A Single-Version Scheme of Fault Tolerant Computing

Goutam Kumar Saha
Scientist-F, Centre for Development of Advanced Computing, Kolkata, India
Mail To: CA – 2 / 4 B, Baguiati, Deshbandhu Nagar, Kolkata 700059, INDIA
gksaha@rediffmail.com, sahagk@gmail.com

ABSTRACT

This paper describes how to design low-cost reliable computing software for various application systems, by incorporating a single-version fault tolerant scheme along with run-time signature-based control-flow checking. Most of the ordinary systems lack fault tolerant software fix. 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 because, both the RB and NVP rely on multiple (at least three) versions of both software and computing machines. However, the proposed approach needs a single version (SV) of an enhanced application program that gets executed on one computing machine only. It is common that we often face interrupted service (caused either by an intermittent fault in an application program or in hardware), during the service delivery period of an ordinary cheaper application system. Execution of an application program often show malfunctions or it gets interrupted due to memory bit errors. Error Correction Codes (ECC) (viz., parity, Hamming codes, CRC etc.,) that are used in memory, are not as effective for *online correction* of multiple bit errors, as they are, for the detection of few bit errors. Again, software implemented ECC has a significant overhead over both time and code redundancy. In other words, built in ECC in memory, cannot recover all bit errors but can detect only. As a result, if an error is detected by ECC, the application program needs to be restarted for its re-execution afresh in various microprocessor based application systems. So, the ECC alone is useful for designing a fail-stop kind of system but it suffers from high time redundancy. Other software implemented fault- tolerance schemes are also towards fail-stop kind. But, the proposed (SV) based approach is capable of tolerating such errors without stopping the execution of an application. This SV Scheme (SVS) aims to provide an uninterrupted service at no extra money, but at an acceptable more execution time and memory space. This SV 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, SV- based computing system.

Keywords: Single-Version Scheme, bit errors in memory and register, fail-stop, fault tolerance.

1. INTRODUCTION

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. 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 [1]. 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 [1,2,3] 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 SV scheme is based on a procedure or application triplication along with a signature-based control-flow checking, 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).

2. PREVIOUS WORKS

In order to design an ultra reliability in computing application, it is necessary to adopt the strategy of defensive programming based on code and time redundancy (i.e. fault - tolerant software), e.g., Recovery Blocks (RB) [4,9], N Version Programming (NVP) [5]. 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)* [6], *Assertions* [7, 17, 18], *and Control Flow Checking* [8] are 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. Again, the success of *Control Flow Checking* largely depends on partitioning an application program in basic blocks (branch - free parts of code). For each block, a deterministic signature is computed and errors can be detected by comparing the run-time signature with a pre-computed one. In most of the control flow checking techniques, one of the main problems is to tune the test granularity that should be used. In *procedure duplication (PD)* [19], *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. Row-checksum based fault detection and tolerance has been discussed in the work [11]. Interested readers should refer to other important works on hardware or software implementations of time-constrained and reliable embedded systems [10, 12, 13] also. Other works [14,15,16,20] also discussed on software hardening and the limitations of ECC and conventional software based techniques through single bit fault injection. Software cost analysis for RB, NVP and SIFT approaches have been discussed in [21,22,23,24].

3. THE SVS DESCRIPTION

The proposed SVS technique relies on procedure triplication in order to tolerate one erroneous computation. This scheme detects errors by executing two copies of an application program with similar inputs and then comparing the results. An inequality in results indicates an error. Again, the SV approach is able to tolerate a fault or to mask errors through executing the three copies of an application program with similar inputs and then comparing or voting upon all the results for getting a result in majority.

The basic steps involved in this scheme are stated below.

Step 1. Triplicate an application program in the form of a procedure: PI1, PI2, PI3

Step 2. Sequentially execute: PI1, PI2 with similar inputs.

Step 3. Validate the signature-based control-flow checking and then compare the outputs say, RI1 and RI2 of PI1 and PI2 respectively.

Step 4. If both the outputs (values) are found to be same on comparison (no transient or permanent bit error has occurred or amidst *fail silent faults*), *then* application-system's output is RI1 or RI2.

Step 5. If both the outputs (values) are not same, then execute the third image PI3 with similar input.

Step 6. Validate the signature-based control-flows and compare the outputs (values) obtained from either of these application-copies that is, either RI1 and RI3, or RI2 and RI3, in order to find out equality and to output it.

Step 7. If run-time signatures of control-flows are detected as erroneous then we need to compare (or vote) all the outputs from all the three replicas of an application, and if there is an output (value) in majority, then application-system's output is the majority one only. Thus, faults in either one of the application-replicas or bit-errors in run-time signatures are tolerated by masking the erroneous output (caused by transient or permanent bit errors).

Step 8. If no output in majority, *then* application is restarted or reloaded for re-execution. In such disagreement, SVS converges to a fail-stop kind scheme. The schematic diagram of the SVS with three replicas of an application is shown in figure-1. This scheme is explained in details in the flowchart (as shown in figure-2).

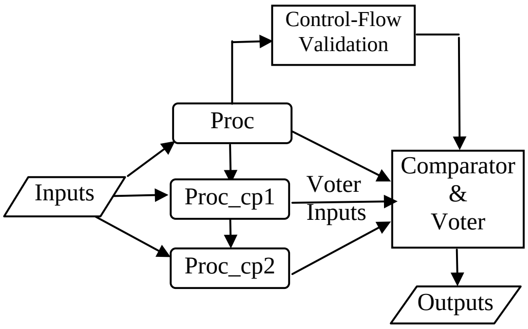

Figure 1. Schematic Diagram of a Single-Version Scheme (SVS).

As shown in the figure 1, the three copies of a procedure that is, *Proc*, *Proc_cp1* and *Proc_cp2* are sequentially executed on similar inputs, and on validating their control-flows, their outputs are compared or voted upon in order to mask an erroneous output. A procedure-copy may compute erroneous result due to transient or permanent- bit errors in its memory space or in processor registers. The scheme is described in detail in a flowchart. The flowchart of the proposed Single Version (SV) based low cost fault tolerant computing scheme is described in Figure 2. Each of the variables: cf1, cf2 and cf3 are to hold the run-time signatures for the control-flows in the replicas: Application_copy_1, Application_copy_2, Application_copy_3 respectively. Here, an application program can also be treated as a procedure itself.

4. THE SV APPLICATION

The SV based fault tolerant scheme is demonstrated on a typical application like "Computing the greatest of 3 numbers" for our understanding. The application program is in the form of a procedure or a function say, *greatest.* We triplicate the *greatest* function into say, *greatest1, greatest2* and *greatest3*. All three copies of the function *greatest* works on similar input variables namely *i1, i2, and i3.* The control flow global variables namely, cf1, cf2, cf3 have been used to keep the run-time signatures of control flows. The cf1, cf2 and cf3 are to store such signatures for the greatest1, greatest2 and greatest3 functions respectively. We use a generic C language programming for understanding.

```
#include <stdio.h>
int greatest1 (int, int, int), greatest2 (int, int, int), 
greatest3 (int, int, int);
int cf1, cf2, cf3;   /* Control-Flow signature variables */
int main (void)
{
  int  i1, i2, i3;  /* input values */
  int  r1, r2, r3, r;
 /* output values; r1, r2 and r3  are the results of
greatest1, greatest2 and greatest3 respectively. The final
output from application is r. */
   int  retrynum;
start: retrynum=0;
/* Initialize the variable for holding number of retry */
            cf1=0;
            cf2=0;
            cf3=0;
      /* Initialize the control-flow signatures variables */

      scanf ("%d %d %d", &i1, &i2, &i3);
                 /* read input data */
      r1= greatest1(i1,i2,i3);
      r2= greatest2(i1,i2,i3);
      r3= greatest3(i1,i2,i3);
      If (cf1==1 &&  cf2==2 && r1 == r2)
 /*validate the control-flow signatures for
greatest1 and greatest2, and compare their output values
*/
          r = r1 ;
          /* final output of the system is either r1 or  r2 */

      else if (cf1==1 && cf3 == 3 && r1 == r3)
/* validate the control-flow signatures for greatest1 and
greatest3 functions, and compare their output values */
         r = r3  ;
    /* final output ( r )of the system is either r1 or  r3 */
       else if (cf2==2 &&  cf3==3 && r2 == r3)
 /*validate the control-flow signatures
for greatest2 and greatest3 and compare their output
values */
         r = r2 ;
   /* final output of the system is either r1 or  r2 */
        else if (r1 == r2 && r1 ==  r3)
 /* compare the output values from all copies */
          r = r1 ;
   /* final output of the system is set to either r1 or  r2 or
r3 */
      else if (retrynum < 1)
        { retrynum++;
         goto start;  }
 /* repeat the computation once again in order to tolerate
transient bit-errors in memory or processor registers */
      else
    /*  only one retry is allowed otherwise re-execute the
application */
        exit(1);
}

int greatest1 (int a, int b, int c)
{
            if ( a > b)
            {
              if (a > c)
                 return a;
              else
                 return c;
             }
              else
              {
                if (b > c)
                  return b;
                else
        return c;
        }
        cf1=1;

 /* set the control flow signature to 1 during execution
of this function. */
}

int greatest2 (int a1, int b1, int c1)
{
   if ( a1 > b1)
      {
        if (a1 > c1)
           return a1;
        else
           return c1;
       }
    else
      {
         if (b1 > c1)
              return b1;
         else
              return c1;
       }
    cf2=2;

 /* set the control flow signature to 2 during execution of
this function. */
}

int greatest3 (int a2, int b2, int c2)
{
   if ( a2 > b2)
   {
     if (a2 > c2)
          return a2;
     else
          return c2;
    }
    else
   {
       if (b2  > c2)
           return b2;
       else
           return c2;
    }
      cf3 =3;

 /* set the control flow signature to 3 during execution
of this function. */

}
```

5. DISCUSSION

The proposed single version scheme (SVS) of fault tolerant computing uses three copies of an application-procedure that reside on memory. Sequential execution of these identical (RAM resident) procedures on a machine can mask *permanent bit errors* in the affected memory space of an image of a procedure. *Transient bit errors in memory* and *processor registers* get masked by executing three images of a procedure, and by validating the run-time control-flow signatures, and then by voting upon the results (similar to NVP) for an output data in agreement. Unlike the conventional fault tolerant scheme, the SV approach does not execute the same procedure-code (or copy) repetitively. We know that the repetitive execution of a code can be helpful for masking transient bit errors only whereas, the SV approach aims to tolerate not only transient bit errors but also permanent-bit errors in memory. Like any other software based fault tolerance approach, the SVS is also not free from both time and space redundancy. Code size increases on an average, here by 3.3 times and, execution time increases by 3.2 times. This approach's overhead on time and memory space is similar to that of a recovery block scheme (RBS), with three alternate application-codes that are based on design diversity. But, the software development cost of a SV based application is almost one-third of both the conventional RBS and NVP Scheme. Unlike PD, this SV scheme does not rely on selective procedure duplication. Thus, even an ordinary programmer who may not have the application system domain expertise can easily implement this SV approach. We assume that bit-errors in voting or comparator code will be detected by the ECC of a modern memory system. However, we may use duplicated comparator codes also with more overheads on memory and execution time to detect errors in voting codes [23, 24]. This SVS can tolerate one control-flow error and one wrong computation in any of the three copies of an application (for example, in *greatest* function). However, the proposed SVS converges to a fail-stop approach when there is no agreement among computed answers or when there is an error in control-flow signature. Like NVP, here also we need *N + 2* copies of an application for tolerating N number of faults. Like any conventional *SIFT, or* triple modular redundancy *(TMR)* based fault tolerance schemes, this SVS approach also cannot claim to be free from an overhead on code and execution time redundancy. Execution time redundancy as observed in SVS on an average is 2.6 times [21] the basic application code without any software fix for fault tolerance.

Overhead Comparison

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 random bit errors on a simple program of Bubble sort (as a benchmark) of 150 integer values, the overhead factors of various approaches including the SVS are listed in *Table-1.* It is found that SVS scheme

Program Approach	**Time Overhead**	**Memory Overhead**
CRC-Non-Distribute	>10	< 2
Hamming	>10	< 3
SVS	> 2.3 and < 3	< 3.2
Triple Modular Redundancy or RBS or an NVP using a Uniprocessor system.	> 3 and < 3.4	< 3.25

Table 1. Overhead factors of various software

can tolerate transient bit errors both on memory and processor registers by error masking through voting. Bit error has been introduced and tested. This approach has also been implemented in a real-life application, e.g., Boiler-Turbine Efficiency computation for a Thermal Power Plant, India. It is observed that this SVS based application (with three copies of application program) is capable of tolerating single memory error amidst an industrial environment. However, software design bug is not tolerated by this SVS approach. Overhead factors are similar to above benchmark program. SVS is applicable to any application where memory constraint does not exist. It is observed that single-version software scheme leads to a better performance.

6. CONCLUSION

The proposed single-version scheme (SVS) for software - based fault tolerance has been described thoroughly in this paper. It is a new variation of other single-version scheme. It does not consider the issue of eliminating software bugs. It is considered that faulty behavior of an application is due to execution - time operational faults that affect the system. Of course, we need to be careful in order to make the program code correct here. Unlike RB and NVP scheme, this SVS does not rely on design diversity in both software and hardware. Rather, it relies on entire application's triplication. Like PD, it relies on two copies for detecting an erroneous result. But unlike PD, it does not rely on selective procedure duplication. Instead it uses entire application's triplication. Unlike other software - based fault tolerance techniques viz., ABFT or control flow checking etc.; this SV scheme is not basically a fail-stop kind scheme. Rather, this is a non-fail-stop kind of fault tolerance scheme. In absence of any fault, the SVS 's time redundancy is only marginally greater than 3. Depending on asking robustness, designer can also use four copies for tolerating two faults. For many applications, the SVS 's overhead (or *limitation*) on memory redundancy and time redundancy is acceptable because of its simplicity in implementation and, of its minimum cost for software development (of one version only), and it does not need multiple machines. Like RB and NVP, this SVS also can suffer from the problem of disagreement among the results. But, unlike both the RB and NVP, this SVS does not crash even if disagreement arises among the results because, SVS then, converges into a fail-stop kind of fault tolerance scheme like any other SIFT schemes viz., ABFT, Control Flow Checking and PD etc. Again, at the modern trend of falling prices on hardware (e.g., processor, memory) cost, we can afford an increased memory, and we can speed up (in order to meet an application's execution time constraint) our SV-based applications' performance by employing an affordable high-speed processor. The cost ratio [22] i.e., (Cost of fault-tolerant software / non-fault-tolerant software) is 2.71 and 2.96 for a three-variant NVP and RB schemes respectively. However, the three-copy SVS 's cost ratio (Cost of fault-tolerant software / non-fault-tolerant software) [24] is only 1.21. It includes the cost of single-version software and the cost for extra codes for voting and branching. We can implement this SVS on a multiprocessor environment also for better performance at the cost of higher hardware-cost. Though SVS is a simple scheme but it is a very useful and effective tool for designing many low-cost, reliable computing applications.

7. REFERENCES

[1] L. Spainhower and T.A. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," IBM Journal of Research & , Vol. 43, No. 5/6, 1999.

[2] T. Sato, "Analyzing Overhead of Reissued Instructions on Data Speculative Processors," Workshop on Performance Analysis and its Impact on Design, held in conjunction with 25th International Symposium on Computer Architecture, 1998.

[3] Stephen B. Wicker, Error Control Systems for Digital Communication and Storage, Prentice Hall, NJ, USA, pp.72- 127, 1995.

[4] B. Randell, "Design - Fault Tolerance," in The Evolution of Fault-Tolerant Computing, A. Avizienis, H. Kopertz, and J.-C. Laprie, eds., Springer-Verlag, Vienna, 1987, pp. 251-270.

[5] A. Avizienis, "The N-Version Approach to Fault – Tolerant Systems," IEEE Transactions on Software Engineering , Vol. SE -11, No. 12, Dec., 1985, pp.1491-1501.

[6] K.H. Huang, J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," IEEE Transactions on Computers, Vol. 33, 1984, pp. 518-528.

[7] M. Zenha Rela, H.Madeira, J.G. Silva, "Experimental Evaluation of the Fail-Silent Behaviour in Programs with Consistency Checks," Proceedings of the FTCS-26, 1996, pp.394-403.

[8] S. Yau, F. Chen, "An Approach in Concurrent Control Flow Checking," IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, 1980, pp. 126-137.

[9] K.H. Kim and H.O. Welch, "Distributed Execution of Recovery Blocks: An Approach for uniform Treatment of Hardware and Software Faults in Real- Time Applications," IEEE Transactions on Computers, Vol.38, No. 5, May 1989, pp. 626-636.

[10] R.K. Gupta, C.N. Coelho, G. De. Micheli," Program Implementation Schemes for Hardware - Software Codesign," IEEE Computer, June 1994, pp. 48-55.

[11] Goutam K. Saha, "Transient Fault Tolerant Processing in a RF Application," International Journal – System Analysis Modelling Simulation, vol. 38, 2000, Gordon and Breach, USA, pp.81-93.

[12] R.K. Gupta, C.N. Coelho Jr., G.De. Micheli, "Sysnthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," Proc. Design Automation Conference, June 1992.

[13] Yervant Zorian, Dimitris Gizopoulos, "Design for Yield and Reliability," IEEE Design & Test, May/June, 2004.

[14] G.K. Saha, "Designing an EMI Immune Software for Microprocessor Based Application," Proceedings 11th IEEE International Symposium, EMC'95, Switzerland, March, 1995, pp. 401-404.

[15] A. Benso, P.L. Civera, M. Rebaudengo, M. Sonza Reorda, "An Integrated HW and SW Fault Injection Environment for Real -Time Systems," Proc. IEEE

International Symposium on Defect and Fault Tolerance in VLSI Systems, 1998, pp. 117-122.

[16] Goutam Kumar Saha, "Transient Fault Tolerance in Mobile Agent Based Computing," INFOCOMP Journal of Computer Science, Vol. 4, No. 4, 2005, pp. 1-11.

[17] Goutam Kumar Saha, "Fault Tolerant Computation for a Scientific Application," CSI Communications, Computer Society of India Press Mumbai, Vol. 20(5), May 1996.

[18] Goutam Kumar Saha, "EMP- Fault Tolerant Computing: A New Approach," International Journal of Microelectronic Systems Integration, Vol. 5, No.3, Plenum Publishing Corp, USA, 1997, pp. 183-193.

[19] D.K. Pradhan, Fault - Tolerant Computer System Design, Prentice Hall, 1996.

[20] B. Nicolescu, R. Velazco, M. Sonza-Reorda, "Effectiveness and Limitations of Various Software Techniques for Soft Errors Detection: A Comparative Study," TIMA Lab. Research Reports: ISRN TIMA-RR-01/10-7-FR, 2001, France.

[21] Goutam Kumar Saha, "Software Implemented Fault Tolerance Through Data Error Recovery," ACM Ubiquity, vol. 6(35), September 2005, ACM Press, USA.

[22] C.V. Ramamoorthy et al., "Software Engineering: Problems and Perspectives," Computer, Vol. 17, No. 10, October 1984, pp. 191-209.

[23] Goutam Kumar Saha, "Low- Cost, Fault Tolerance Applications," IEEE Potentials, Vol. 24, No. 4, 2005, IEEE Press, pp. 35-39.

[24] Goutam Kumar Saha, "Software Based Fault Tolerant Computing," ACM Ubiquity, vol. 6, No. 40, November 2005, ACM Press, USA.

Received: Nov 2005. Accepted: Feb. 2006.

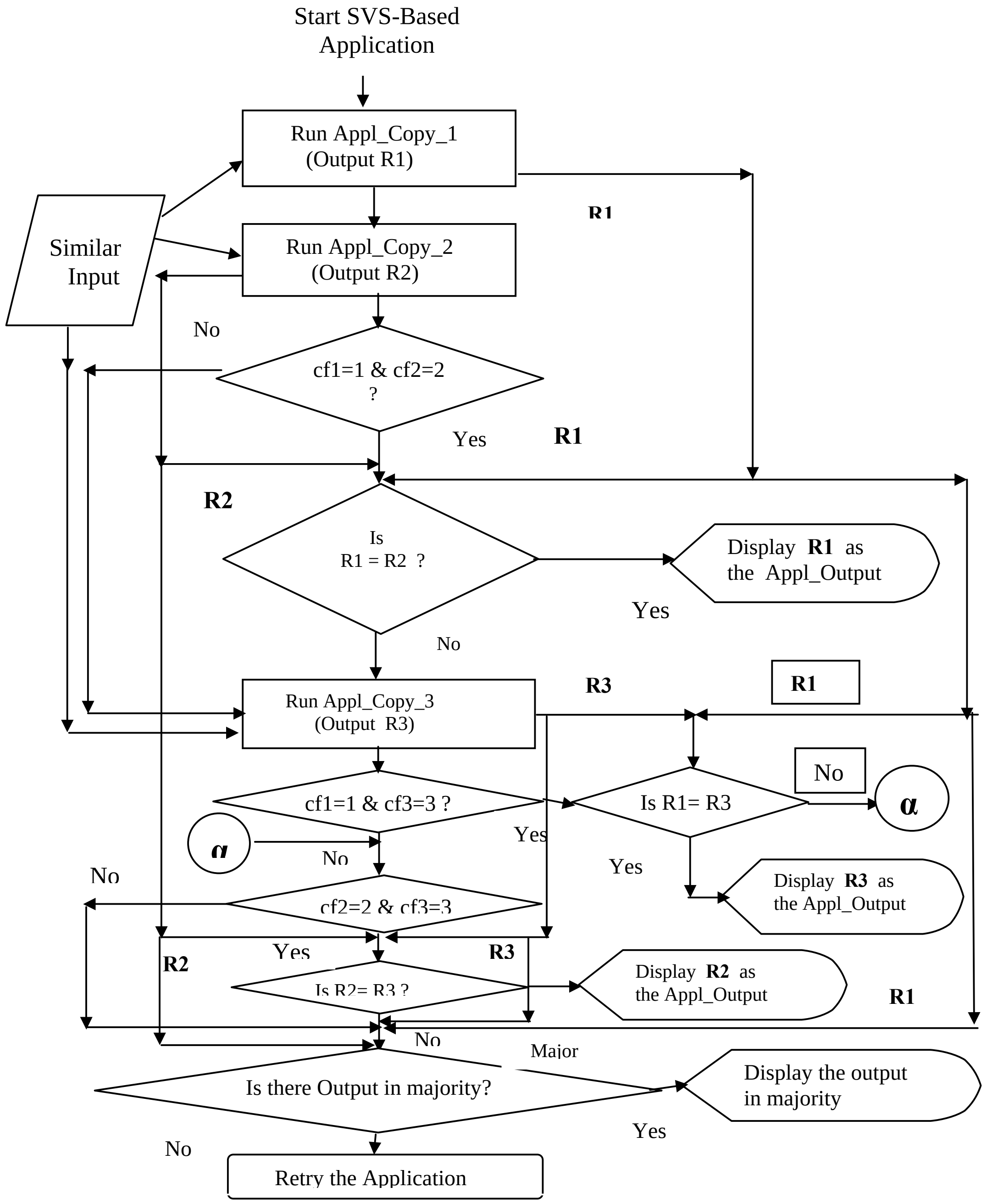
Start SVS-Based Application
Run Appl_Copy_1 (Output R1)
R1
Similar Input
Run Appl_Copy_2 (Output R2)
No
cf1=1 & cf2=2 ?
Yes
R1
R2
Is R1 = R2 ?
Display R1 as the Appl_Output
Yes
No
Run Appl_Copy_3 (Output R3)
R3
R1
cf1=1 & cf3=3 ?
Is R1= R3
No
α
Yes
α
No
No
Yes
Display R3 as the Appl_Output
cf2=2 & cf3=3
Yes
R2
R3
Is R2= R3 ?
Display R2 as the Appl_Output
R1
No
Major
Is there Output in majority?
Display the output in majority
Yes
No
Retry the Application

Figure 2. Flowchart of the SVS based Fault Tolerant Computing Scheme.