A Single-Version Algorithmic Approach to Fault Tolerant Computing Using Static Redundancy

Goutam Kumar Saha

Centre for Development of Advanced Computing (CDAC), Kolkata, India
Contact: CA –2 / 4B, CPM Office Road, Baguiati, Deshbandhu Nagar,
Kolkata-700059, India. <sahagk@gmail.com >

Abstract

This paper describes a single-version algorithmic approach to design in fault tolerant computing in various computing systems by using static redundancy in order to mask transient bit errors in processor-memory and registers. This low-cost single-version scheme relies on a time redundancy approach. This software scheme does not intend to tolerate software design bugs. Instead of using multiple and independent versions of an application program, this single-version approach uses multiple copies of an application program. This low-cost approach is useful to tolerate various malicious code modifications and transient-faults during the run time of a computing application system without incurring any additional cost for extra hardware and extra software versions as an N-version programming scheme (NVP) or a Recovery block scheme (RBS). This proposed model is a practical and usable one that demands an affordable redundancy in time and space. The proposed scheme is capable of tolerating various operational faults that might occur during the execution time of an application.
Keywords: Fault masking, Computing Security, Transient Fault Tolerance, Time Redundancy, Single Version Programming (SVP)

1. INTRODUCTION

This paper describes how to design a single-version algorithm toward attaining low-cost reliable computing software for various application systems, by incorporating a single-version fault tolerant scheme along with code-integrity checking. Most of the ordinary systems lack fault tolerant software fix because the conventional fault tolerant approaches viz., Recovery Block (RB), N Version Programming (NVP) etc., are too costly to fix in an ordinary low-cost application system.

This proposed scheme is a non- fail-stop kind fault tolerance scheme that can be implemented in various computing systems without spending an additional money, and as a result, major part of common people in our society, can gain reliable service from the low – cost time redundancy-based computing system. Many of us in our society cannot always afford to buy a costly - computing system. A costly-system is expected to be a reliable one because of its built in redundancy in its various components. Many commodity systems use off- the- shelf - microprocessor or micro-controller that may lack ECC scheme. Electrical surges, transients, alpha particles or cosmic rays etc., often cause multiple bit errors in a memory or in a processor register [1, 2, 3]. As a result, an application fails often. The vast majority of hardware - failures in modern microprocessors (MP), especially for memory faults (for example, multiple byte errors or random bit-errors), is because of the limited hardware detection in them. Though, memory has Forward Error Correction (FEC) or Error Correcting Codes (ECC) (e.g. Parity bits, Hamming Code, BCH, and Cyclic redundancy codes in which bits are interpreted as coefficients in a polynomial etc.) that are capable of detecting and correcting a few bit errors on using both code and high time redundancy. For example, BCH (63, 45) can correct only 3 errors in a 45 information bits. CRC - 32 codes detects any single - bit, all double - bit, any odd number of errors, and error bursts of 32 bits errors. In general, CRC can detect burst errors up to length < number of redundancy bits. However, CRC (polynomial codes) take high processing time to calculate some function y = f(m), where m is the message data, for coding and decoding. Again, in CRC, there is a chance to have false negative test for error. Though CRC is more complicated than parity or checksum (that is, computing the sum

of all words in the application memory space before the application starts and re-compute the sum to validate with the earlier sum), it can be implemented in hardware. Checksum or such Error Correcting Codes (ECC) or Error Detection Mechanism (EDM) in the memory or in a processor, are useful for detecting and correcting a few bit errors only in memory. Software implemented ECC is not effective for online detection and correction of all bit errors in memory, but they are effective for a single or few bit flips in memory. Transient faults (whose presence is bounded in time) are random events. Transient bit errors can be tolerated by re-computing an application afresh. A permanent fault is one that continues to exist until the faulty component is repaired. Software Fault Tolerance is the reliance on "Design Redundancy" to mask residual design faults present in software program. Current fault tolerant techniques utilized in commercial systems such as IBM S/390 G5 in [4, 5, 6] rely on redundancies. For example, duplicating chips and comparing results implement error checking. These techniques need two times or more hardware overhead. In addition, the duplicate and compare is adequate for error detection only. Hence, low-cost fault tolerant technique is necessary for future microprocessor systems. This paper describes an economically very important method to tolerate multiple bit-faults, permanent and transient bit errors by acting on software only. The proposed Single-Version (SV) scheme is based on a procedure or application triplication, and comparison of the outputs of two copies for errors detection, and in case an error is detected, then it is followed by voting upon the outputs of all three copies that get executed sequentially in order to tolerate one fault, and to produce a correct output (that is, the output in majority). Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. We should accept that, relying on software techniques for obtaining dependability means accepting some overhead in terms of increased size of code and reduced performance (or slower execution).

Again, designing and implementing to allow correct functioning in the face of attacks is fault tolerance. An attack is a deliberate fault (the mechanical or algorithmic cause of an error in [4] that disrupts service, causing errors. Designing and implementing to prevent attacks is simply fault avoidance. This proposed work is concerned with how we can keep a system secure when part of it is corrupted during an attack in the context of denial of service. This work is not concerned with the preventing of attacks or fault avoidance. The aim of this work is to gain a survival computer system that continues to operate even after one or more of its components fail. This paper delineates a fault tolerant computing (through code redundancy) in support of information survivability to provide access to information in the face of attacks on integrity and availability. Any computer-based system has both real and theoretical weaknesses. The computing security aims to devise ways to prevent or to avoid various weaknesses from being exploited. Confidentiality, integrity, and availability are the three prime aspects of a computing system. Confidentiality is to ensure that only authorized ones access computing related assets of hardware, software, and data. It is also called secrecy or privacy. Integrity is to mean that assets can be modified only by authorized parties or only in authorized ways. Availability is to mean that assets are accessible to authorized parties at appropriate times. Computing assets of hardware, software, and data are often vulnerable to various types of vulnerabilities e.g., interruption, interception, fabrication, and modification. These three assets and the connections among them are all potential security weak points. Software may be changed, replaced or destroyed maliciously in [7], or modified, deleted, or misplaced accidentally. Whether intentional or not, these attacks often exploit the software's vulnerabilities. Sometimes the attacks are certain as when the software no longer runs. More subtle are attacks in which the software has been modified but seems to run without any abnormality. Again software is vulnerable to malicious alterations that either cause it to fail or cause it to perform an unintended task. Indeed, because software is so susceptible to "off by one" errors, it is quite easy to modify. Changing a bit or two can convert a working program into a failing one. Depending on which bit was changed, the program may crash when it begins, or it may execute for some time before it falters. The alteration can be much more subtle so that the program works well most of the time but fails in specialized circumstances. For example, the program may be maliciously modified to fail when certain conditions are met or when a certain date or time is reached. Because of this delayed effect, such a program is often called as a logic bomb. Again, data are especially vulnerable to malicious modification. Small and skillfully done alterations may not be detected ordinarily. On the other hand, faults can be classified as transient or permanent. A transient fault will eventually disappear without any apparent intervention, whereas a permanent one will remain unless it is removed by some external agency. While it may seem that permanent faults are more severe, from an engineering perspective, but they are much easier to diagnose and handle. The intermittent transient faults that recur often unpredictably are the most problematic. Availability is defined here in terms of providing fault tolerance to running applications and enhancing resources for future computing applications. Fault Tolerance is the ability of a system to perform its function correctly even in the presence of internal faults.

The software based fault tolerance approaches in [8] use protective code redundancy. This paper describes a novel software technique to validate the integrity of the application program and data codes that are often vulnerable to malicious code modification in [7, 8, 9, 10] or to transient bit – errors. The proposed technique is useful to prevent an application's failure because of its maliciously in [10] modified codes or transient affected codes. On detecting such modifications or transient faults in the application program and data codes, the proposed software based approach enables the program control to exit the faulty program and then to switch to other image of the application program and data. The aim of this approach is to detect and tolerate the transient faults, or malicious modifications or attacks on integrity that occurred in program and data. In other words, the proposed software based approach is an effective and low cost solution towards establishing fault tolerance and high computing security at an application system, without incurring any additional cost for additional hardware and for developing multiple versions of an application program. The overhead with both time and space here is not at all an unaffordable one. This technique uses moderate time and space redundancy (of the order of 3), to tolerate its various faulty codes that are caused either by intended modifications therein or by potential transients, and, thus to establish *fault tolerance* and *computing security* in various computing systems without any additional cost. Interested readers may refer to other related works in [11, 12, 13].

2. CONVENTIONAL APPROACHES

Both the RB and NVP rely on software design diversification and multiple machines. In other words, these schemes rely on multiple versions of an application running on different machines. In Recovery Blocks, the acceptance test condition is expected to be met by the successful execution of either the primary module or the alternate (different version) modules. When an acceptance test detects a primary module's failure, an alternate module executes. If all alternate modules are exhausted, the system crashes. In NVP, N number of variants (different versions) or alternates run simultaneously on N different machines and at the end of program, the results are voted upon to find an answer in majority and it is considered as a correct result. If no consensus result is found, then the NVP system crashes. However, both RB and NVP need multiple versions of software to be developed independently using different languages, tools etc. In reality, designing one version of reliable software is itself a very costly and challenging task. Again, designing multiple versions of software is found to be very expensive and beyond reach for many low cost applications. The RB scheme needs $f+1$ number of alternates to tolerate f sequential faults. The NVP scheme needs $f+2$ number of alternates to tolerate f sequential faults. The various *single-version software* implemented fault tolerance *(SIFT)* schemes, for example, Algorithm Based Fault Tolerance *(ABFT)* in Huang *et al.* (1984), is meant for supplementing the intrinsic *error detection mechanisms (EDM)* of a microprocessor system only for designing fail-stop (that is, stopping an application on detection of error) kind of fault tolerance against the fault model of transient bit errors in memory. *ABFT* is suited for applications using regular structures. Its applicability is valid for a limited set of problems. Therefore, it lacks of generality. The use of logic statements or assertions at different points in the program that reflect invariant relationships between the variables of the program can lead to different problems. Because, assertions are not transparent to the programmer and their effectiveness largely depends on the nature of an application and on the ability of a programmer. In *procedure duplication (PD)* in [14], *a programmer decides to duplicate critical procedures* and to compare the obtained results for detection of transient bit - errors. Here, *a programmer has to define a set of procedures to be duplicated* and to introduce the proper checks on the results. So, PD approach is useful to detect a few bit errors only, towards fail-stop fault tolerance through re-starting an application. These *SIFT* techniques that basically rely on a set of carefully chosen software detection techniques, *aim towards detection of few bit - errors in memory towards fail-stop kind of fault tolerance* through system reset and they lack of generality and applicability. Checksum based fault detection and tolerance has been discussed in the work [15]. Interested readers should refer to other important works on hardware or software implementations of time-constrained and reliable embedded systems also. Software cost analysis for RB, NVP and SIFT approaches have been discussed in [16].

3. THE SINGLE-VERSION ALGORITHM

This approach employs an application program and data in triplicate along with a popular EDM namely checksum. Checksum is employed to detect bit- errors in the program code itself. Application in triplicate

is to mask an operational fault during the run-time of an application. Three images (say, I^1, I^2 and I^3) of an application program and data code have been used. The block diagram of the proposed SV approach is shown in Fig. 1. The algorithm has been described in the following steps describe the functioning of this novel approach. The proposed approach does not intend to correct errors. This aims to mask various errors (intended or unintended) in order to survive and tolerate various faults for establishing higher system safety. We utilize the popular checksum to detect errors in codes that got induced prior to the execution of the application code. The proposed software approach aims to detect and to tolerate various operational or run-time errors. We have developed an application that is based on common input and sequential execution of at least two programs and comparison of results.

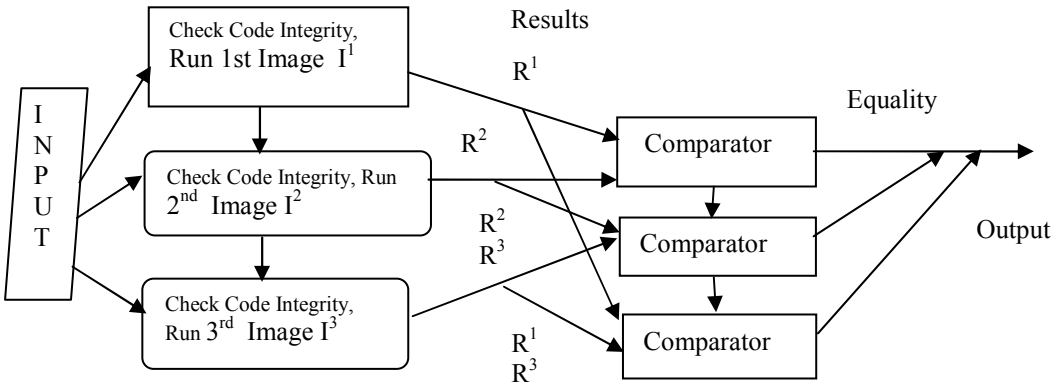

Figure 1: Block Diagram of a Single-Version Approach

Again, the comparison of the outputs (held on global variables namely R_1, R_2, R_3) on executing with similar inputs is to mask errors that might have induced (after checksum-validation) during the run time of an application code. In other words, those faults that are fail – silent during the checksum run, are detected by comparing the outputs on sequential executing three images of an application system with similar inputs. The symbols "/* */" are to include the comments or remarks. The *algorithm-steps* involved in SVP are stated below.

Step 0: Initialize the global output variables R_1, R_2, R_3 to 0.
/* Check the integrity of the first image */
Step 1: Compute checksum on I^1
/* Try the first image of the application */
Step 2: Compare the computed checksum with the pre-computed one.
Step 3: *If* a *mismatch*, Then: *Branch* to *Step 4.* /* checksum-mismatch */
/* Malicious code modification or transient faults */
Else: Execute I^1 /* save this result at memory word R_1 (global) */
End If
/*Check the integrity of the second image */
Step 4: Compute checksum on I^2
/* Try the second image of the application */
Step 5: Compare the computed checksum with the pre-computed one.
Step 6: *If* a *mismatch*, Then: *Branch* to *Step 7.*
/* Skip execution of second copy because of checksum mismatch indicating an attack or malicious code modification or transient faults in I^2 */
Else: *Execute* I^2
/* Otherwise execute the second redundant application code with similar inputs*/
/* Save this result at memory word R_2 (global) */

If $R_1 = R_2$ Then: /* Compare the result from I^1 and I^2 */
Output R_1 /* Output the correct result */
Exit
End If
End If
/* Check the integrity of the third image */
Step 7: Compute checksum on I^3
/* Try the third image of the application */
Step 8: Compare the computed checksum with the pre-computed one.
Step 9: *If* a *mismatch*, Then: *Branch* to *Step 10.* /* skip executing I^3 */
Else: Execute I^3
/* execute the third image with similar inputs */
/*save this result at memory word (global) R_3 */
End If
Step 10: *If* ($R_1 = R_2 = R_3 = 0$) , Then: *Error.*
/* All these results are zero */
/* No application image is executed because of checksum errors in all the redundant application codes. System crashes because all three redundant application codes are corrupted */
/* Checksum-mismatch, so branch to an error routine to repair or to reload and to re-execute the application for fail-stop tolerance */
Else If ($R_1 = R_2$!= R_3) , Then: $R = R_1$
/* Erroneous output R_3 of the third image is tolerated or masked, and the final result based on comparing upon the results from all images (to find a majority one) is stored in global memory variable *R* which is the final output of an application */
/* checksum or fail-silent faults or application run time malicious alterations or transient faults are detected and tolerated */
Else If ($R_1 = R_3$!= R_2) , Then: $R = R_3$
/* faulty output R_2 of the second image is tolerated or masked, and the final result based on comparing (to find a majority one) is stored in global memory variable *R* */
Else If ($R_2 = R_3$!= R_1) , Then: $R = R_2$
/* faulty output R_1 of the first image is tolerated or masked, and the final result based on comparison (to find a majority one), is stored in global memory variable *R* */
Else: Error
/* No majority in outputs is found i.e., faults occurred during the run time of these three images of the application system- an indication of system crash, so branch to error routine for application repair or to restart */
/* Secured Computation */
End If
{End of the algorithmic steps}

4. DISCUSSION & RESULTS

This single-version software-based scheme uses three copies or images of an application program. Thus it does not use design diversification to mask software design bugs. The work is concerned with tolerating transient faults and malicious code modifications. Before attempting to execute an image, it verifies for the code integrity by checksum method. If code is faulty then program control goes to verify the code integrity of the next image by checksum. If an image's code is error free then program control executes it. If an image is not executed then its corresponding output remains zero. However, if an image is executed, then its corresponding output is stored in its output variable. Again, all the images' non-zero outputs are compared upon to get an output in majority (also considered as the final output of an application system). An erroneous output (with zero values) does not take part at such comparisons. The errors that get induced after the checksum verification and during the execution of an image are masked by such comparator module. In other words, checksum detects errors that occurred before and after the execution of an image, and voting is to mask or eliminate an erroneous result that might have occurred during the execution of an image of an

application. The proposed novel approach (with three images) tolerates one fault. In general, this approach is capable of tolerating (N-2) faults on using N images of an application. For an application of multiplication of two matrices composed of 10x10 integer values, the source code size grows here by 3.31 whereas the executable code size increases by 1.90 and performance slows down by a factor of 3.35 only. Faults are randomly generated. It is observed that out of total 2000 injected faults, this software approach detects 854 faults, whereas the hardware based error detection mechanism (EDM) detects 503 errors. Again, from Bayesian network analysis, it is observed that the worst case reliability which is defined as *(3Tc + 2.25Ta + Tv)/ 2(3Tc + 2Ta+ Tv),* where *Tc* is the time for checksum verification, *Ta* is the application's execution time, and *Tv* is the execution time of the voting module, is *0.54.* Thus the reliability ($R(t) = e^{-\lambda t}$, where λ is the failure rate) of this proposed approach varies from *0.54 to 1.0.* Average case it is *0.87.* Thus, the reliability as well as the availability of an application based on this proposed approach is increased significantly. Unlike, N versions software approach this proposed approach does not use hardware redundancy and the various software design redundancy. Thus, this is a low cost solution towards establishing fault tolerance and computing security as well.

4.1 Comparison of Overheads

The major drawback of error detection and fault tolerance by software means come from the increase in execution time and the memory area overhead. On studying over a simple program of Bubble sort of 120 integer values, the overhead factors are listed below in Table 1. It is observed that single- version software scheme leads to a better performance.

Table 1: Overhead Comparison

Program Approach	Time Overhead	Memory Overhead
CRC-Non-Distribute	**>10.2**	**< 2**
Hamming	**>10.1**	**< 3**
Single-Version Algorithm	**> 3.1 and < 3.5**	**< 3.4**
Triple Modular Redundancy or RBS or an NVP using a Uniprocessor system	**> 3 and < 3.4**	**< 3.25**

5. CONCLUSION

The overhead with time and space redundancy here is not at all an unaffordable one. Instead of using fail – stop kind of fault tolerance, we have masked the various intended or unintended faults on employing multiple images of an application program. We must need code redundancy for establishing software based computing security and fault tolerance. At such case however, we need to spare higher space and time redundancy. The proposed approach is not intended to mask software design bugs. It is assumed that software code is correct. Here, the redundancy with space and time is of the order of three only. However, for faster execution of this approach, we can employ a multiprocessor system for parallel execution of these images at lock-steps. This is a significant step forward towards detecting and tolerating various transient faults and

malicious code alterations at a very low cost. This is also an effective tool for designing various safety critical computer controlled systems. The cost ratio i.e., (Cost of fault-tolerant software / non-fault-tolerant software) is 2.71 and 2.96 for a three-variants NVP and RB schemes respectively. However, the SV's cost ratio (with three copies) is only 1.22. Thus, this is also very useful tool for gaining high computing security and added high system safety without using any extra hardware and extra versions of an application system. Availability of an application system is also increased. This single-version algorithmic approach is a useful and simply implement able tool and this is based on static redundancy.

References

[1] Saha, G. K. Software implemented fault tolerance through data error recovery, *ACM Ubiquity,* Vol. 6, No. 35, ACM Press, USA, 2005.
[2] Spainhower, L. and Gregg, T.A. IBM S/390 parallel enterprise server G5 fault tolerance: a historical perspective, *IBM Journal of Research,* Vol. 43, No. 5/6, 1999.
[3] Sato, T. Analyzing overhead of reissued instructions on data speculative processors, In Proceedings of *Workshop on Performance Analysis and its Impact on Design,* held in conjunction with 25th International Symposium on Computer Architecture, 1998.
[4] B. Randell, B., Lee, A.P. and Treleaven, C.P. Reliability issues in computing system design, *ACM Computing Surveys,* Vol. 10, No. 2, 1978.
[5] Avizienis, A. The n-version approach to fault –tolerant systems, *IEEE Transactions on Software Engineering,* Vol. 11, No. 2, pp. 1491-1501, 1985.
[6] Huang, K.H. and Abraham, J.A. Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers,* Vol. 33, No. 3, pp. 518-528, 1984.
[7] Pfleeger, C. The fundamentals of information security, *IEEE Software,* Vol. 14, No. 1, USA, 1997.
[8] Saha, G. K. Low-cost fault tolerance applications, *IEEE Potentials,* Vol. 24, No. 4, IEEE Press, USA, 2005.
[9] Saha, G. K. Software based computing security & fault tolerance, *ACM Ubiquity*, Vol. 5 No. 15, ACM press, USA, 2004.
[10] Parker, D. and Nycum, S. Computer crime, *Communications of the ACM,* Vol. 27, No. 4, USA, 1984.
[11] Bellare, M. Practice – oriented provable – security, In *Proceedings of 1998 1st International Workshop on Information Security,* Springer, Berlin, 1998.
[12] Saha, G.K. Software based fault tolerant array, *IEEE Potentials,* IEEE Press, Vol. 25, No. 1, pp. 41-45, USA, 2006.
[13] Saha, G. K. Software based fault tolerant computing, *ACM Ubiquity,* Vol. 6, No. 40, ACM Press, USA, 2005.
[14] Pradhan, D.K. *Fault - Tolerant Computer System Design*, Prentice Hall, 1996.
[15] Wicker, S. B. *Error control systems for digital communication and storage,* Prentice Hall, NJ, USA, pp.72- 127, 1995.
[16] Laprie, J.C. Arlat, J. Beounes, C. and Kanoun, K. Definition and Analysis of Hardware and Software Fault Tolerant Architectures, *IEEE Computer,* Vol. 23, No. 7, pp. 39-51, 1990.

Author's Biography

In his last nineteen years' research and teaching experience, he has worked as a scientist in LRDE, Defence Research & Development Organization, Bangalore, and at Electronics Research & Development Centre of India, Calcutta. At present, he is with the Center for Development of Advanced Computing, Kolkata, India, as a Scientist-F. He has authored more than one hundred research papers at various national and international journals, magazines and proceedings etc. He is a senior member in IEEE, Computer Society of India, ACM and Fellow in IETE, IMS, UWA and MSPI and member of the W3C Internationalization Tag Set Working Group. He has received various awards, scholarships and grants from national and international organizations. He is a referee of the CSI Journal, AMSE Journal, an IEEE magazine and IJCPOL etc. He is an associate editor of the ACM Ubiquity and a member in the Editorial Board of the IJCIS (Canada). He has received the "ACM Recognition of Service Award" in December 2005 for Contributing Author to ACM. He has also received the "IEEE Member Recognition" citation in July 2006. His field of interest is on software based fault tolerance, dependable computing, robust web application and natural language engineering. He can be reached via gksaha@ieee.org or via <sahagk@gmail.com>.