

Transient Software Fault Tolerance Using Single-Version Algorithm

by Goutam Kumar Saha

Introduction

Today, *fault tolerance* is a much researched field in computer science. It means that a system can provide its specified services even in the presence of faults. A system *fails* when it cannot meet its promises. An *error* is a part of a system's state that may lead to a failure. Designing a *dependable system* closely relates to controlling faults. Intermittent faults may occur and then may vanish and then reappears, and so on. *Transient* faults occur once and then disappear. At the next operation, program may work without any fault. A *permanent* fault continues to exist till the faulty component is repaired.

Electrical short duration noises or *transients* often cause errors in program flow and data. Often transients generate unpredictable random bit-errors or *soft errors* in an application system. A *soft error* is one that can be recovered by reprogramming. Traditionally many techniques exist for checking the validity of data. Some of these are the use of a parity bit, checksums, cyclic redundancy checks, and use of error correcting codes. Most of these techniques detect the presence of an error but cannot correct. Error-correcting codes, implemented by hardware circuits, is capable of both detecting and correcting certain types of errors. However, they are not free from limitations. For example, by adding six extra bits per sixteen-bit word, single- or double-bit errors can be corrected. However, random multiple-bit errors are very common type errors caused by potential transients. Conventional hardware fixes cannot detect and correct multiple-bit errors.

Again, software implementations of such error-correcting codes use an unaffordable redundancy in both time and space.

The best-documented techniques for tolerating software faults are the recovery block (RB) [1] approach and N-version programming (NVP) [2]. They are based on design-diversifications. The different systems produced from a common service specification are called variants or alternates. A diversified design has at least two variants plus a decider or an acceptance test. Decider monitors the results of variant execution on given consistent initial conditions and inputs. In reality, designing a single-version software itself is very difficult task and costly one. Then thinking of multiple version software is nearly looks like a fantasy. *How large the value of N should be in NVP or RB to provide higher availability? What is the optimum value of N to assure us for true fault tolerance? What is the end value of N ?* We cannot avoid the threat of introducing common errors or introducing various new independent errors in case of a multi-version based system. Rather, we must concentrate on designing a single-version, which is a reliable one. It is something like in order to get uninterrupted telephone service; you need to possess telephone receivers of all available makes and brands, to receive or to make a phone call. Instead we may go for designing a reliable one with the capability of self-detection and recovery. Interested readers may refer to [3,4,5,6] for related works on single-version software fault tolerance through replicated process and data for higher availability and fault tolerance.

Work Description

In this paper, the author introduces a low-cost and unconventional technique for designing a single-version software. It is capable of self-detection and recovery. Instead

of using multiple versions of application, the author uses only one version of the application with the design in self-detection and recovery technique. Two images of both the application program and data have been used. For example, say, A_1 and A_2 be the two images of the application program, and, d_1 and d_2 be the two images of the application data residing on memory. Again say, $A_1 d_1$ denotes the computed result using the application image A_1 and data image d_1 , and so on.

The Scheme:

The noble scheme based on replicated process and data along with appropriate usage thereof have been stated below.

If $(A_1 d_1 == A_2 d_2)$, Then :

Continue with $A_1 d_1$ or $A_2 d_2$

/ A_1, A_2, d_1, d_2 are not faulty */*

Else If $(A_1 d_1 == A_2 d_1) \wedge (A_1 d_2 \neq A_2 d_2)$, Then :

Continue with $A_1 d_1$ or $A_2 d_1$

/ d_2 is faulty */*

Else If $(A_1 d_2 == A_2 d_2) \wedge (A_1 d_1 \neq A_2 d_1)$, Then :

Continue with $A_1 d_2$ or $A_2 d_2$

/ d_1 is faulty */*

Else If $(A_1 d_1 == A_1 d_2) \wedge (A_2 d_2 \neq A_2 d_1)$, Then :

Continue with $A_1 d_1$ or $A_1 d_2$

/ A_2 is faulty */*

Else If $(A_2 d_1 == A_2 d_2) \wedge (A_1 d_1 \neq A_1 d_2)$, Then :

Continue with $A_2 d_1$ or $A_2 d_2$

```
/* A1 is faulty */  
Else  
    /* A1, A2, d1, and d2 are faulty and Recover them */  
    Copy A1, A2, d1, and d2 from master file  
    Resume Execution of the application program. /*Go to the 1st step */  
End If
```

From the above steps, it is understood that we gain effectively four sets of application runs for example, by just using two images of the same version of the application program and data. In case transients corrupt both the images, we can scrub the memory or recover them from the master copy of the application and data, and then continue the execution of the application system. This technique bears the overhead over both the execution time and space redundancy of the order of two only.

Conclusion

This unconventional technique uses space redundancy of two only with a little extra and affordable time redundancy. It is a useful and low cost solution towards transient software fault tolerance. Whereas NVP or RB can not ensure high reliability against random errors induced at all the versions, caused by potential transients. Because when transients affect few or all the versions' codes randomly, then the application system based on NVP or RB techniques collapse completely, whatever may be the value of N. Readers may modify or tailor this approach to fit their application systems on distributed or mobile computing also. However, it demands a thorough study on the application system.

References

- [1] B. Randell, "Design – Fault Tolerance," in *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, and J.-C. Laprie, eds., Springer-Verlag, Vienna, 1987, pp. 251-270.
- [2] A. Avizienis, "The N-Version Approach to Fault-Tolerant Systems," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 12, Dec. 1985, pp. 1491-1501.
- [3] Goutam K Saha, "Transient Fault Tolerant Processing in a RF Application," *International Journal of System Analysis Modeling Simulation (SAMS)*, Gordon and Breach, 2000, Vol. 38, pp. 81-93.
- [4] Goutam Kumar Saha, "Beyond the Conventional Technique of Software Fault Tolerance," *ACM Ubiquity*, Vol. 4(47), 2004.
- [5] Goutam Kumar Saha, "Fault – Tolerance & Program Security - a Novel Approach," *Proceedings of The ICMS 2004*, AMSE Press, Spain, 2004.
- [6] Goutam Kumar Saha, "Transient Fault-Tolerance Through Algorithms," in press, *IEEE Potentials*, IEEE Press, USA, 2005.

Goutam Kumar Saha
Member ACM
CA –2 / 4 B, CPM Party Office Road, Baguiati, Deshbandhu Nagar, Kolkata – 700059,
West Bengal, INDIA
gksaha@rediffmail.com

Author Biography: Goutam Kumar Saha has been working for last seventeen years as a computer scientist in various renowned R&D organizations. He is presently working as a Scientist-F in the Centre for Development of Advanced Computing, CDAC, Kolkata,

India. His past employer is LRDE, Defence R&D Organization, Bangalore, India. He has authored many international research papers on Fault Tolerant Dependable Computing and NLP. He is a referee for IEEE Magazine, AMSE J (France), IJCPOL and CSI Journal. He is a fellow in IETE, senior member in CSI, IEEE and member in ACM. He can be reached via gksaha@rediffmail.com.

Source: Volume 6, Issue 28 (August 2 - August 9, 2005)
<http://www.acm.org/ubiquity/>