Journal of Advanced Computer Science and Technology, 1 (4) (2012) 187-194
©Science Publishing Corporation
www.sciencepubco.com/index.php/JACST

Introducing Software-Based Fault Handling Mechanism to Cope with Electromagnetic Interference (EMI) in Digital Electronic Circuits

[1]Jinadu Olayinka, [2*]Arobieke Oluwole, [3]Kayode Idowu, [4]Osafehiniti Samuel

[1]Department of Computer Science, Rufus Giwa polytechnic, Owo
[2,3,4]Dept of Electrical Electronics Engineering Technology, Rufus Giwa Polytechnic, Owo
* E-mail of the corresponding author: oluarobieke@yahoo.com

Abstract

Digital circuits operating under radiation are subject to different kinds of permanent and transient effects. Most electromagnetic (EM) environment in which electronic systems have to operate is becoming increasingly hostile while dependence on electronics is widespread and increasing. The need for digital architectures to survive faults and remain dependable despite the multiple-fault injection nature of the electromagnetic interference (EMI) in microprocessors calls for the introduction of a software-based fault handling mechanism. Redundancy, which is a common answer to increasing error-coverage in most safety-critical applications offers higher dependability but for most low-cost computer based systems (including Digital Signal Processors), another technique is implemented for effectiveness. This paper implements the duplicate j-instruction rule on high-level programming to detect faulty jumps. Code redundancy and consistency checks cover the fault to increase system reliability.

Keywords: *Digital Circuits, Digital Signal Processing, Redundancy, Fault Handling Mechanism, EMI.*

1 Introduction and Motivation

Digital technology, a study and development of devices that stores and manipulates numbers has realised numerous digital devices including those that translate words (text), voices (audio) and pictures (graphics) into numbers (digits) for the computer to process and those that equally translates these numbers (digits) back into pictures, voices or words as the situation requires. Digital audio, speech recognition, cable modem, radar, high-definition televisions etc are but a few of the modern computer and communication applications that relies solely on digital signal processing as corroborated in Keshab (2009). The much flexibility in digital signal processing arises from the fact that most operations are implemented by simple program (algorithm) unlike in analog systems, where a re-design of the hardware must be carried out.

Digital circuits are circuits designed to respond at input voltages at one or finite number of levels and, similarly, to produce output voltages at one or finite levels (McGrawHill, 2004). Digital circuits functions on a number of different logic gates. Shenoi (2006) affirmed that digital circuits are becoming more and more popular as technology requires the electronic devices used day-to-day to become smaller and smaller, making the items more readily accessible regardless of location or circumstance

Signal processors are processing units capable of processing discrete-time signals. A signal is any physical quantity that varies with time, space or other independent variables (Monsoon, 1999). Signals can be described mathematically as a function of one or more independent variables, therefore enabling digital processes such as filtering, measurements, generations and reconstructions to be effected via linear or quadratic computations. Chen and Jan (2005) asserted that most of the technology of this information age is based on the theory of digital signal processing.

Shenoi (2006) explained that with the Digital Signal Processors (DSPs), these operations are specified mathematically using algorithms, implemented either in hardware or software. The availability of these efficient algorithms for the various digital signal processes, coupled with rapid development in integrated circuit technology has spurred the development of powerful; special-purpose digital hardwares. Reabaudengo et al (1999) further explained that these systems are capable of performing even complex digital signal processing functions and tasks evident in EMI environments.

Fault tolerance is a technique of providing, by redundancy, services complying with the specification in spite of faults having occurred or occurring. Fault tolerance is a method that assumes that a system has unavoidable and undetectable faults. It makes provision for the system to operate currently even in the presence of fault(s). Laprie (1995) defines this concept as a measure of reliability.

In digital circuits, faults are physical defects occurring in components while errors are the manifestation of these defects. Most often, it is an incorrect behaviour caused by a fault. However, the presence of faults may not ensure an error but transient or permanent faults are categorised by their duration. Faults in digital circuits exist as a deviation of one or more logic variables from their design-specified values within the systems; fault handling is a method of using redundancy to enable recovery even without explicit error detection.

Redundancy is a common answer to the increasing demand of high dependability in most safety-critical applications but for most low-cost computer based systems, another technique is proposed for effectiveness. The society's reliance on computer automations for domestic and industrial applications required software fault-tolerance more necessarily than the hardware fault-tolerance (Gray and Siewiorekc, 1991). Having this same view, Chen and Jan (2005) asserted that digital systems can be implemented by combining digital hardware and software fault-tolerance, each of which performs its own set of specified operations, enabling the effectiveness of introducing a software-based fault handling mechanism into the EMI digital signal processing environments.

To provide software reliability, information redundancy is implemented to provide the technique of handling the errors and hardware voting method is also simulated to provide the software redundancy.

Though, on-line fault detection through hardware redundancy is a viable solution in many different applications, but it is not feasible where cost is a critical issue. Software fault-tolerance is the ability of software to detect and recover fault that is happening or has already happened as the software is running in order to provide service in accordance with the specifications (Lyu, 1995). Although, systems hardware redundancy is unacceptable in some of these critical applications as in EMI environment, but Chris (1998) and Murray et al (2000) both affirmed that software redundancy is a design modification that can possibly be introduced to improve the system fault-tolerance. This technique creates a redundant system, because the software-based solution is efficient when it is not isolated from its hardware fault-tolerance counterpart.

2 Motivational Objective

The need for exploiting the high-computing performance of state-of-the-art processors (including the Digital Signal Processors) coupled with cost containment, provides a strong motivation for introducing feasible alternatives to existing traditional solutions. This software-based fault handling technique provides a low-cost solution of enhancing the reliability of the computer systems without modifying the hardware.

Therefore, the aims of this work are to:

(i) Develop an algorithm suitable for fault handling mechanisms in Digital Electronic Circuits and

(ii) Implement the above using C programming language

3 Design Procedures

In order to estimate the correctness of the adopted techniques, some benchmarks such as (integer matrix multiplication, filtering, summation of floating point numbers and library functions) of C program were considered. For this research, code modification approach for a conditioned loop was implemented on a block of C program code. Tests between two variables a and b for jump calls and further computations was implemented using the Control Flow Graphing shown in Figures 1a and 1b.
Semantically, most of the Control Flow faults in the program was identified using the Software Error Detection (SED) method and data redundancy techniques was adopted (Benso, 1997). The strategy of source-code modification is the partitioning of the program code into basic blocks *BB* for the two different modifications to detect the transient bit-flip fault in memory location containing the data. This approach, according to Gray and Siewiorek (1991) is a form of electromagnetic interference, which introduces transient faults because the microprocessors use registers to store data.
The first modification corresponds to *duplicating* some or all of the program *variables* to introduce data redundancy and modifying all the operators to manage the introduced replica of the variable. The second aims at introducing *consistency checks* into the control flow to periodically verify the consistency between the two copies of each variable.

3.1 Definitions

A basic block *BB* is a sequence of consecutive instructions in which in the absence of faults, the control flow enters at the beginning and leaves at the end. It does not contain any instruction (such as *jump*, *branch* or *call instructions*) that may change the control flow except for the last one. High-level programming define specific structure to delimit the borders of each block, as evident in using the symbol "{" used in C programming (Bjarne, 1997).
Namjoo (1993) explained that a program can be represented by a graph having set of nodes and set of edges. Adopting this definition, the source code in Fig 1a is represented by a corresponding Control Flow Graph (CFG) indicated in Fig 1b. (Goloubeva et al, 2003).
From Fig.1b, the graph has a set of nodes V and a set of edges E, $P = \{V, E\}$, where $V = \{v_1, v_2, ..., v_i, ..., v_n\}$ and $E = \{e_1, e_2, ..., e_i, ..., e_m\}$. Each node v_i represents a basic block *BB* while each edge e_i represents the branch $br_{i,j}$ from v_i to v_j. Though, these edges $br_{i,j}$ are not necessarily explicit branch instructions, they represent jumps, subroutine calls and return instructions.
Any illegal and incorrect branch indicates a control flow error. This would be checked by code redundancy and consistence checks.

0	i = 0;
	while (i < n) {

1	If (a[i] < b [i])
2	x[i] = a [i];
3	Else x[i] = b[i];
4	i++; }
5	

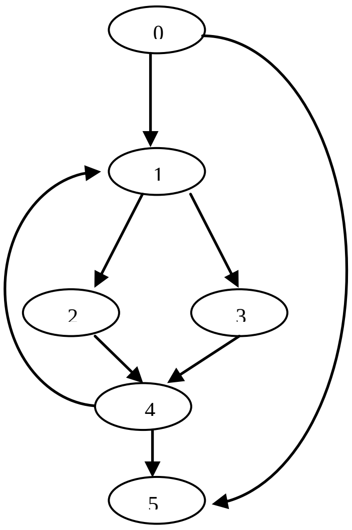

Fig 1a) Source code with basic block BB b) CFG for the source program segment

3.2 Code Transformation Rules

A *code* is a special program integer variable during any program execution (Goloubeva et al, 2003). To check the control flow, code redundancy is used by the implementation of two given assertions. At the beginning of the BB, the first (set) assertion, the code is updated to store the value corresponding to the currently transverse BB. The second (test) assertion then verifies the run time value of the code variable at the beginning if it corresponds to the value at the end. The consistency checks are performed before each code variable update by means of the set.

3.3 Methodology of using Hidden branches: Escaping faults

Considering the program CFG $P = \{V, E\}$, for each node v_i, the set of nodes successor of v_i is defined as $suc(v_i)$ while the set of nodes predecessor of v_i is defined as $pred(v_i)$. A node v_i belongs to $suc(v_i)$ if and only if $br_{i,j}$ is included in E. Similarly, v_j belongs to $pred(v_i)$ if and only if $br_{j,i}$ is included in E.

A code as a special program integer implements the masking operation. During program execution, the special program integer variable (called *code*) is updated through the set assertion defined above while the consistency check is performed before each *code* variable update is achieved by means of the test assertions. During the execution of P, $br_{i,j}$ is illegal if $br_{i,j}$ is not included in E. With a

legal branch $br_{i,j}$ included in E transformed in different branch $br_{i,k}$ included in E, $br_{i,k}$ is incorrect because these illegal and incorrect branches will indicate a Control Flow Errors, caused by transient or permanent faults.
The new value of the *code* variable is calculated from the old value of the *code* variable applying the *set* and *test* assertion rules described by the expression 1.1

$$code = (code \,\&\, M_1) \oplus M_2 \qquad ....(1.1)$$

where M_1 represents a constant mask depending on the identifiers of the nodes belonging to $pred(v_i)$; while M_2 represent a constant masks depending on the identifier of the current node and nodes belonging to $pred(v_i)$. The code fault is masked by an ORed combination of M_1 and M_2.
Choosing the identifiers of the *BB* to make the new value of the *code* variable equal to the targeted value is necessary to avoid an aliasing effect of the masking technique. This is achievable only if the old value of the *code* variable is legal according to the program CFG. For set assertion at the beginning and end of the BB, masks M_1 and M_2 is defined as 1.2 and 1.3 respectively.

$$M_2 = (I2_j \,\&\, M_1) \oplus I1_i \qquad (1.2)$$

and $$M_1 = 1; M_2 = I1_i \oplus I2_i \qquad (1.3)$$

When a $BB\ v_i$ is entered, the code variable is set to an integer variable value $I1_i$ and when the $BB\ v_i$ is exited, the code variable is set to another integer variable value *I2i*.
Test assertions introduced at the end of the $BB\ v_i$ is implemented as in expression 1.4

$$ERR\_CODE \mid= (code \mathrel{!=} I1_i) \qquad (1.4)$$

where the *ERR-CODE* is the control flow error. If the newly introduced test produce incorrect result, an error is detected as shown in Fig. 2 (source code) but if no error is signalled, the fault is masked.

```
code = B0;
ERR_CODE = 0;
i = 0;
ERR_CODE  |=  (code != B0);
code  =  code ^ (B0 ^ B1);
while  (i < n)  {
...
ERR_CODE |= (code = B4)  &&  (code  !=  B6);
code = (code & M1_4_6) ^ M2_4_6_7;
i1 = i;  i++;  i1++;
ERR_CODE  |=  (code != B7)  ||  (i  !=  i1);
code  = code  ^  (B7  ^ B8);
}
ERR_CODE  |=  ((code!= B1)  && (code !=  B8))  ||
```

Fig.2 Source code for masking fault using code redundancy technique

All illegal and incorrect branches indicate control flow errors, which are due to transient or permanent faults. The source code avoids the duplication of the whole set of variables therefore the life time of each variable can be determined.

4 Result and Discussions

The new code transformation rules suggested when applied produces an effect-less (EL) program output because the faults does not modify the results produced by the program. This technique identifies most (transient or permanent) faults leading to program control flow errors because there is fault free execution of the target application.

Calculated jumps, subroutines and other branches become efficient eliminating control flow errors. The source code modification is a variable duplication approach while code consistency check implements data redundancy. These techniques improve fault-tolerance in EMI environments in digital signal processing. An exhaustive application of these approaches dramatically increases system resources (CPU time and memory) overheads through masked fault-coverage. The fault-free behaviour therefore offers more dependable and reliable system.

Finally, to provide a balance trade-off between fault-coverage and resource overheads, there is the need to optimize the automatic modification of the source codes. The number of variables to be duplicated and the granularity (set and test assertions) of the consistency and code redundancy checks must be tuned. This is suggested for further research.

References

[1] Avizeinis S. A. (1995) “The N-version Approach to Fault Tolerant Software” IEEE Transactions of Software Engineering. Vol. SE-11 No 12 Pg. 1491-1501.

[2] Benso A., Corno F., Prinetto P.. Reubandengo M. And Sonza Reorda M. (1997). “FATO: a software Fault Tolerance approach”, IEEE International On-Line Testing Workshop.

[3] Bjarne Stroustrup (1997). “The C++ Programming Language. Third edition. Addison Wesley, Longman.

[4] Chen J. H. and Jan T. S. (2005) “A system dynamics model of the semiconductor industry development” Journal of the Operational Research Society, Vol 56 pp 1141-1150.

[5] Chris Incio (1998) “Software Fault Tolerance: Dependable Embedded System”. A paper presented at Carnegie Mellon University.

[6] Goloubeva O., Rebaudengo M., Sonza Rieorda M. and Violante M. (2003) “Soft-Error Detection using Control Flow Assertion” Proc. on Defects and Fault Tolerance in VLSI Systems Pg. 581-588.
[7] Gray J. and Siewiorek D.P. (1991) “High-Availability Computer Systems” 1st edition. IEEC Computer, USA.
[8] Keshab K. Parhi (2009). “VLSI Digital Signal Processing Systems: Design and Implementation”. 1st edition. Wiley Inc., USA.
[9] Laprie J. (1995). “Conceptual Framework for System Fault tolerance”. 1st edition. John Wiley Inc., USA.
[10] Lyu M.R. (1995). “Software Fault Tolerance” 1st edition Chichester, England, John Wiley & Sons Inc.
[11] Monsoon H.H. (1999). “Digital Signal Processing Schaum Outline Series” 1st edition McGraw-Hill Companies Inc. USA.
[12] Murray P, Fleming R, Harry P and Vickers P. (2000) “Somersault Software Fault Tolerance” HP Labs Whitepaper, California.
[13] Namjoo M. (1993) . “CERBERUS-16: An Architecture for a General Purpose Watchdog Processor”. Proc. Symposium on Fault Tolerant Computing. Pp 216-219.
[14] Rebaudengo M., Sonza Rieorda M., Torchiano M. and Violante M. (1999). “Soft-error Detection through Software Fault Tolerant Techniques”. DFT’99: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems. Pp 210-218.
[15] Shenoi B.A. (2006) “Introduction to Digital Signal Processing and Filter Design” 1st edition. John Wiley & Sons Inc.