

Replicated Instruction Based Fault Tolerant Computing

Goutam Kumar Saha

sahagk@gmail.com gsaha@acm.org

In this discussion, we do not consider the issue of eliminating software bugs; rather, we assume that the code is correct and the faulty behavior is only due to transient faults affecting the system. The software fixes here aim to supplement the intrinsic error detection mechanism of a processor. Fault tolerance is the ability of an application system to perform its function correctly even in the presence of internal faults. Today, fault tolerance (FT) is a much researched subject. Essentially, a system fails when it cannot meet its promises. Transient faults occur once and then disappear. An intermittent fault occurs, then vanishes of its own accord, and then reappears and so on at irregular intervals. 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 various components. Many commodity systems use off-the-shelf microprocessors or micro-controllers that may lack Error Correcting Codes (ECC). Electrical surges, transients, alpha particles or cosmic rays etc., often cause multiple bit errors in a memory or in a processor register. As a result, an application fails often. The vast majority of hardware failures in modern microprocessors, especially for memory faults, are because of limited hardware detection in them.

Transient faults (whose presence is bounded in time) are random events. Transient bit errors can be tolerated by re-computing an application afresh. Designing low-cost fault tolerant technique is necessary for future microprocessor systems. The simplest way to detect an error in computation is to execute a code twice with similar input and then to compare both the outputs. An inequality in outputs indicates errors in a microprocessor system. Here, time redundancy is of the order of two. There are other approaches that intend to complement the intrinsic Error Detection Mechanisms (EDM) with a set of carefully chosen software error-detection techniques. Such techniques include Control Flow Checking (CFC), and Assertions. In *CFC*, the application program is partitioned in basic blocks and a deterministic signature is computed for each block and faults can be detected by comparing the run-time (or observed) signature with a pre-computed and stored (or expected) signature for the corresponding block. The main problem with CFC is to tune the test granularity that needs to be used. The basic idea of Assertions is to insert logic statements at different points in the program to reflect invariant relationships between the variables of the program. It may lead to different problems, since assertions are not transparent to the programmer and their effectiveness depends on the nature of the application and on the ability of the programmer.

Both the CFC and Assertions rely on application semantics. Readers may refer to the work that makes available different fault-tolerant configurations and maintains run-time adaptation to changes in the availability requirements

of an application. Other works describe various fault tolerance approaches (Recovery Block and N-version Programming Schemes, Triple Modular Redundancy etc.,) that use design diversity to tolerate software design bugs. Most of the conventional methods to detect faults in microprocessors are based on Parity bits, Cyclic Redundancy Checks and Check-sum or such Error Correcting Codes (ECC) in the memory or in a processor. ECC is useful for detecting and correcting a few bit errors only in memory. Current fault tolerant techniques utilized in commercial systems such as IBM S/390 G5 rely on redundancies. For example, duplicating chips and comparing results implement error checking. This work is intended to tolerate electrical short-duration noises by verifying the intermediate state of the program during its run time. Interested readers may refer to related works as further readings. Triplicate modules, as mentioned in Saha 2003, are used to detect faults as well as recovery thereof. The work described here is primarily based on the scheme as described in the works of M. Rebaudengo et al, 1999 and of Saha 2006.

The basic steps involved in Duplicated Instruction approach are as follows:

- Each processing instruction is duplicated including input & output data.
- Each of the replicated program instruction including initialization, assignment and processing instruction is executed sequentially one after another.
- After the execution of all the duplicated instruction blocks, the intermediate computation answers from both the duplicated instruction-blocks are voted upon for the similar one.

- If intermediate answers are same then processing continues further to next program block otherwise, an error flag is generated to produce an abnormal exit in order to void erroneous computation. If an error is detected, the program can be re-executed to tolerate transient fault. If no error is detected, the program continues further to produce a correct final result.

As a benchmark, a simple C-language program (as shown in Table-1) meant for finding an average of a set of real numbers is used in this paper. We used duplicated input, output and temporary variables along with duplicated program instructions and the code for comparison of intermediate and final results.

The work here does not rely on multiple (design) version of an application. It relies on a single version of program for an application. This is a low-cost solution for fault tolerance and can easily be implemented in our daily life application on affording an extra time and memory redundancy. The overhead on memory space is of the order of 2 here, whereas the execution time overhead is of the order of 1.6 only. Such overhead is an affordable only for designing a reliable computing application. This low-cost simpler approach is suitable for designing reliable commodity systems using ordinary available components only.

The interested reader may consult the following references:

1. D. Pham, "Software Fault Tolerance," IEEE Computer Society Press, 1996.
2. H. Hecht and M. Hecht, "Fault- Tolerance in Software, in Fault – Tolerant Computer System Design," D.K. Pradhan, Prentice hall, 1996.

3. M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, "Soft-error Detection through Software Fault-Tolerance Techniques," Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems, pp.210-218, 1999.
4. Goutam Kumar Saha, "A Low-Cost Testing for Transient Faults," ACM Ubiquity, Vol. 7(2), January, 2006, ACM Press, USA.
5. Goutam Kumar Saha, "Software Fault Tolerance Through Run – Time Fault Detection," ACM Ubiquity, Vol. 6(46), December, 2005, ACM Press, USA.
6. Goutam Kumar Saha, "Instruction Replicated Scheme of Fault Tolerant Computing," Proceedings of the International Conference on Modelling and Simulation, MS 2006, Argentina.
7. Goutam Kumar Saha, "Transient Fault Tolerance Through Recovery," ACM Ubiquity, Vol. 4(29), September, 2003, ACM Press, USA.
8. Goutam Kumar Saha, "Application Semantic Driven Assertions toward Fault Tolerant Computing," ACM Ubiquity, Vol. 7(22), June 2006, ACM Press, USA.
9. Goutam Kumar Saha, "Transient Fault –Tolerance Through Algorithms," IEEE Potentials, Vol. 25(5), 2006, IEEE Press, USA.

GUTAM KUMAR SAHA

In his last nineteen years' research & development and teaching experience, Goutam Kumar Saha e has worked as a scientist in LRDE, Defence Research & Development Organisation, Bangalore and at the Electronics Research & Development Centre of India, Calcutta. At present, he is with the Centre for Development of Advanced Computing, Kolkata, India, as a Scientist-F. He is a fellow in IETE and senior member in

IEEE, Computer Society of India, and ACM etc. He has received various awards, scholarships and grants from national and international organizations. He is a referee of CSI Journal, AMSE Journal (France), IJCPOL (USA), IJCIS (Canada) and of an IEEE Journal / Magazine (USA). He is an associate editor of the ACM Ubiquity (USA), International Journal of the Latin American Center for Informatics Studies (CLEIJ) and of the International Journal of Computing and Information Sciences (Canada). His fields of interest include software based fault tolerance, web technology, EIS, Ontology Engineering and Natural Language Processing. He can be reached via sahagk@gmail.com, gksaha@rediffmail.com.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int i, ic, n, nc;

    float num = numc = 0, sum = sumc = 0, avg = avgc = 0;
    clrscr();
    printf("\n Enter value of n:");
    scanf("%d", &n); /* number of real values */
    printf("\n Re-enter the value of n:");
    scanf("%d", &nc); /* number of real values */
    for(i = ic = 1; i <= n; i++)
    {
        printf("\n Enter real number %d:", i);
        scanf("%f", &num);
        printf("\n Re-enter the real number %d:", ic);
        scanf("%f", &numc); /*duplicate input instruction*/
        sum += num;
        sumc += numc; /* duplicate processing instruction */
        if(sum != sumc) /* voting for consensus answer */
        {
            printf("\n Error");/* error reporting */
            exit(1);
        }
        ic++; /* duplicate instruction */
    }
    avg = sum / n;
    avgc = sumc / nc; /* duplicate processing instruction */
    if (i != ic != n != nc)
    {
        printf("\n Error");/* program control error reporting */
        exit(1);
    }
    if(avg != avgc) /* voting for consensus answer */
    {
        printf("\n Error");/* error reporting */
        exit(1);
    }
    else
        printf("\n Average of numbers = % .3f \n", avg);
        /* correct final answer */
}
```