Making Resilient Application

Goutam Kumar Saha
Email: gksaha@ieee.org, sahagk@gmail.com

Application is made resilient here in order to survive against operational hazards. This simple approach uses variable time redundancy. This approach does not aim to avoid software design bugs. This approach is also useful to tolerate transient bit-errors in the processor registers and memory. This low-cost method is a useful tool for designing an application program that can avoid and tolerate operational transient, intermittent or Byzantine faults without using design diversification of application's code.

The approach here needs only one instance of the application program along with decider or voter program code running sequentially on one processor in a multiprocessor system in order to tolerate various types of operational faults. At an ideal situation with no fault, this scheme needs to execute the application twice only. At each run with similar input, the computed result is stored or copied onto a different global variable. For example, at the first run, result is stored or copied on to G_1. At the i th run, the result is stored or copied on to G_i and so on. Here, with the limit of maximum four possible retries and with one output at each execution, the number of global output variables will be four only. For example, G_1, G_2, G_3, G_4 are the four global variables for holding the results. These public variables are initialized to zero. The following steps describe how to develop an application with resilience without hardware redundancy. The approach uses various extra codes meant for the decider-cum-voter. The symbol "/* */" is used to enclose a remark.

Step 0: Set the Global Variables $G_1 = G_2 = G_3 = G_4 = 0$
/* Initialize the public output variables */
/* These global variables are referred by the decider-cum-voter codes */

Step 1: ***Branch to*** **the Application Program** /* First run */
/*At the first run, result is copied on to the global variable G_1 */

Step 2: ***Branch to*** **the Application Program** /* Second run */
/*Second run result is copied on to the global variable G_2 */
/* Decider-cum-Voter Codes begin here*/

If $G_1 == G_2$ **, Then:**
Go to Step 0.
/* After two runs, both the corresponding results are compared */
/* Results are same. No fault is detected, so
application continues. */

Else IF **(**G_1***.NE.*** G_2**), Then:**
/*First two runs' results are not matching to each other */
/* Transient Fault is detected. Decider retries it for the third time*/

Branch to **the Application Program.**
/* Third run */
/* Result is copied on to the global variable G_3 */
If **(**G_1***.EQ.*** G_3 ***.OR.*** G_3 ***.EQ.*** G_2**), Then:**

```
                /* Two results among the three are matching.
                                Result with majority is found  */
                        Go to   Step 0.
                                /* Transient fault is tolerated,
                                and the application continues. */
                    Else If  (G1.NEQ. G 2  .NEQ.  G3), Then:
                                        /*  No result with majority */
                                /* Results out of the previous three  runs,  are not  matching */
                        Branch to  the  Application Program.
                                                /* Fourth run */
            /*At the fourth run, result  is copied  on to  the  global variable G4 */

                    If  (  G4 .EQ. G3  .OR .  G4  .EQ.  G2  .OR .  G4  .EQ.  G 1 )  ,  Then:
                        /*Because of two intermittent faults,
                          result in majority is found after four runs. */
                            /* Two Intermittent faults are tolerated by four runs. */
                                Go to Step 0.  /* So application continues. */
                        Else:
/* Result in majority is not found after four runs i.e., Byzantine failure. */
                                Do Cut_ Over
                                        /* Switch to a backup processor to tolerate a
                                        faulty primary processor. */
                                Execute through Step 0.
                                /*On tolerating the Byzantine failures exhibited
                                        by a faulty processor,  application  continues.  */
                            End If          /* End of  the innermost  If  block */
                    End IF                  /* End of the intermediate If block */
            End If      /* End of the outermost If block */
              /* End of the decider-cum-voter codes */
            {END of the Logical Steps}
```

Reading:

- Avizienis, "The N-Version Approach to Fault – Tolerant Systems," IEEE Transactions on Software Engineering , vol. SE -11, No. 12, Dec., 1985,pp.1491-1501.
- 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.
- S. Dolev, E. Schiller and J. Welch, " Random Walk for Self – Stabilizing Group Communication in Ad-Hoc Networks," Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems, SRDS 2002, pp. 70-79.
- Goutam Kumar Saha, "Software Based Fault Tolerance: a Survey," ACM Ubiquity, 2006.