Software Based Fault Tolerance Against Byzantine Failures

Goutam Kumar Saha

Centre for Development of Advanced Computing, Kolkata, India.
Mail to: CA – 2 / 4 B, Baguiati, Deshbandhu Nagar, Kolkata - 700059,
West Bengal, India. gksaha@rediffmail.com, sahagk@gmail.com

Abstract

The proposed software technique is a very low cost and an effective solution towards designing Byzantine fault tolerant computing application systems that are not so safety critical. It does not rely on multiple versions of software running simultaneously on multiple machines. The proposed software approach is to mask various hardware random errors on adopting the so-called, ESVP (an enhanced single-version program) scheme, while an application is being executed. It is not intended to eliminate software design bugs. In other words, it is assumed that code is correct and the faulty behavior is only due to transient or Byzantine faults affecting an application system. Implementation of this approach is also easy. A test program's present state is compared with its pre-computed state also in order to detect state transition - fault also. ESVP is intended to be suitable for a computer-based process monitoring system.

Keywords: Transient Fault, Byzantine Failures, Error Masking, Hardware Fault Tolerance, ESVP.

1. INTRODUCTION

The demand for more dependable software or software fault tolerance is increasing rapidly as our high – tech society becomes increasingly dependent on computers. We can say a processor system has a Byzantine failure when an incorrect and odd answer results from a computation. A Byzantine failure [1] represents the most difficult type of fault caused by faulty processing system. Clearly, a system that can protect itself from Byzantine faults is impervious to any kind of fault. In a distributed environment, this may not necessarily be due to a processor, but rather the medium of communication. If a processor returns an incorrect result, there is not any hardware to reassure us that the correct result was computed. When a processor system returns wrong result because of a bug, or due to a malicious design, — an incorrect and inconsistent computation simply is classified as a Byzantine failure. A message sent over a network often becomes corrupted in some manner. Though Byzantine failures do not occur often but the consequences of incorrect results are very disastrous in our real life applications. Nobody wants erroneous results in his or her computing system. A fault – tolerant system is one that can sustain a reasonable number of process or communication failures of both permanent and intermittent nature. Developing software for tolerating Byzantine faults is obviously a very challenging task. The objective of this paper is to demonstrate the low-cost design technique for gaining Byzantine fault tolerance in computing system without using an extra processing system. It is assumed that the application program code is correct and has no software design bugs. It is assumed that the conventional error detection and correcting codes based on hardware implementations will complement the proposed approach. Software implementations of such error codes (e.g., Hamming codes, CRC etc.), have significant overhead on both run-time and memory space. It is assumed that an application program has been rigorously tested and the program has no software design bugs. The proposed enhanced single-version program (ESVP) based technique (e.g., with three

copies of an application program that executes one after another with its similar but own input data set along with a test program execution with its known data) detects the Byzantine-faults and hardware transient faults as early as possible. And, it can initiate the necessary recovery actions to repair the erroneous code (for availability at next run) immediately for preventing any disastrous consequence or, further damage of the computing system. However, such recovery is optional here. At the same time, it provides correct result through masking the erroneous computed answers. It is our common belief that when systems are not so safety critical, it is difficult to suggest the conventional expensive techniques that rely on triple modular redundancy (TMR) for tolerating one fault.

An ESVP with three images tolerates one fault. In case there are two or more faults then such system crashes. Though the ESVP applies time redundancy in its simplest form, it is intended to be useful as a low-cost solution to design a dependable system (e.g., a commodity system) that is not so safety critical. The proposed ESVP scheme is intended to get fault tolerance using ordinary computer. The paper is not intended to eliminate software design bugs. It is intended to mask various hardware faults including transient and Byzantine faults in the process, processor registers or in memory etc., during the run-time of an application for gaining a low-cost, fault tolerant and dependable computing using an ordinary off-the-shelf computing machine. It is assumed that hardware error detection mechanisms (e.g. microprocessor exceptions) will control the non mask-able errors that may occur during execution time of an ESVP application. We adopted three copies of an application program along with a test program and a test data. All the copies of an application program run (with similar but their own set of input data in order to mask errors in their input data) in a sequence one after another and, the results collected from several executions are compared (or voted upon) to mask potential errors. This scheme has been tested on an application of Boiler Turbine Efficiency Computation for the Kolaghat Thermal Power Station in India. Efficiency computation is carried out using various online sensor data on buffer and displayed for further monitoring works by the plant engineers. It is intended to produce reliable result so that the engineers can take appropriate action based on the reliable efficiency result. It has been manually verified that the computation is very reliable over about last six months. It has been reported that 1% -2% errors have been observed because of communication or sensor hardware errors. Like any other software based fault tolerance technique, the time redundancy here is also a significant one (2.65 times). We believe that a faster microprocessor will be good enough to cope up such time redundancy for computing a reliable answer for an application (based on ESVP) in accordance to an asking response time. Space redundancy here, (2.75 times for the executable code) is not a significant problem nowadays because memory is also becoming cheaper day by day like other hardware. We examined the ESVP scheme also with few C programs as a benchmark.

The conventional Recovery Block Scheme (RBS) and N- Version Program (NVP) scheme depend on multiple versions of software and hardware. That is why they are costlier than the ESVP. Such schemes are to mask software design bugs. Again, the algorithm based fault tolerance (ABFT) and the control flow checking (CFC) schemes (based on one copy with robust data structure) lack of generality and their universal applicability. Whereas, the ESVP inherits its own generality, wide-applicability and simplicity.

Both the ABFT [2, 11, 12, 13] and CFC [3, 12, 13] are intended to detect errors only and, they are intended to complement the intrinsic hardware error detection mechanisms (e.g., microprocessor exceptions) in a microprocessor system. The so-called ESVP also inherits similar time and space redundancy as that of with the conventional fault tolerance schemes: RBS [4], ABFT and CFC. The NVP [4] also suffers from high synchronization overhead because N versions of an application run concurrently on three different machines (to tolerate one fault) followed by voting upon the answers for an agreement. Similar to NVP, the ESVP also uses comparison code for obtaining an answer in majority. Unlike the NVP and RBS, the proposed ESVP does not rely on multiple versions of software and hardware. Rather, it relies on single reliable version of software and hardware only.

2. WORK - DESCRIPTION

The proposed software technique does not need multiple processors or multiple software versions of the application. Rather it needs only one version of reliable application software that needs to be executed on single processor system. We need three copies (e.g., A_1 , A_2 , and A_3) of the same version of an application program. One sample test program (TP) is executed on similar processor with known inputs and the computed result is stored in an output data file namely, *STPR* for future reference during the run-time of an

application based on ESVP. Now, this enhanced technique contains only three copies or replicas of the application program i.e., A_1, A_2, A_3 , and the test program's pre-computed output STPR data file, along with the test program TP and its known input data only. Now, the application copies are executed one after another with similar inputs and, the corresponding results or states are said, R_1, R_2, R_3 from A_1, A_2, A_3 respectively at one run. In some cases where program relies on timing of certain events such as external interrupts, or rely on reading things such as the system time, these inputs need to be buffered. A built-in *test program* is also executed (after the execution of A_1, A_2, A_3) on a known input and its state (*TPR*) is compared with the expected state (that is, a pre-computed and stored state *STPR*). The *test program's* (*TP*) result is say, *TPR* at the same run. Run time or observed result or state *TPR* needs to be compared with the expected result at *STPR*. The following steps explain how the proposed technique is capable of detecting and tolerating the Byzantine faults or hardware faults (during the run-time of an application) more efficiently, without using costly multiple hardware resources. Here, recovery procedure is an optional one. For single run of the ESVP based application, we do not need to execute the recovery routine. However, it is required to make the code available for the subsequent usage only.

The basic steps involved in an ESVP Scheme with three images are stated below.

```

If ( $R_1.NE. R_2.NE. R_3$ ) .AND. ( $TPR.NE. STPR$ ), Then:
    Display Byzantine Failures /* Incorrect computed results */
                                /* Application crashes */
    Call Recovery_Proc1 /* Reload application copies and the test
                        program- TP; restart the execution for gaining
                        the Fail – Stop kind of fault tolerance. */
Else If (( $R_1.EQ. R_2$ ) .AND. ( $R_2.NE. R_3$ ) .AND. ( $TPR.EQ. STPR$ )), Then:
    Go ahead with application result  $R_1$  or  $R_2$ 
/* The answer in majority is considered as a correct result. One fault is tolerated. */
    Call Recovery_Proc2 /* Recovery is optional at this stage.  $R_3$ , the result of  $A_3$ ,
                        is faulty. Recover  $A_3$  by copying from master copy at
                        a subsequent time. */

Else If (( $R_2.EQ. R_3$ ) .AND. ( $R_1.NE. R_2$ ) .AND. ( $TPR.EQ. STPR$ )), Then:
    Go ahead with application result  $R_2$  or  $R_3$ 
    /*Go ahead with the majority answer. One fault is tolerated. */
    Call Recovery_Proc3 /* Recovery is optional at this stage.  $R_1$ , the computed
                        result of  $A_1$ , is faulty. Recover  $A_1$  by say, copying
                        from master copy */

Else If (( $R_1.EQ. R_3$ ) .AND. ( $R_1.NE. R_2$ ) .AND. ( $TPR.EQ. STPR$ )), Then:
    Go ahead with application result  $R_3$ 
    Run Recovery_Proc4 /* Recovery is optional at this stage.  $R_2$ , the computed
                        result of  $A_2$ , is faulty. Recover  $A_2$  by say, copying
                        from master copy. One fault is tolerated. */
Else If (( $R_1.EQ. R_3$ ) .AND. ( $R_1.NE. R_2$ ) .AND. ( $TPR.NE. STPR$ )), Then:
    Go ahead with application result  $R_3$ 
    Run Recovery_Proc5 /* Recovery is optional at this stage.  $R_2$ , the
                        Computed result of  $A_2$ , is faulty. Recover faulty  $A_2$ 
                        and TP by say, copying from master copy. Two
                        faults are tolerated. */
Else If (( $R_1.EQ. R_3$ ) .AND. ( $R_1.EQ. R_2$ ) .AND. ( $TPR.NE. STPR$ )), Then:
    Go ahead with application result  $R_1$ 
    Run Recovery_Proc6 /* Recovery is optional at this stage. Recover
                        TP by say, copying from master copy. One fault is
                        tolerated. */

```

End If

3. DISCUSSION

The above steps are self – describing ones and very easy to understand also. Here, we compare the corresponding three outputs on executing the three copies of the application program (one after another) with similar but their own sets of input data (to tolerate errors in input data) and, select the output data that are agreed upon voting, as a correct answer of an application based on ESVP scheme. At an event when all the three outputs differ from each other (an instance with faults at two or three copies) then it might be a case of Byzantine failure that results in a system crash. Again, when the run – time result of the test program (*TPR*) does not match with the pre-stored test program's expected result (*STPR*) then also we should consider it as a Byzantine failure. At the most disastrous case (as stated at the first step of the algorithm), when all the outputs of *A1*, *A2*, *A3* of one application run, do not match to each other and, the run-time test program result or state *TPR* does not match to the expected one at *STPR*, then we need to recover the codes of *A1*, *A2*, *A3*, *TP* and *STPR* by reloading them (from a stable storage) in order to restart the execution of the application program for re-computation (e.g. fail-safe kind of fault tolerance).

For other related recovery works, interested readers may refer to various software techniques as discussed in the works [5,8,9,10]. However, in the event of single Byzantine-fault, we do not need to restart the application for re-computing because, we can carry on further with the correct computed result (i.e., the answer in majority) and then, the faulty program code is repaired or recovered by copying from the master copy on a stable storage for later usage. When a re-start is not wanted then the byte-by- byte recovery work can be carried out based on algorithm as discussed in the work [6]. This proposed approach is also very suitable for tolerating random errors in application code also.

In general, we need $(f + 2)$ replicas of the application program in order to tolerate f -number of hardware or memory faults. Based on *Bayesian* statistical approach, we get the probability of uncertainty with an application adopting this *ESVP* approach, of the ratio of $((3 * T_c + T_v) / 5 * (3 * T_c + T_v))$ or, 0.2 only, over an application-execution session. Where, T_c denotes the execution time for one copy or replica and, T_v denotes the time for comparing the answer. Here, we do not carry out the recovery work for a single fault at any copy of an application.

4. EXPERIMENTAL RESULTS

We have also tested a C-program on bubble sorting one hundred integers, as a benchmark, for evaluating the feasibility and effectiveness of this *ESVP* approach. We modified the source code (manually) of the following two programs according to the proposed scheme.

a) The approach has been executed for bubble sort algorithm with 12 integer elements. The source code size increases by 3.82. The executable code size increases with a factor of 2.72 and performance slows down by a factor of 2.6.

b) The approach has also been executed on matrix multiplication of two 8 x 8 integer values. The source code size increases by 3.69. The executable code size increases with a factor of 2.79 and performance slows down by a factor of 2.7.

We have used a *Motorola 68040 processor* and the *SDS Single Step 7.4* as an adopted compiler. An average slow-down (time redundancy) of about 2.65 times is observed.

We have adopted the *fault model* of single-bit flip into memory locations on an environment [7] built around an application board hosting a 25 MHz *Motorola 68040 processor*, 2 MB RAM and some peripheral devices. *Fault injection* has been performed exploiting an ad hoc hardware device that allows monitoring the application program execution and triggering a fault injection procedure when a given point is reached. We have injected 1000 bit-flips in either one of the application copy and we have observed that on an average 45 % bit- flip errors have been masked by this *ESVP* scheme. Hardware error detection mechanism (e.g., microprocessor exceptions) detects about 34 % errors. The rest 21 % errors have been classified as fail silent (causing no difference in the program behavior).

5. CONCLUSION

The proposed ESVP technique is a very low cost but an effective solution towards designing Byzantine fault - tolerant system. This is a generalized scheme that does not rely on conventional, costly multiple versions of both the application software and hardware. It is easy to implement also. We believe that the significant overhead on run-time and space is easily afforded if we use a faster, modern microprocessor in order to gain higher dependency in computing system at no extra cost for multiple versions of hardware and software. The ESVP approach does not lack of generality and applicability. In particular, system engineers might opt for this low cost ESVP scheme for various computer-based processes monitoring application systems. This approach has also been implemented and tested on a thermal power plant application and trusted as a reliable low-cost solution.

References

- [1] Gartner, F. C. Byzantine Failures and Security: Arbitrary is not (always) Random, *Technical Report IC/2003/20*, April 4, Swiss Federal Institute of Technology (EPFL), 2003.
- [2] Huang, K.H. and Abraham, J.A. Algorithm-Based Fault Tolerance for Matrix Operations, *IEEE Transactions on Computers*, 33: 518-528, 1984.
- [3] Yau, S. and Chen, F. An Approach to Concurrent Control Flow Checking, *IEEE Transactions on Software Engineering*, SE-6(2): 126-137.
- [4] Pradhan, D.K. *Fault - Tolerant Computer System Design*, Prentice Hall PTR, 1996.
- [5] Saha, G.K. Software Based EMI – Fault Tolerance in a PC – Peripheral, *International Journal – Cybernetics and Systems Analysis*, Vol. 32, No. 5, Plenum Press, USA, 1996.
- [6] Saha, G. K. Transient Software Fault Tolerance Through Recovery, *ACM Ubiquity*, 4(29), ACM Press, USA, September 16, 2003.
- [7] Benso, A. Civera, P.L., Rebaudengo, M. and Reorda, M.S. An Integrated HW and SW Fault Injection Environment For Real - Time Systems, In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp. 117-122, 1998.
- [8] Kalbarczyk, Z.T. and Iyer, R.K. et al., Chameleon: A Software Infrastructure for Adaptive Fault Tolerance, *IEEE Transactions on Parallel and Distributed Systems*, pp. 560-579, 1999.
- [9] Saha, G.K. Software Implemented Fault Tolerance Through Data Error Recovery, *ACM Ubiquity*, Vol. 6, No. 35, 2005.
- [10] Gupta, R.K., Coelho, C.N. and Micheli, G.D. Program Implementation Schemes for HW-SW Codesign, *IEEE Computer*, Jan 1994, pp. 48-55.
- [11] Saha, G.K. Software Based Fault Tolerance – a Survey, *ACM Ubiquity*, Vol. 7 No. 25, July 2006, USA.
- [12] Saha, G.K. Software Fault Avoidance Issues, *ACM Ubiquity*, Vol. 7, No. 46, December 2006, USA.
- [13] Saha, G.K. Software Based Transient - Fault Tolerance, *International Journal of Mathematics and Computer Science*, Vol. 1, No. 2, pp. 179-190, France.

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>.