at random times.

For comparison, the overhead and coverage figures of some of the previous control flow checking techniques are reported in table 2. It is evident that the SWTES technique, which is a general purpose technique, has the high error detection coverage in comparison with the other general purpose techniques.

Table 2. Comparison of the SWTES technique with some of the previous, software-based CFC techniques

CFC Methods	Memory Overhead (%)	Performance Overhead (%)	Error Detection Coverage (%)	General Purpose
ECIC [17]	5~10	42~67	90.5~98.2	No
CFCSS [15]	26.6~63.6	16.2~69.2	28.8~41	Yes
ECCA [1]	303~490	260~622*	22.6~83.4*	Yes
ACFC [20]	48~112.2	41~136.2	10~92.1	Yes
YACCA [6]	191~496	110~354	21.1~56	Yes
CFCBTE [5]	33~44	110~304	89.4~94.3	No
SWTES	90.9~174.8	10.95~191.36	81.25~98.2	Yes

* Previously reported in ACFC [20]

6. Conclusions

Two broad categories for designing reliable systems using COTS processors have been proposed in the literature: structure-based techniques and behavior-based techniques. The structure-based techniques, which are based on hardware redundancy, are considered to be expensive. On the other hand, behavior-based techniques usually provide adequate levels of error detection and hence system reliability, with acceptable overheads. Among the behavior-based techniques, Control flow checking techniques provide a viable solution to the modern embedded Systems reliability requirements.

In this article, a software-based control flow checking technique which does not depend on a specific processor or platform is proposed and the SWIFI approach is used to evaluate the technique. The results show that this technique detects about 90% of the injected control flow errors. The memory overhead is 135.18% on average, and the performance overhead varies between 10.95% and 191.36% depending on the workload used.

7. References

[1] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy and J.A. Abraham, "Design and evaluation of system-level checks for on-line control flow error detection," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 10, Issue 6, June 1999, pp. 627 – 641.

[2] E.W. Czech and D. Siewiorek, "Effects of transient gate-level faults on program behavior," *Proc. of 20th Internat-ional Symposium on Fault-Tolerant Computing (FTCS-20*), Newcastle-Upon- Tyne, UK, June 1990, pp. 236-243.

[3] J.B. Eifert and J.P. Shen, "Processor Monitoring Using Asynchronous Signatured Instruction Streams," *Proc. of 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, Los Angeles, CA, USA, June 1995, pp. 106-111.

[4] A. Ersoz, D. M. Andrews, and E. J. McCluskey, "The watchdog task: Concurrent error detection using assertions," Center for Reliable Computing, Stanford Univ., CA, CRC-TR 85-8, 1985.

[5] M. Fazeli, R. Farivar and S.G. Miremadi, "A Software-Based Concurrent Error Detection Technique for PowerPC Processor-based Embedded Systems," Proc. of *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'05)*, Monterey, CA, USA, Oct. 2005, pp. 266-274.

[6] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda and M. Violante, "Soft-Error Detection Using Control Flow Assertions", Proc. of *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03)*, Boston, Massachusetts November 2003, pp. 57-62.

[7] Keil - An ARM Company, µVision IDE tool, http://www.keil.com/uvision2, 2006.

[8] J. H. Lala and R. E. Harper, "Architecture Principles for Safety-Critical Real-Time Applications", *Proc. of the IEEE,* vol. 82, January 1994, pp. 25-50.

[9] D.J. Lu, "Watchdog processors and structural integrity checking," *IEEE Trans. on Computers*, Vol. C-31, Issue 7, July 1982, pp. 681-685.

[10] H. Madeira and J. G. Silvia, "On-line signature learning and checking", in *Dependable Computing for Critical Applications 2*, J. F. Schlichting and R. D. Schlichting, Eds: Springer-Verlag, 1992, pp. 395–420.

[11] I. Majzik and A. Pataricza, "Control flow checking in multitasking systems," *Periodica Polytechnica-Series Electrical Engineering*, Vol. 39, Issue 1, Technical University of Budapest, 1995, pp. 27-36.

[12] L.D. Mcfearin and V.S.S. Nair, "Control-flow checking using assertions," *Proc. of IFIP International Working Conference Dependable Computing for Critical Applications (DCCA-05)*, Urbana-Champaign, IL, USA, September 1995.

[13] G. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson, "Use of Time, Location and Instruction Signatures for Control Flow Checking", *Proc. of the DCCA-6 International Conference*, *IEEE Computer Society Press*, Urbana-Champaign, IL, USA, September 1995, pp. 201-221.

[14] M. Namjoo, "Techniques for concurrent testing of VLSI processor operation," *Proc. of International Test Conference (ITC'82)*, Philadelphia, PA, USA, November 1982, pp. 461-468.

[15] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. on Reliability*, Vol. 51, Issue 1, March 2002, pp. 111-122.

[16] A. Pataricza, I. Majzik, W. Hohl and J. Hoenig, "Watchdog processors in parallel systems," *Proc. of 19th Sym-posium on Microprocessing and Microprogramming (EUROMICRO'93)*, Barcelona, Spain, 1993, pp. 69-74.

[17] A. Rajabzadeh and G. Miremadi, "Transient Detection in COTS Processors Using Software Approach", Proc. of *10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2004)*,Papeete, Tahiti, March 2004, pp.49-54.

[18] M.A. Schuette and J.P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Trans. on Computers*, Vol. C-36, Issue 3, March 1987, pp. 264-277.

[19] J.P. Shen and M.A. Schuette, "On-line self-monitoring using signatured instruction streams," *Proc. of International Test Conference (ITC'83)*, Philadelphia, PA, USA, Oct. 1983, pp. 275-282.

[20] R. Venkatasubramanian, J.P. Hayes and B.T. Murray, "Low-cost on-line fault detection using control flow assertions," *Proc. of 9th IEEE On-Line Testing Symposium (IOLTS'03)*, Greece, July 2003, pp. 137 - 143.

[21] K. Wilken and J.P. Shen, "Continuous signature monitoring: low-cost concurrent detection of processor control errors," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 9, Issue 6, June 1990, pp. 629-641.

[22] S.S. Yau and F.C. Chen, "An approach to concurrent control flow checking," *IEEE Trans. on Software Engineering*, Vol. 6, Issue 2, March 1980, pp. 126-137.