# A Dependable Microcontroller-based Embedded System

Amir Rajabzadeh
Department of Compute Engineering
Razi University
Kermanshah, Iran
rajabzadeh@razi.ac.ir

Mahdi Vosoughifar
Department of Computer Engineering
Islamic Azad University Arak Branch
Arak, Iran
mehdi_vosoughifar@yahoo.com

*Abstract*—**This paper presents a method to make a dependable microcontroller-based system for detecting any violation from the program flow caused by transient faults. The method is based on a duplication and comparison technique and employs a "synchronous interrupt" in both microcontrollers to monitor and compare the program counters (PCs) of the microcontrollers. This is done by adding an interrupt service routine in both microcontrollers and without any modification of the application programs. The method has been experimentally evaluated using AVR ATMega-32 microcontrollers. The results show that error detection coverage of the method is 100% based on the fault models. The error detection latency varies about 1184 cycles (74 μsec) to 128147 cycles (8 msec) and the execution time overhead of the method varies between 0.5% and 50% for different PC exchange interrupt frequencies. The hardware and software overheads are about 100% and less than 0.5% respectively.**

***Keywords- dependable system; control flow checking method; concurrent error detection; microcontroller-based system; embedded system.***

## I. INTRODUCTION

We are used to hearing about extended computer applications and explosive growth in the computation ability of processors. Based on usage patterns, processor cores can be divided into four categories [1]:

1) Computational micros: they are 32-bit or 64-bit general-purpose processors, and typically deployed as the central processing unit of mainframes, workstations, and personal computers. Most commercial off-the-shelf RISC and CISC processors fall into this category. This group has accounted for less than 2% of the volume of processors shipped.

2) Embedded general-purpose micros: they are general-purpose processors, usually 32-bit processors, designed for embedded systems. These are often scaled-down versions of existing computational micros. Embedded general-purpose micros constituted about 8% of total volumes of processors shipped.

3) Digital signal processors: they are specific-purpose processors with the ability to execute arithmetic operations efficiently. This group accounted for about 10% of the volume of processors shipped.

4) Microcontrollers: they have 8-bit, 16 bit or 32-bit processor core with memory, I/O, and peripherals on a chip. Microcontrollers have been estimated to be about 80% of the processors shipped.

Embedded systems are widely used in industrial control systems [2]. Industrial control systems usually have fairly low computational requirements and low memory capacity. This is within the domain of 8-bit and 16-bit microcontrollers. Small 8-bit CPUs still dominate the market, representing about 70% of overall processor shipments [1].

These embedded systems are usually involved with some aspects of dependability issues and system failures can severely damage human life or equipments. In these systems, dependability is an important concern and error detection mechanism has a key role in designing the system. On the other hand, as the number of transistors per chip continues to grow, the error rate per chip is expected to increase [3], the fault occurrence rates are increasing by approximately 8% per chip [4]. These trends show that to ensure correct operation of embedded systems, they must employ dependability methods against transient faults.

This paper actually presents a concurrent error detection method for embedded systems based on microcontrollers. The proposed method employs the synchronous external burst interrupt in duplication microcontrollers and compares the run time program counters of the microcontrollers in a service routine. The method has been experimentally evaluated on an AVR microcontroller-based system. The results show that error detection coverage of the method is 100% based on the fault models. The error detection latency varies about 1184 cycles (74 μsec) to 128147 cycles (8 msec) and execution time overhead of the method varies between 0.5% and 50% for different PC exchange interrupt frequencies. The hardware and software overheads are about 100% and less than 0.5% respectively.

The next section depicts the related work. Section 3 discusses the error models in this experiment. Section 4 describes the proposed method. Section 5 gives method evaluation and argues over a system under test. The results are presented in section 6, and finally, section 7 summarizes and concludes the paper discussion.

## II. RELATED WORK

This section describes how embedded systems based on microprocessor or microcontroller have been equipped to detect transient faults.

To design a dependable embedded system at least two options are available:

- Using Application-Specific-IC (ASIC) processors: such as ERC32 processor [5], LEON-FT processor [6] and THOR processor [7] with internal error detection mechanisms.
- Using Commercial Off-The-Shelf (COTS) processors: such as Intel Pentium family, PowerPC and ARM processors, or AVR and PIC microcontrollers.

Designing an embedded system with fault tolerant ASIC processor is a useful way of making a dependable system. Fault tolerant ASIC processors have many facilities for tolerating faults and have a high percentage of error detection coverage.

Concurrent error detection or fault masking mechanisms in ASIC processors are often applied at VLSI, transistor, gate or RTL levels. Chip-level and behavioral-based mechanisms may be used as well. ERC32 is a 32-bit processor [5] and it is compatible with SPARC V7 ISA. The processor has been designed for embedded space flight applications. The hardening techniques in the VLSI level (layout hardening) have been applied to reach the radiation tolerance. All registers in integer and floating unit have been provided with parity bits (gate level). Program flow control has been implemented using embedded signature monitoring (behavioral-based mechanism) and master/checker mechanism at chip-level is supported by the processor. LEON-FT is a 32-bit processor [6] and it is compatible with SPARC V8 ISA. Internal cache memory and register file in the processor has been provided with error-detection in form of parity bits. Flip-flops are implemented using triple modular redundancy (TMR) and master/checker mechanism at chip-level is supported by the processor as well.

Although fault tolerant ASIC processors present a good way of designing a dependable system, nevertheless, the use of commercial off-the-shelf (COTS) processors are phenomenally popular, because it decreases the cost significantly.

COTS processors have a low or moderate percentage of error detection coverage, but short time-to-market [8], availability in the market [8], trust in products [9], low development, test equipment and maintainability cost [10] of the systems are important matters to design a low-cost dependable system. Meanwhile, engineers can make use of a wide range of facilities in available market [8], [9].

Since COTS microcontrollers have not been designed for fault tolerant applications [9], [11], they require additional methods to enhance error detection capability in these systems [9]. The use of COTS processor incurs additional error detection mechanisms that must be employed.

Concurrent error detection methods are extremely popular among dependability methods, against transient faults. Concurrent error detection mechanisms in COTS-based systems have been classified as follow:

- Structural-based mechanisms
- Behavioral-based mechanisms

Structural-based mechanisms are based upon hardware replication. For COTS-based systems, hardware replications can be applied at chip-level, such as master/checker [12] mechanism, and system-level.

Behavioral-based mechanisms extract an abstraction from the application program, memory access etc., usually performed during “compile time”, and checking the abstraction during runtime. It has been indicated that more than 70% of all transient faults lead to deviation from the program’s normal instruction execution flow, i.e., Control Flow Errors (CFE) [13]. Control Flow Checking (CFC) techniques (i.e., techniques to detect CFEs) have been known as an effective concurrent error detection method [14]. Most of the CFC techniques are using signature monitoring technique. In this technique, at setup time, the program is decomposed into basic blocks of instructions and a signature is derived from each basic block and saved somewhere, during runtime the signatures based on the basic blocks will be regenerated and compared with the saved one. CFC techniques can be implemented by pure software such as CFCSS [15] and feature specific CFC [16], pure hardware such as watchdog direct processing (W-D-P) [17] and CFCET[9], or hybrid (combined hardware-software) such as TTA[18] and CIC[19].

The workload program in CFCSS [15] is divided into basic blocks. The blocks in the program are assigned different arbitrary signatures, which are embedded into the program during compile time. A run-time signature is generated using XOR function and compared with the embedded signatures when instructions are executed.

Feature specific CFC [16] is a pure software control flow checking technique. In this technique, the program is decomposed into basic blocks of instructions and partition blocks between them. A signature is derived from each block (i.e., basic block and partition block) at the compile time, which is the number of instructions in the block. At runtime, the technique uses performance monitoring in modern COTS processors and employs their internal counters to regenerate the signatures (i.e., instructions executed in each block) and compares them with saved ones.

Usually, the big problem of software-based CFC is the weakness of detecting an error in program crash or CPU crash states. The above drawback of the software-based CFC techniques can be eliminated in hardware based approaches.

The watchdog direct processing (W-D-P) [17] and the CFCET [9] techniques are pure hardware and they do not need any program modification.

The W-D-P verifies the application program using a separate checking program executed by a watchdog processor (watchdog program). In this technique, each application program is represented by a reference control flow graph (i.e., sequencing nodes and destination nodes) and the watchdog program shadows the application program and contains one instruction for each node in the application program.

The CFCET uses the internal execution tracing feature in modern COTS processors, which provides the ability to monitor the addresses of the taken branches in a program at run-time, and an external watchdog processor to detect any violation from branch address saved at compile-time.

As these techniques control some processor pins signals to extract signatures, they cannot be applied to microcontrollers-based system.

TTA [18] and CIC [19] are hybrid CFC techniques. The TTA technique decomposes the workload program into branch-free blocks (BFBs) and partition blocks (PBs). The scheme uses an external watchdog processor and combines five error detection mechanisms. The TTA uses three timers into the watchdog processors; BFB-timer, PB-timer and WL-timer to check each BFB, PB and whole workload execution time respectively. The address mechanism in TTA sends the size of a BFB in bytes when the BFB is entered. At the same time, the watchdog processor reads the start address of the BFB from the address bus and calculates the exit address of the BFB. At the end of the BFB, the watchdog processor is signaled. An error has occurred if the calculated exit address is different from the observed exit address. The phase mechanism in TTA checks the entering and exiting of each BFB and PB.

The CIC uses two external special pins, called event-ticking pins PM0 and PM1, which can signal out when an instruction is committed into the processor pipeline. The number of instructions executed in each BFB and PB, and also whole workload program are counted externally by the watchdog processor using the processor event-ticking pins.

This paper actually presents a concurrent error detection method based on HWSW-CFC technique. The proposed method employs the synchronous external burst interrupt in duplication microcontrollers and compares the run time program counters of the microcontrollers in a service routine. The main advantages of the proposed method are:

- Instead of using high costs ASIC components, the method uses low cost COTS processors to perform on-line system-level error detection
- It can be applied to the microcontroller-based system, and can also be applied to the processors with pipeline and on-chip caches.
- It can detect control flow errors caused by data errors.
- No modification of the workload programs is required, but it needs to add an interrupt service routine.
- Program size overhead is very low (only an interrupt service routine must be added)

## III. Error Models

The basic model of errors used in this work is a violation of program’s normal instruction execution flow which will be explained in this section. These violations can be caused by transient or permanent faults in the memory or address circuits [21]. Based on these faults, five types of error models are defined as follows:

*Error model 1:* Program Counter Error (PCE): a PCE occurs when a fault changes program counter bits and an illegal jump occurs.

*Error model 2:* Branch Condition Error (BCE): a BCE occurs when a data fault (data register, flag register or data memory) causes the condition of a branch instruction is changed and a taken branch changes to non-taken branch or vice versa.

*Error model 3:* Branch Insertion Error (BIE): a BIE occurs when one of the non-branch instructions in the program is changed to a branch instruction as the result of a fault and the branch instruction actually causes a taken branch.

*Error model 4:* Branch Target Modification Error (BTME): a BTME occurs when the target address of one branch instruction is modified as the result of a fault and this instruction actually causes a taken branch.

*Error model 5:* Branch Deletion Error (BDE): a BDE occurs when a fault causes a branch instruction of a program changes to a non-branch instruction.

## IV. Proposed Method

The proposed method uses a duplication and comparison technique at chip level for checking the correctness of the program's instruction execution flow. The program flow checking is done using motivation of synchronous external interrupts in both microcontrollers and comparison the run time program counters of the microcontrollers in the service routine regularly.

The hardware part of the method is shown in Fig. 1. It contains two microcontrollers that run an identical program with an identical external clock.

*Power on Reset:* It resets both microcontrollers when turn power supply is on.

*Pulse Generator for Synchronization:* this unit generates a pulse to motivate an interrupt for synchronization of the microcontrollers to start program execution.

*Pulse Generator for PC exchange:* this unit generates periodic pulses to motivate interrupts periodically for exchanging and comparing PCs between the microcontrollers to check the existence of any discrepancy.

The software part of the method is shown in Fig. 2. It contains two programs that run in both microcontrollers: 1) Synchronization Program and 2) PC Exchange Routine.

The Synchronization Program synchronizes two microcontrollers' program to start. It contains a sleep instruction and an interrupt routine that sets PC to the address of the original program. This microcontroller has an internal power on reset that delays (for several milliseconds) starting the program. This delay makes the microcontrollers execution asynchronous, because the two microcontrollers do not have exactly the same delay.

The PC Exchange Routine is regularly invoked. This routine sends its own PC register to another microcontroller, and then gets another microcontroller's PC and compares two PC contents (i.e., its own PC and got PC) to check the existence of any discrepancy.

The assembly or C codes of workload programs can be used to add the extra instructions needed to implement the method. The pseudo code of the Synchronous Program and PC Exchange Routine are shown in Fig. 3.

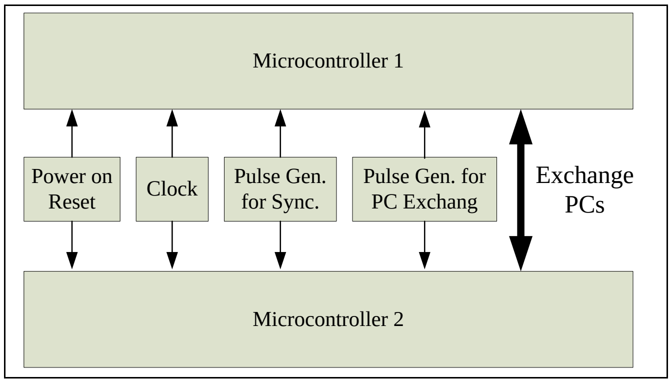

Figure 1. Hardware part of proposed method.

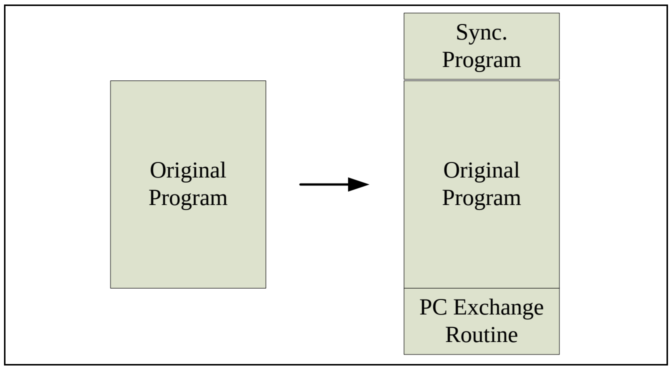

Figure 2. Software part of proposed method.

## V. METHOD EVALUATION

The architecture of the experimental system is shown in Fig. 4.

```
ORG 0
Initial Ports & Interrupts
start:   sleep  /* wait until sync. interrupt is occurred*/
--------------------------------------------------------------------------------
Int_Sync_Routine(){
    Read PC from Stack
    PC = Begin        /*set PC to address of  original program*/
    Write PC to Stack
    reti       /*after return from interrupt, original program is  beginning*/
   }
Begin:
/* main body of the Original Program */
PC_Exchange_Routine(){
    OwnPC = Read PC from stack
    Send OwnPC  to another micro
    OtherPC = Get PC from another micro
    If ( OwnPC =! OtherPC){
         ErrorReport()
         }
    reti
    }
```

Figure 3. Pseudo codes of extra software codes

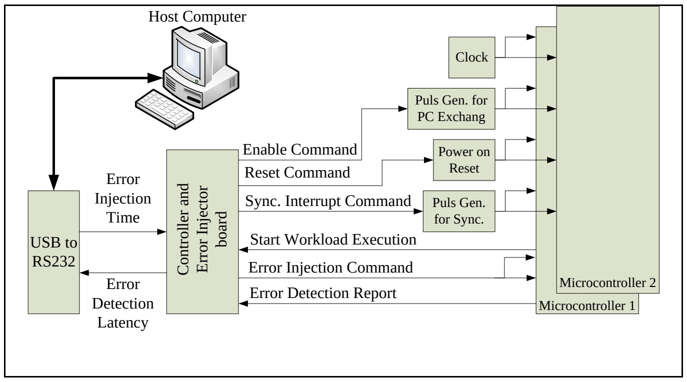

Figure 4. Software part of proposed method.

The system consists of three parts: an AVR microcontrollers board, a controller and fault injector board, and a host computer.

**AVR microcontrollers board:** the board has been equipped with two AVR microcontrollers that run an identical program, a 16 MHz clock generator for microcontrollers' clock pins, a monostable circuit to generate a pulse to invoke synchronization interrupt, and a clock generator with 100Hz, 1KHz and 10KHz frequencies to invoke PC exchange interrupts.

Two types of programs were executed on the AVRs board; the workload programs and a fault injector routine.

*The workload programs:* Three programs written in assembly language have been used in the experiment: 1) a 10 × 10 matrix multiplication ($M = A \times A^{-1}$), 2) a linked list (List) containing 100 records, and 3) a quick sort (QSort) containing 100 elements. 184 copies of the Matrix program, 124 copies of the List program, and 93 copies of the QSort program were consequently stored in the memory. These copies fill microcontrollers' flash memory (i.e., 32KB) with program codes. They were executed one after another in a loop until a fault occurs. The workload program were started when the system were reset.

*The fault injector routine:* The fault injection method used in these experiments is based on the software implemented fault injection (SWIFI). This paper focuses on the transient effects called SEUs (single event upsets). Several reports have mentioned that the SEU is important not only for the circuits operating in the space, but also for the digital equipments operating at the ground level [20]. It is reported in [21] that the majority (>60%) of control flow errors differ from the correct ones in only a single bit (i.e., SEU) of an address. SEUs are responsible for the modification of memory cells content (registers, internal memory, etc.). Usually, memories are protected against SEUs by means of error detecting/correcting codes (Hamming code, CRC code, Reed-Solomon code, etc.) [20]. In such cases, internal registers are of much important. Several reports have mentioned that SEU in the PC register are a major source of CFEs in comparison to other internal registers [21]. Therefore, to generate CFEs, the bits of the program counter (PC) are changed, one bit for each fault. This is done as follows: 1) the fault injector logic activates the INT0 pin of the microcontrollers, 2) the interrupt service routine reads the return address from the stack, changing a bit of the return address and then writing it back to the stack, 3) after returning from the interrupt service routine, the execution continues at an unexpected address due to the change of the value of the return address. To make sure about the coverage results, we assume that the probability distribution of the error occurring in PC bits (14 bits for 16K×16bits flash memory in AVR ATmega-32) will be uniform. The manager program on host computer issues the error injection command randomly in time during the execution of the workload program.

**Controller and Error Injector board:** the board has been equipped with a microcontroller and interface logic.

The interface logic establishes communication between the host computer and the controller board.

The controller board has five main tasks: 1) waiting to get a start command from the host and sending a Reset Command to reset the AVR Microcontroller Board, 2) waiting for the Start Workload Execution from the AVR Microcontroller Board and sending the Synchronization Command, 3) sending the Enable Command to activate pulse generator for PC exchange interrupts, 4) getting an Error Injection Time from the host and waiting until the time elapses and sends a command to activate INT0 pin of the two microcontrollers on AVR microcontroller board when a fault is to be injected, and 5) initialization of a timer to record the coverage and latency information.

**Host Computer:** The host computer contains a manager program and an offline data analyzer. The task of the host computer is to manage and control the whole experiment.

The offline data analyzer program analyses the raw data collected from the experiments and extracts the results.

## VI. EXPERIMENTAL RESULTS

This section presents the experimental results of the program size overhead, execution time overhead, error detection coverage, and error detection latency. Three programs written in assembly language, i.e., quick sort (QSort), matrix multiplication (Matrix) and linked list (List), have been used in this experiment.

**Error Detection Coverage**: Table 1 shows error detection coverage for each workload. The basic model of errors used in this evaluation is Program Counter Errors (PCE). Although, five types of errors have been modeled in Section III, all of them change the PC finally. The changed PC causes a violation of the program normal instruction execution flow. These violations can be caused by transient faults in the memory or address circuits. The error detection coverage is 100% based on fault model for all workloads. Although, it is obvious that the method can detect all PC errors, this method has been implemented for feasibility checking and to obtain other parameters.

**Program Size Overhead**: The assembly (or C) codes of workload programs can be used to add the extra instructions needed to implement the method. The structure of a program after inserting the extra instructions is shown in Fig. 3. Three programs (i.e., Matrix, List, and QSort) have been used as workloads and the extra codes needed to implement the method were added to the workloads. The extra instructions inserted in the workload programs incur program size. As shown in Table 1. program size overhead is about 0.47%. This parameter achieved similar results for different workloads because several copies of each workload were consequently stored in the flash memory. These copies fill microcontrollers' flash memory (i.e., 32KB) and extra codes for each workload is constant (i.e., 152 bytes), therefore, the program size overhead is approximately constant (i.e., 0.47%).

**Execution Time Overhead**: The method uses synchronous external interrupts in both microcontrollers and compares their run time programs in a service routine. Interrupt handling incurs execution time. A workload is run in two cases, with presence and no presence of PC exchange interrupts, and a timer is set for measuring the relevant execution times. The execution time overhead based on different PC exchange interrupt is shown in Table 2. As Table 2 shows, the percentages of execution time overhead in the method vary between 0.5% and 50%.

**Error Detection Latency**: error detection latency is the average time between fault injections to error detections. A timer is set to work after each fault injection. After each fault detection, the timer is read and saved. The error detection latencies are shown in Table 2 .The mean latencies varied between 1184 and 128147 cycles for different interrupt frequencies. The latency values were calculated with respect to the processor external clock frequency which was 16 MHz.

**Power Consumption Overhead**: Two microcontrollers were connected together to be able to work in a duplicate configuration. The microcontrollers have all inputs connected together, but only one of them drives the outputs. It is reasonable to assume that a duplicate configuration can make duplicate of power. In this method, the total consumption of power is risen about 100%.

TABLE I. DETECTION COVERAGE AND PROGRAM OVERHEAD

| | **Workloads** | | |
|---|---|---|---|
| | ***QSort*** | ***Matrix*** | ***List*** |
| Errror Detection Coverage(%) | 100% | 100% | 100% |
| Original Program Size (bytes) | 32600 bytes | 32482 bytes | 32360 bytes |
| Extra Codes (bytes) | 152 bytes | 152 bytes | 152 bytes |
| Program Size Overhead(%) | 0.47% | 0.47% | 0.47% |

TABLE II. TIME OVERHEAD AND DETECTION LATENCY

| | **Frequencies of interrupt** | | |
|---|---|---|---|
| | ***100Hz*** | ***1KHz*** | ***10KHz*** |
| Execution Time Overhead (%) | ≈0.5% | ≈5% | ≈50% |
| Error Detection Latency (CLK) | 128147 CLK | 12162 CLK | 1184 CLK |
| Error Detection Latency (msec) | 8009 μsec | 760 μsec | 74 μsec |

## VII. CONCLUSION AND FUTURE WORK

A hardware-software-based control flow checking method for COTS-microcontroller-based applications has been presented and evaluated. The method is based on duplication of microcontrollers and employs synchronous burst interrupts in both microcontrollers to monitor and compare their program counters (PCs). An implementation of the method has been experimentally evaluated. The method has been experimentally evaluated using AVR ATMega-32 microcontrollers and software-based error injection method. The results show that error detection coverage of the methods are 100% based on the fault models. The hardware and software overheads are about 100% and 0.5% respectively. The distinctive advantages of the proposed method over previous hardware-software-based error detection methods are the ability to apply in microcontrollers and the ability to detect control flow errors caused by data errors. For future works, we are going to add a system recovery mechanism after error detecting.

## REFERENCES

[1] J. A. Fisher, P. Faraboschi, and C. Young, ”Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools”,Morgan Kaufmann Publishers, ISBN: 1-55860-766-8, 2005.

[2] Y. He and A. Avizienis, “Assessment of the applicability of COTS microprocessors in high-confidence computing systems: a case study,” Proceedings of the international conference on dependable systems and networks (DSN2000), pp. 81–86, June 2000.

[3] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, ”PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures”, IEEE Transaction on Dependable and Secure Computing, vol. 6, no. 2, pp. 135-148, APRIL-JUNE 2009.

[4] S. Borkar, ”Designing Reliable Systems from Unreliable Components: the Challenge of Transistor Variability and Degradation,” IEEE Micro, vol. 25, issue: 6, pp. 10-16, November-December 2005.

[5] V. Stachetti , J. Gaisler, G. Goller, and C.L. Gargasson, ”32-bit processing unit for embedded space flight applications”, IEEE Tranaction on Nuclear Science, 43(3), pp. 873–878, 1996.

[6] J. Gaisler, ”A Portable and fault-tolerant microprocessor based on the SPARC 8 architecture”, Proceedings of international conference on dependable systems and networks, pp. 409–415, June 2002.

[7] S. Asserhall, T. Petersson, and P. Blomqvist, ”RAD HARD THOR microprocessor description”, Saab Ericsson Space, Document No P-TOR-NOT-0004-SE, issue 2, Jan 1999.

[8] P. Croll and P. Nixon, ”Developing safety-critical software within a CASE environment,” Proceedings of the IEE colloquium on computer aided software engineering tools for real-time control, pp. 8, April 1991.

[9] A. Rajabzadeh and S. Gh. Miremadi, ”CFCET: A hardware-based control flow checking technique in COTS processors using execution tracing,” Elsevier Journal of Microelectronic Reliability, vol. 46, issue 5-6, pp. 959-972, May-June 2006.

[10] P. Chevochot and I. Puaut, ”Experimental evaluation of the failsilent behavior of a distributed real-time run-time support built from COTS components,” Proceedings of the international conference on dependable systems and networks (DSN-2001), pp. 304–313, July 2001.

[11] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, ”Experimental evaluation of a COTS system for space applications,” Proceedings of the international conference on dependable systems and networks (DSN-2002), pp. 325–330, June 2002.

[12] A. Rajabzadeh, S. G. Miremadi, and M. Mohandespour, “Experimental Evaluation of Master/Checker Architecture Using Power Supply- and Software-Based Fault Injection”, Proceedings of the 10th IEEE International On-Line Testing Symposium (IOLTS 2004) Madeira Island, Portugal, pp. 239-244, July 2004.

[13] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, ”Low-Cost On-Line Fault Detection Using Control Flow Assertions,” Proceeding of the 9th IEEE International Online Testing Symposium (IOLTS'03), pp. 137-143, July 2003.

[14] A. Mahmood, and E. J. McCluskey, ”Concurrent error detection using watchdog processors-a survey,” IEEE Transaction on Computers, vol. 37, issue 2, pp. 160-174, February 1998.

[15] N. Oh, P. P. Shirvani, and E. J. McCluskey, ”Control-Flow Checking by Software Signatures”, IEEE Transactions on Reliability, vol. 51, no. 1, pp. 111-122, March 2002.

[16] A. Rajabzadeh, ”Feature Specific Control Flow Checking in COTS-based Embedded Systems”, Third IARIA International Conference on Dependability (DEPEND 2010), Venice, Italy, pp. 58-63, July 2010.

[17] T. Michel, R. Leveugle, and G. Saucier, ”A new approach to control flow checking without program modification,” Processding of the 21st international symposium on fault-tolerant computing, pp. 334-341, June 1991.

[18] S. Gh. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson, “Use of Time, Location and Instruction Signatures for Control Flow Checking”, Dependable Computing and Fault Tolerant System, IEEE Computer Society Press, vol. 10., ISBN 0-8186-7803-8, 1998, pp. 201–221.

[19] A. Rajabzadeh, S. Gh. Miremadi, and M. Mohandespour, ”Error detection enhancement in COTS superscalar processors with performance monitoring features,” Journal of Electron Testing: Theory and Applications (JETTA), pp. 553-567, 2004.

[20] B. Nicolescu, R. Velazco , M. Sonza-Reorda, M. Rebaudengo , and M. Violante, ”A software fault tolerance method for safety-critical systems: effectiveness and drawbacks”, Proceedings of the 15th symposium on integrated circuits and systems design (SBCCI-02), pp. 101-106, 2002.

[21] M. Rimen, J. Ohlsson, and J. Karlsson, ”Experimental evaluation of control flow errors”, Proceedings of the Pacific Rim international symposium on fault tolerant systems (PRFTS-95), pp. 238-243, December 1995.