Future Generation Computer Systems 87 (2018) 58–65

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Enhancing transient fault tolerance in embedded systems through an OS task level redundancy approach

Seyyed Amir Asghari [a], Mohammadreza Binesh Marvasti [a,*], Amir M. Rahmani [b,c]

[a] Department of Electrical and Computer Engineering, Kharazmi University, Tehran, Iran
[b] Department of Computer Science, University of California, Irvine, USA
[c] Institute for Computer Technology, TU Wien, Vienna, Austria

ARTICLE INFO

Article history:
Received 3 January 2018
Received in revised form 12 March 2018
Accepted 17 April 2018
Available online 3 May 2018

Keywords:
Embedded system
Fault tolerance
Operating system
Transient fault

ABSTRACT

In numerous safety critical applications, the use of high-reliability or radiation-tolerant equipment may not be a viable option due to the presence of several constraints (such as cost) and the need to utilize Commercial off-the Shelf (COTS) equipment. However, such equipment may not meet reliability requirements, and therefore certain appropriate measures need to be taken to enhance their reliability. In this paper, a fully software-based method is presented to increase the reliability of COTS equipment against transient faults. The reliability of COTS is increased by utilizing a task-level redundancy in operating system. The proposed method is evaluated using a software fault injection method and a full system prototype. The experimental results show that the proposed method increases the fault coverage up to 99.34%. Moreover, the proposed method can be used in embedded systems without any hardware, software, or information redundancy.

1. Introduction

Although the last decade has seen a rapid increase in the use of embedded systems, their reliability in critical missions is still limited. To alleviate the reliability issues, many studies have been done in permanent faults and propose different methods to detect them [1–5]. Transient faults that occur in highly radiated environments have also received a considerable attention. As the probability of SEU (Single Event Upset) occurrence is increased, the dimensions of the problem have been increased. This happens due to reduction in size and increase in the voltage levels of transistors [6–11].

Many techniques presented in literature to enhance fault tolerance require redundant peripheral hardware, which may pose high infrastructure and reconfiguration costs in many applications. As a consequence, many researches endeavor for fully software based techniques to increase fault tolerance in computer systems as well as embedded systems [12–20].

The work done in this paper is motivated by the fact that COTS hardware with COTS operating systems are nowadays widely utilized in embedded systems running critical tasks. Software based techniques that are implemented in operating system level or designed in instruction level, cannot be utilized in many embedded applications and they need specialized operating systems, compilers, and custom preprocessors. These methods usually result in extra overheads in memory and performance. In order to alleviate the foregoing issues, this paper presents a fully software-based method to increase fault tolerance at application level for the real time embedded systems. To the best of our knowledge, our method can be used in all COTS processors or operating systems. Moreover, the proposed method is capable of detecting data faults. To demonstrate the efficiency of our method, we have implemented it on a commonly used embedded system and evaluated it via a software fault injection method.

We propose a data fault tolerance method based on TTMR (Time Triple Modular Redundancy). In this method, voting is done by Manager Task. If an error happens on any tasks, the system may be failed. This failure is monitored by a technique called Control Flow Checking method, which is used as a terminology in embedded system domains. However in the software engineering papers [21,22], the Work Flow Checking phrase is utilized for the monitoring technique. Since our system target is embedded systems, all the monitoring techniques in this paper are expressed by Control Flow Checking.

The organization of the paper is as follows. In the following section, a review of literature in this field is given. In Section 3, the proposed method is described. The laboratory system on which the method is implemented is explained in Section 4, while in Section 5, the experimental results are analyzed. Finally, the conclusion and future work are presented in Section 6.

* Corresponding author.
E-mail addresses: asghari@khu.ac.ir (S.A. Asghari), marvasti@khu.ac.ir (M.B. Marvasti), amirr1@uci.edu (A.M. Rahmani).

https://doi.org/10.1016/j.future.2018.04.049
0167-739X/

2. Related works

Provision of redundancy is one of the classical methods of ensuring fault tolerance in computer systems. The approaches in this respect can be classified into four basic types: (i) hardware redundancy, (ii) software redundancy, (iii) information redundancy, and (iv) time redundancy [23–28].

Time redundancy based methods are incorporated in different levels; such as instruction level, procedure level, operating system level and task level. Redundancy based methods in operating system level that are used in distributed systems can include time redundancy. In these methods, the operating system itself is in charge of voting the task. Triple Modular Redundancy (TMR) method is one of the redundancy based methods that is applicable in both time and hardware redundancies [1–3].

A time redundancy method is presented in [4], in which each task is run twice. After the first and the second run, the operating system compares the execution results and initiates a third run in the case of mismatching of these two results. One of the disadvantages of this method is its application dependency and the need to change the kernel. Furthermore, the detection of a fault in the operating system cannot be guaranteed by this method.

Some other methods that aim to increase fault tolerance have concentrated on Control Flow Checking. These methods that are implemented in hardware or software, can only detect control flow checking errors [5–8]. It is worth mentioning that these faults constitute 70 percent of total faults [7–14] [27–33]. In hardware based methods, the classic approach is to use a peripheral hardware, called watchdog processor, which checks the program control flow in the main processor via signature monitoring or bus monitoring methods [8]. With software methods no additional hardware is required; the process is entirely performed in software. Among the well known software based control flow checking methods, Relationship Signature for Control Flow Checking (RSCFC) [9] and Control Flow Checking by Software Signature (CFCSS) [10] are the most well-established methods. However, since these methods require a specific preprocessor, they cannot be utilized in most embedded systems.

Some other software based classical methods for fault detection use exception handling, watchdog timer and assertion, which can be utilized in many embedded systems. In the monograph [3], a general study is given, in which the methods proposed for recovery are divided into two general classes, named as forward recovery and backward recovery. Backward recovery methods are the simplest recovery methods and correspond to the use of check pointing or reset methods. Triple Modular Redundancy is one of the most well known forward recovery methods in which the result of the failed module is masked with the results of correct modules [11].

For the evaluation of the fault tolerance capability of computer systems, many techniques can be seen in the literature that can be divided into two general classes as theoretical and practical techniques. Practical techniques that are also called fault injection are divided into four classes: (i) simulation based fault injection, (ii) software based fault injection (iii), physical fault injection, and (iv) debug mode based fault injection. A number of tools are available for fault injection such as Exception [12], GOOFI [13], DOCTOR [14], FERRARI [15] and BDM based technique. One of the methods that utilizes debug mode for fault injection is presented in [16].

3. The proposed method

In order to increase fault tolerance in a real time multitask system, we propose an efficient time redundancy method for embedded systems that are based on COTS processors with multitask operating systems. It is assumed that a real time operating system with a priority based scheduling is used, which is common in modern embedded systems.

The method is fully software based and no additional hardware or changes to the operating system kernel are needed. The main objective of the proposed method is to detect transient faults in the processor core. To detect other fault types, such as memory or input–output faults, other fault detection techniques such as Error Detection and Correction (EDAC) [2] and information redundancy are utilized.

In the proposed method, the software developer produces two redundant versions of a task and also a Manager Task for every computing task. The Manager Task is in charge of input preparation, scheduling, and comparing the outputs of the redundant tasks as well as recovery operation of an error. In fact, this method is a TTMR, however for improving system performance, the first and the second version of the task is run first and then, in case that the results of these two versions differ, the third version of the task will run. Once the Manager Task detects a mismatch in the outputs of the redundant task, it runs the third version of the task and then recreates the first and the second faulty versions. In this way, the status of the faulty task stack is returned to its initial status and the task is prepared for another running. In Fig. 1, the system behavior is shown in two states, which are fault existence and no fault existence. As seen in Fig. 1, the Manager Task performs input reception phase, and then the inputs of first and second versions of the task are adjusted accordingly. Then the Manager Task assigns the processor to these two versions for a specific time, which equals to the sum of the longest time of running the first and the second versions.

Since the first version takes priority over the second version, it must run first and then the second version runs next. Afterwards, the Manager Task preempts the processor and compares output results of these two tasks. If the results are equal (Fig. 1(a)), the Manager Task starts the output adjustment phase and returns the results in output. The pseudo code of the algorithm is given in Fig. 2.

As seen in Fig. 1(b), if a failure happens in either the version 1 or 2, the Manager Task starts running the third version. If the third version of the task prioritizes the Manager Task, the Manager Task preempts the processor just after running Task 3 and the voting phase is run.

If one of the results has the major repetition, the failed task is recreated and then the output adjustment phase is carried out. Otherwise based on the type of application, a specific operation (for e exception handling) is done.

In this method, it is assumed that in each execution period only one fault occurs and enough time for system recovery is available. More precisely, if each task has a deadline period T, the execution time of each version is C, the execution time of the Manager Task is B, and D is the time necessary for running the other tasks and the operating system functions, the equation $B + 3C \leq T - D$ must always be true. It is also necessary to prioritize the Manager Task to the redundant tasks because if the first and second version of a task encounter any fault and face a livelock (infinite loop), the Manager preempt the CPU at the assigned time.

If the results of the third run deviate from the previous results, the system will not be able to generate an output in that state and there is no need to reset the whole system. In the case of any failures in the Manager Task or in the operating system level background tasks for system recovery, it is necessary to reset the system. A watchdog timer can also be used for detecting faults that result in timeout. In this case, timeout event can also result in a system reset.

If exception occurs in the version 1 and 2 of a task, the assumed task should suspend itself and the related exception handling routine starts handling the exception. Fig. 2 illustrates the management algorithm implemented in the Manager Task. As seen

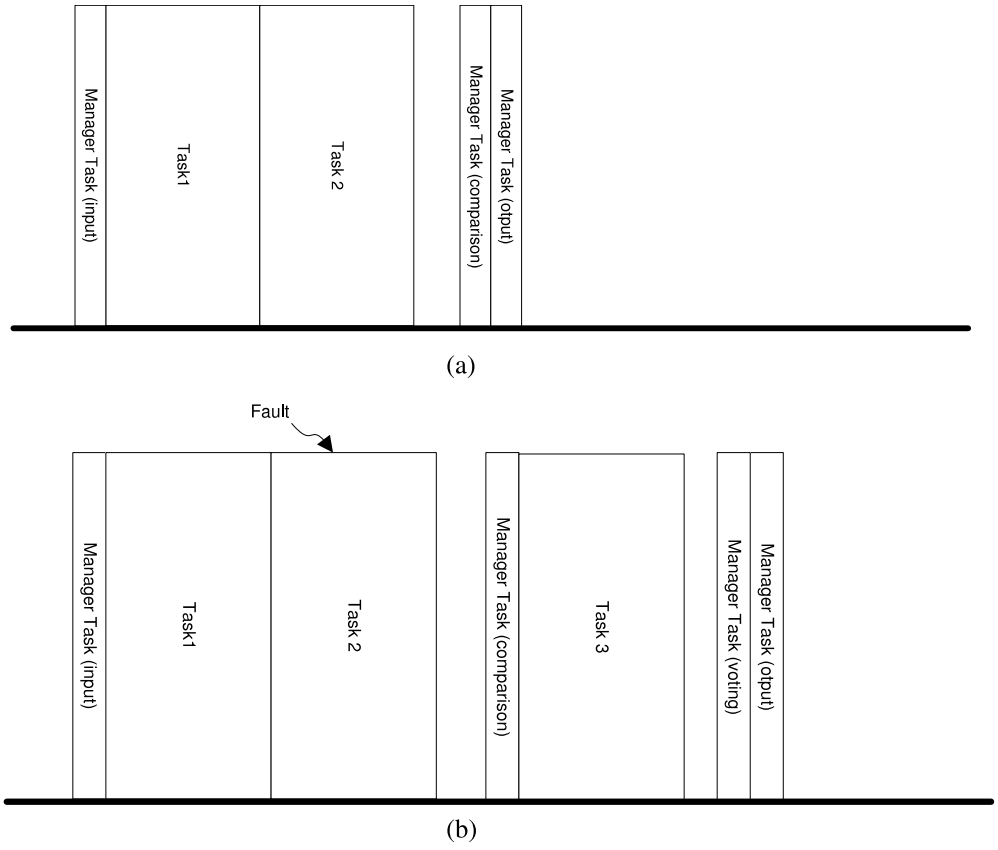

Fig. 1. Running flow (a) without any fault (b) with a fault.

```
void manager_task(){
    create tasks();
    init_watchdog_timer();
    while(1){
        inputs();
        os_resume(task_a);
        os_resume(task_b);
        os_delay(task_a plus task_b excution time);
        compare_result = compare(a_result, b_result);
        if (compare result = false){
            os_suspend(task_a);
            os_suspend(task_b);
            os_resume(task_c);
            os_delay(task_c_excution time);
            voting_result = voting(task_a_result, task_b_result, task_c_result);
            switch(voting_result)
            {
                 case a_result != b_result == c_result: error_recovery();outputs();
                 case b_result != a_result == c_result: error_recovery();outputs();
                 case others : error_in_manager();reset();
            }
        }else   outputs();
    }
}
```

Fig. 2. Manager task algorithm.

in Fig. 2, the management algorithm uses the utilized operating system functions.

Control Flow Checking techniques and value control with assertion are utilized for improving fault tolerance in the Manager Task. In this paper, we use a method presented in [5] and adjust the Manager Task Control Flow Graph (CFG) based on basic blocks at application level to detect inter-block flow control errors.

Since operating system functions are used in the Manager Task, each of these functions is assumed as a basic block. Input reception and output adjustment phases are also considered to be basic blocks. Fig. 3 shows the Control Flow Graph of the Manager Task for the management algorithm.

For inter-block error detection in Manager Task Control Flow Graph, we define *S[ID]* as a global variable. This variable *S[ID]* is always updated before entrance into a block. Then after exiting from this block, the value of this variable is compared with the expected value and in the case of mismatch, it can be concluded that a control flow error has occurred.

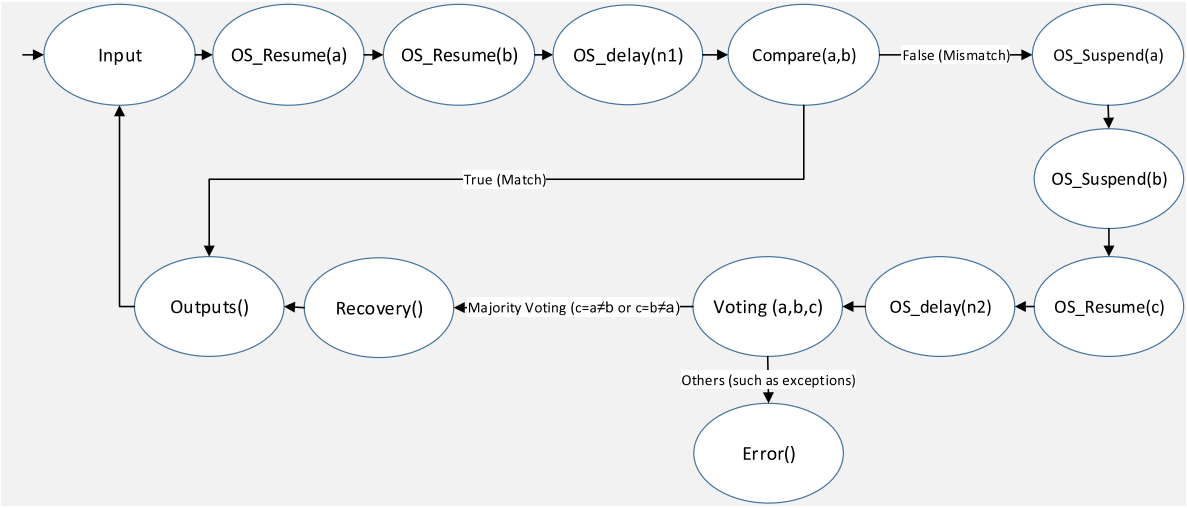

Fig. 3. Manager Task control graph in application level.

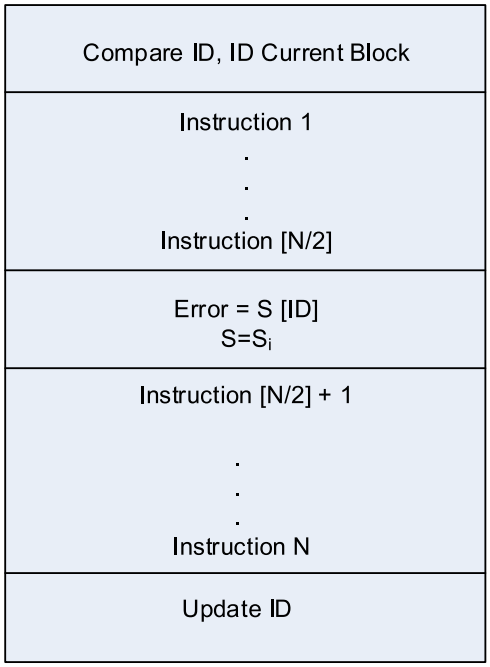

Fig. 4. The basic block under the suggested method for Manager Task graph [5].

Fig. 4 shows a basic block using this method. The Set and Test instructions are defined based on each block position in control flow graph. If assertion is used in Manager Task output, some of the data errors taken place in the Manager Task can be detected.

Since the execution time of the Manager Task compared with other tasks is short, the probability of a failure in this task is less than the other tasks. Therefore, in systems with heavy results for comparison, it is better to use a signature comparing technique rather than comparing all the results. In this way, the Manager Task execution time is reduced but the signature computation overhead is added to the process.

In our proposed method, the execution time equals to $M + in\text{–}out + 2T + O$, where M is the Manager Task execution time without computing input–output phase time, $in\text{–}out$ is input–output execution time, T is the execution time of each version of a task, and O is the execution time of the operating system function. By comparing this value against the task execution time in addition to input–output operation time, the overhead is expressed by Eq. (1):

$$n = \frac{M + in\_out + 2T + O}{T + in\_out} \tag{1}$$

The memory overhead computation is similar to Eq. (1). In fact the data memory overhead equals to the sum of the global and the local variable memories and stack memory of each task. If we assume that variable memories and stack memories are equal, the data memory overhead will be 300% in all tasks.

If M_{MEM} is the required instruction memory for the Manager Task and T_{MEM} is the required instruction memory for each task, the instruction memory overhead is denoted by n and given by Eq. (2):

$$n = \frac{M_{MEM} + 3T_{MEM}}{T_{MEM}} \tag{2}$$

4. Experimental setup

In order to evaluate the efficiency of the proposed method, an experimental setup that has several elements is implemented.

4.1. Processor

To evaluate the efficiency of the proposed method, a Phycore-MCF5485 evaluation board is utilized. This board includes a 32-bit processor with the ColdFire architecture, 32 MB flash memory, and 128 MB SDRAM memory [17]. The ColdFire processor has several exceptions with the capability of detecting many different errors. Table 1 shows some of these exceptions [18].

4.2. Operating system

To provide multitask capability, an open source real time operating system called MicroC/OS-II is used. This operating system utilizes preemptive scheduling based on task priority with micro kernel.

Table 1
Some of the ColdFire exceptions [18].

Vector number	Stacked program counter	Assignment
2	Faulty instruction address	Access error
3	Faulty instruction address	Address error
4	Faulty instruction address	Illegal instruction
5	Faulty instruction address	Divide by zero
8	Faulty instruction address	Privilege violation
10	Faulty instruction address	Unimplemented line-a opcode
11	Faulty instruction address	Unimplemented line-f opcode
14	Faulty instruction address	Format error
15	Next	Uninitialized interrupt
24	Next	Spurious interrupt
61	Faulty instruction address	Unsupported instruction

4.3. Benchmark

To evaluate the system behavior, Stanford Integer Benchmark is utilized. This benchmark includes bubble sort, quick sort, matrix multiplication, and several other well-known applications. The first two applications are used in evaluation of the proposed method. In each application, one Manager Task and three versions of a task with different priorities are created to run the desired benchmark function.

4.4. Supervisor computer

To analyze the behavior of the system under study, a Phycore-MCF5485 board is connected to a PC via RS232 link. The software of Phycore board is adjusted in a way that it sends the location and the time of fault injection, the results of the error detection mechanism, and the exception-reset happening to the supervisor computer. Moreover, for analyzing the correctness of the error detection mechanism, it is necessary to compare the system output with the expected output that is performed in the supervisor computer through the Fault Injector Analyzer software.

4.5. Fault model

The analyzed fault model in this experiment is a single bit flip transient fault which is the most commonly seen SEU. SEU usually happens due to radiation or the crashing of energy full ions onto transistors [19,20,23–29]. The location of fault in this experiment is in processor core registers that include 8 data registers (D0–D7), 8 address registers (A0–A7), PC and SR registers [20].

Fault time and location are selected uniformly and randomly. It is also assumed that in each run, at-most error takes place. It should be noted that in the input reception phase and output adjustment, fault injection is not executed.

After the fault injection and result evaluation processes, the system is reset by the watchdog timer embedded in the MCF5485 processor.

4.6. Fault injection

Our fault injection method [30] is SWIFI (Software Implemented Fault Injection) in which one of the timers of the processor is set to interrupt after the lapse of a random time period. In the interrupt routine of this timer, the system fault is injected according to the desired fault model. In the interrupt routine, all processor core registers are saved in a stack. Afterwards, the value of one of the stored registers is changed by using the value of the stack pointer and the fault model. Finally, all the stored registers in the stack are returned to the processor. There is a capability of fault injection to all processor registers, except the stack pointer, because any changes in the value of the stack pointer result in changes in the values of the other registers, which is equal to changing the fault model.

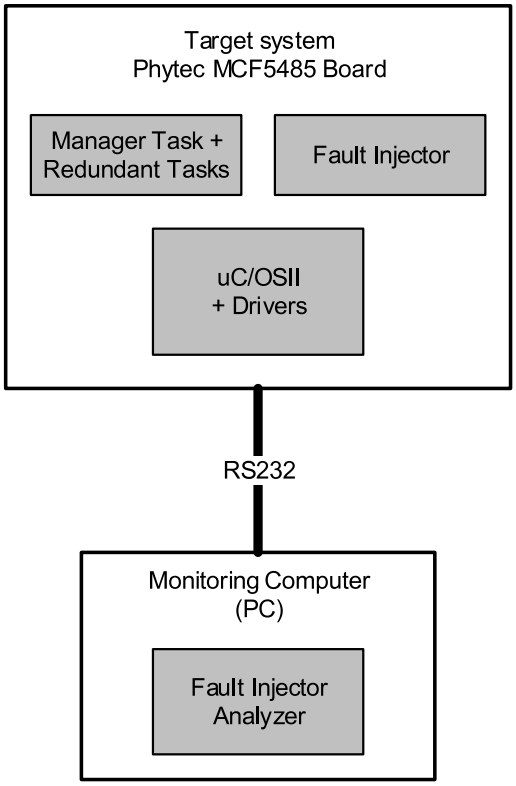

Fig. 5. Experimental system block diagram.

In Fig. 5, the block diagram of the system is shown. Fault injection interval of 10 ns is considered in this procedure (using timer setting).

Fig. 6 illustrates the stack content in the interrupt service routine after the register content storage.

5. Experimental results

Fig. 7 shows the possible system states that may arise after a fault injection. Here are the explanations of each state:

- *State A* is the case when the fault occurrence causes the program to crash. One of the reasons that *state A* happens in MCF5485 processor is the effort to access to the address space in which it is not in the memory range address.
- *State B* occurs when exception takes place during the execution of the Manager Task.
- *State C* happens when the applied Control Flow Checking method in the Manager Task detects a control flow error occurrence in the stack. If the error is not detected by any of these methods, *state D* will happen.

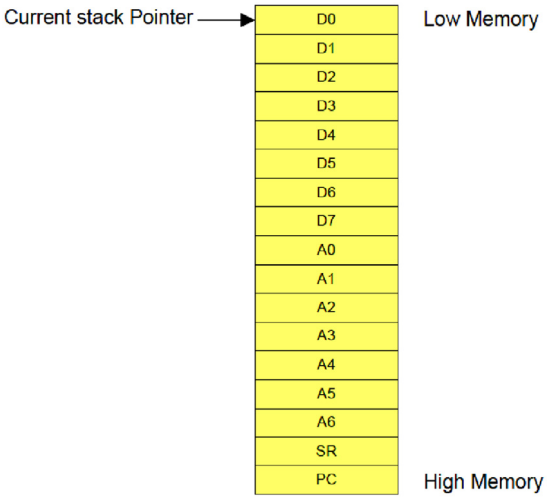

Fig. 6. Stack content in fault injection timer interrupt service routine.

- *State E* occurs when the error is detected by the repetition and comparison mechanisms. However, in this state the output data is invalid.
- *State F* happens when the output is correct when recovery is done.
- *State G* takes place when there are latent or overwritten faults.

According to the state explanations, in the *states A*, *B* and *C*, output is generated and in the *states D* and *E*, the output is faulty.

In order to evaluate the proposed method, the behavior of the system is analyzed for two cases of single task and multi-tasks.

Single task is when a task has a role of fault injection without the use of any software based method. Multi-tasks is the state in which the method proposed in this paper is utilized.

The outcomes of fault injection in each category of processor, registers are analyzed separately. Table 2 shows the relative occurrences (in percentages) of the *states A* to *G* when a fault is injected into the PC register 1000 times. In this table, "simple" is the situation that the proposed fault tolerance method is not used, and "embedded" is the case that the proposed method is used.

As seen in Table 2, every column of this table corresponds to one of the states shown in Fig. 7. The figures in Table 3 are the ones obtained when a fault is injected into SR register of MCF5485 processor, again 1000 times. This register consists of status register bits, interrupt priority mask, and processor mode. The interrupt priority mask bit is very important to manage the critical section and disable the interrupts. Table 4 shows the similar results in the case of the address registers of MCF5485 processor. These registers consist of registers A0 to A6. It is clear that in this state, latent faults rate is increased in relation to PC register. Table 5 shows the results of the fault injection into data registers, and finally Table 6 shows the average of the fault injection final results. It is seen that when the proposed approach is used, it is possible to reduce the incorrect output rate from 8.23% ((7.56+1.43+17.24+6.69)/4) to 0.66% ((0.8+0.12+1.73+0)/4) and increase the fault coverage from 91.77% to 99.34%.

As seen in these tables, timeout occurrence rate does not change so much in the cases of single task and multitasks. However, the errors result in system exceptions decrease steeply in the multitask state as in this state the occurred exceptions in redundant tasks do not lead to stopping of all system operations.

Fig. 8 illustrates the results of the fault injection in a simple run for different registers without using our proposed method. For

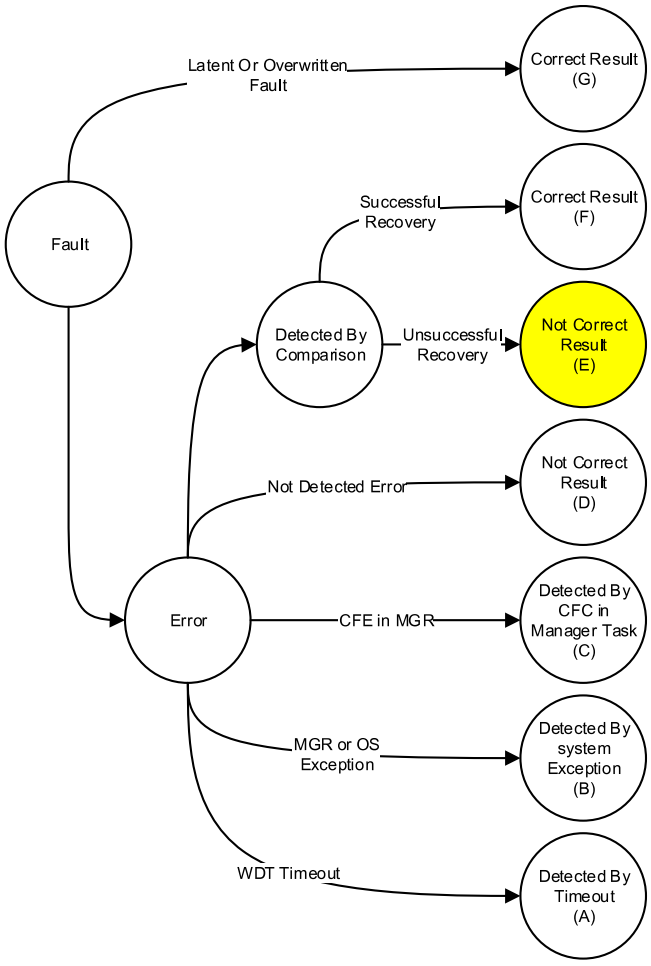

Fig. 7. System states after fault injection.

fault detection, basic processor fault detection mechanisms such as error control methods are used in this state.

Fig. 9 shows the results of fault injection for different registers when the proposed method is used. As seen in Fig. 9, the worst case of the fault injection occurs in the Data registers, while the best result is in the SR register.

It is worthwhile to mention that in the simulation, the states that generate output after the deadline T are not considered because those outputs miss their deadline. The experimental results also indicate that the error detection and the error correction rates in processor registers are significantly improved when the proposed method is used. However, our proposed method is not proper to be used in the hard real-time embedded system, which they must absolutely hit every deadline.

6. Conclusion and future work

In this paper, the fully software-based method has been provided to increase the reliability of COTS equipment against transient faults. The reliability has been increased by utilizing operating system task-level redundancy as a software level redundancy. It has been shown that by utilizing the proposed method, the percentage of fault coverage is increased by up to 99.34%. Moreover, the propose method offers considerable benefits to be used in embedded systems (not the hard real-time one) without any hardware, software, or information redundancy. Therefore, it is feasible to achieve an acceptable rate of SEU even in safety-critical

Table 2
Occurrences of different states after the injection of 1000 faults into PC register (in %).

Simple or Embedded	Benchmark	State A	State B	State C	State D	State E	State F	State G
Simple	Quick Sort	26.40	55.20	0.00	12.00	0.00	0.00	6.40
Embedded		26.50	13.17	1.60	0.60	1.00	46.5	10.63
Simple	Bubble Sort	28.46	57.89	0.00	3.12	0.00	0.00	10.53
Embedded		27.15	13.17	2.20	0.00	0.00	46.91	10.57

Table 3
Occurrences of different states after the injection of 1000 faults into SR register (in %).

Simple or Embedded	Benchmark	State A	State B	State C	State D	State E	State F	State G
Simple	Quick Sort	1.67	0.56	0.00	1.67	0.00	0.00	96.10
Embedded		6.97	0.70	0.00	0.00	0.00	2.65	89.68
Simple	Bubble Sort	0.54	0.27	0.00	11.72	0.00	0.00	87.47
Embedded		11.80	0.76	0.00	0.00	0.00	31.90	55.54

Table 4
Occurrences of different states after the injection of 1000 faults into address register (in %).

Simple or Embedded	Benchmark	State A	State B	State C	State D	State E	State F	State G
Simple	Quick Sort	9.77	18.51	0.26	0.00	0.00	0.00	71.46
Embedded		10.82	1.77	0.35	0.18	0.00	13.65	73.23
Simple	Bubble Sort	3.45	0.84	0.00	2.86	0.00	0.00	92.85
Embedded		12.34	0.94	0.06	0.06	0.00	3.70	82.90

Table 5
Occurrences of different states after the injection of 1000 faults into data register (in %).

Simple or Embedded	Benchmark	State A	State B	State C	State D	State E	State F	State G
Simple	Quick Sort	12.06	0.00	0.00	11.36	0.00	0.00	76.58
Embedded		13.93	0.00	0.40	0.00	0.40	13.04	72.23
Simple	Bubble Sort	16.58	1.31	0.00	23.12	0.00	0.00	58.99
Embedded		18.61	0.00	1.64	0.41	2.66	18.51	58.17

Table 6
Total average results of fault injection.

Simple or Embedded	State A	State B	State C	State D	State E	State F	State G
Simple	12.36	16.82	0.00	8.23	0.00	0.00	62.55
Embedded	16.01	3.80	0.78	0.15	0.51	22.10	56.61

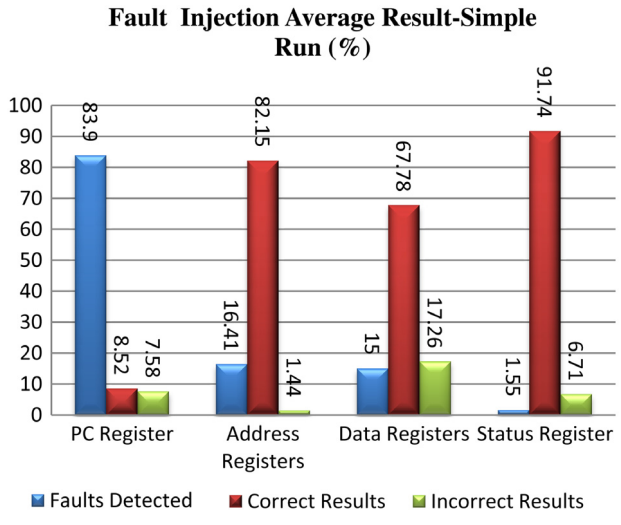

Fig. 8. The results of the fault injection in a simple run without the use of the proposed method (in %).

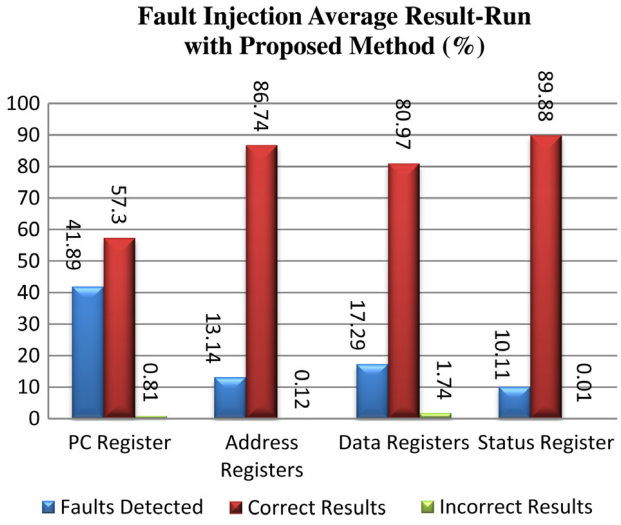

Fig. 9. The results of the fault injection using the proposed method for different registers (in %).

missions using standard processors and COTS operating systems. As a future work, we are going to apply Control Flow Checking to application tasks to reduce Control Flow Errors. Moreover, we are going to apply Data Error Detection techniques to application tasks in order to lessen error rate on Task versions and improve system performance significantly.

References

[1] P. Printo, A. Benso, Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation, Kluwer Academic Publishers, 2004.
[2] M.L. Shooman, Reliability of Computer Systems and Networks, Fault Tolerant Analysis and Design, John Wiley and Sons, 2002.
[3] I. Koren, C. Mani Krishna, Fault Tolerant Systems, Elsevier, 2007.
[4] A. Ejlali, B.M. Al-Hashimi, M.T. Schmitz, P. Rosinger, S.G. Miremadi, Combined time and information redundancy for seu-tolerance in energy-efficient real-time systems, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 14 (4) (2006) 323–335.
[5] S.A. Asghari, H. Taheri, H. Pedram, O. Kaynak, Software-based control flow checking against transient faults in industrial environments, IEEE Trans. Indust. Inform. (TII) 10 (1) (2014) 481–490. http://dx.doi.org/10.1109/TII.2013.2248373.
[6] A. Abdi, S.A. Asghari, H. Taheri, H. Pedram, An effective software implemented data error detection method in real time systems, Adv. Intell. Soft Comput. (Springer) 166/2012 (2012) 919–926. http://dx.doi.org/10.1007/978-3-642-30157-5_91.
[7] M. Maghsoudloo, H.R. Zarandi, N. Khoshavi, An efficient adaptive software-implemented technique to detect control-flow errors in multi-core architectures, Microelectron. Reliab. 52 (2012) 2812–2828.
[8] F. Khosravi, H. Farbeh, M. Fazeli, S.G. Miremadi, Low cost concurrent error detection for on-chip memory based embedded processors, embedded and ubiquitous computing, EUC, in: 2011 IFIP 9th International Conference on, 2011, pp. 114–119.
[9] A. Li, B. Hong, Software implemented transient fault detection in space computer, Aerosp. Sci. Technol. 11 (2–3) (2007) 245–252.
[10] N. Oh, P.P. Shirvani, E.J. McClusky, Control flow checking by software signature, IEEE Trans. Reliab. 51 (1) (2002) 111–122.
[11] L.L. Pullum, Software Fault Tolerance Techniques and Implementation, Artech House, London, 2001.
[12] J. Carreira, H. Madeira, J.G. Silva, Xception: Software fault injection and monitoring in processor function units, IEEE Trans. Softw. Eng. 24 (2) (1998).
[13] J. Vinter, J. Aidemark, D. Skarin, R. Barbosa, P. Folkesson, J. Karlsson, An Overview of GOOFI - A Generic Object-Oriented Fault Injection Framework, Chalmers University of Technology, 2005.
[14] S. Han, H. Rosenberg, K. Shin, DOCTOR: An integrated software fault injection environment, in: Proceeding of International Computer Performance and Dependability Symposium, Springer Verlag, 1996.
[15] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, FERRARI: A flexible software based fault and error injection system, IEEE Trans. Comput. 44 (2) (1995) 248–260.
[16] S.A. Asghari, H.R. Zarandi, H. Pedram, M. Ansarinia, M. Khademi, A fault injection attitude based on background debug mode in embedded systems, in: Proceeding of the International Conference on Computer Design, CDES 2009, 2009, pp. 100–104, USA.
[17] PhyCore-MCF548x Hardware Manual, Phytec Technology, 2005.
[18] MCF548x Refrence Manual. Freescale Semiconductors, 2006.
[19] J. Lebrosse, MicroC/OS-II The Real-Time Kernel, Newnes, 2002.
[20] ColdFire Programmers Reference Manual Freescale Semiconductor, 2005.
[21] T. Baker, D. Lamb, A. Taleb-Bendiab, Facilitating semantic adaptation of web services at runtime using a meta-data layer, Developments in E-systems Engineering, 2010.
[22] T. Baker, O.F. Rana, R. Calinescu, R. Tolosana-Calasanz, J.A. Bañares, Towards autonomic cloud services engineering via intention workflow model, GECON, 2013, pp. 212–227.
[23] J.A. Felix, J.R. Schwank, M.R. Shaneyfelt, J. Baggio, P. Paillet, V. Ferlet-Cavrois, Paul E. Dodd, S. Girard, E.W. Blackmore, Test procedures for proton-induced single event latchup in space environments, IEEE Trans. Nucl. Sci. 55 (4) (2008) 1–5.
[24] M.R. Shaneyfelt, J.R. Schwank, P.E. Dodd, J.A. Felix, Total ionizing dose and single event effects hardness assurance qualification issues for microelectronics, IEEE Trans. Nucl. Sci. 55 (4) (2008) 1926–1946.
[25] R.C. Lacoe, Improving integrated circuit performance through the application of hardness-by-design methodology, IEEE Trans. Nucl. Sci. 55 (4) (2008) 1903–1925.
[26] J.A. Maestro, P. Reviriego, Reliability of single-error correction protected memories, IEEE Trans. Reliab. 58 (1) (2009) 193–201.
[27] M. Murat, A. Akkerman, J. Barak, Electron and ion tracks in silicon: Spatial and temporal evolution, IEEE Trans. Nucl. Sci. 55 (6) (2008) 3046–3054.
[28] S. Rocheman, F. Wrobel, J.R. Vaillé, F. Saigné, C. Weulersse, N. Buard, Th. Carrière, Neutron induced energy deposition in a silicon diode, IEEE Trans. Nucl. Sci. 55 (6) (2008) 3146–3150.
[29] P. Reviriego, J.A. Maestro, Study of the effects of multibit error correction codes on the reliability of memories in the presence of MBUs, IEEE Trans. Device Mater. Reliab. 9 (1) (2009) 31–39.
[30] S.A. Asghari, H. Pedram, H. Taheri, M. Khademi, A new background debug mode based technique for fault injection in embedded systems, Int. Rev. Model. Simul. (IREMOS) 3 (3) (2010) 415–422.
[31] Du Boyang, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-García, A. Lindoso, L. Entrena, Online test of control flow errors: a new debug interface-based approach, IEEE Trans. Comput. 65 (2016) 1846–1855.
[32] H. Cho, E. Cheng, T. Shepherd, Ch. Cher, S. Subhasish Mitra, System-level effects of soft errors in uncore components, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 36 (2017) 1497–1510.
[33] A. Lindoso, L. Entrena, M. García-Valderas, L. Parra, A hybrid fault-tolerant LEON3 soft core processor implemented in low-end SRAM FPGA, IEEE Trans. Nucl. Sci. 64 (2017) 374–381.

Seyyed Amir Asghari received his B.Sc. degree in 2007 (hardware engineering major), M.Sc. and Ph.D. in 2009 and 2013 respectively (computer architecture major) from Amirkabir University of Technology. In 2013, He was a visiting researcher in Mechatronics Research Center, Bogazici University, Turkey. His current research interests include fault tolerant design, real time embedded system design, and operating systems. He has served as a faculty member in the the Department of Electrical and Computer Engineering at Kharazmi University.

Mohammadreza Binesh Marvasti received the M.Sc. degree in computer engineering from the University of Tehran, Tehran, Iran, in 2007 and the Ph.D. degree in electrical and computer engineering from McMaster University, Hamilton, Canada, in 2013. His current research interests include computer architecture, low power digital design, embedded system design, FPGAs, approximate computing, and various aspects of on-chip interconnection network, including power/area/delay modeling and evaluation. He has served as a faculty member in the Department of Electrical and Computer Engineering at Kharazmi University.

Amir M. Rahmani received his Master's degree from Department of ECE, University of Tehran, Iran, in 2009 and Ph.D. degree from Department of Information Technology, University of Turku, Finland, in 2012. He also received his MBA jointly from Turku School of Economics and European Institute of Innovation & Technology (EIT) ICT Labs, in 2014. He is currently Marie Curie Global Fellow at University of California Irvine (USA) and TU Wien (Austria). He is also an adjunct professor (Docent) in embedded parallel and distributed computing at the University of Turku, Finland. His research interests span Self-aware Computing, Energy-efficient Many-core Systems, Runtime Resource Management, Healthcare Internet of Things, and Fog/Edge Computing. He has served on a large number of technical program committees of international conferences, such as DATE, DFT, ESTIMedia, CCNC, MobiHealth, and others, and guest editor for special issues in journals such as JPDC, FGCS, MONET, Sensors, Supercomputing, etc. He is the author of more than 150 peer-reviewed publications, and a senior member of the IEEE.