N-MODULAR REDUNDANCY+K

Esin Sanem İmamoğlu
Electronic and Communication Engineering
IzTECH
Izmir,Turkey
esinsimamoglu@gmail.com

Aslı Özkan
Computer Engineering
IzTECH
line 4: City, Country
asliozkan@std.iyte.edu.tr

Barış Kıpkıp
Electronics and Communication Engineering
IzTECH
Izmir,Turkey
barisk.kpkp@gmail.com

Abstract

This article provides a review of NMR + K which is an application of Hybrid Redundancy. We developed a pattern and implemented it in Python programming language.

Keywords

NMR, hybrid redundancy, pattern, reliability, Python

I. INTRODUCTION

Fault tolerance is the ability of a system to continue to function appropriately in the event of a failure. Hybrid redundancy is the combination of static redundancy and dynamic redundancy [1]. N systems perform a process, and the outcome is processed by a majority-voting system to generate a single output in N-modular redundancy. If one of these systems fails, the others can compensate and mask the error [2]. The procedure is as follows: There are N active nodes and S spare nodes in the network. There are N modules that create outputs. If they all match, a common output is generated. If a mismatch is found, the voter calculates the majority and discovers the faulty module, which is then replaced with a spare node. The majority is obtained by determining if the number of common results is greater than half of the total number of modules, which is also known as percent 51 [3].

II. PATTERN DESCRIPTION

In the code, there are N copies of the original module and S spare nodes. Spare nodes are modules doing same job with original modules in a different way. The results of these original modules are compared via voter. It determines the majority and then faulty modules are replaced by the spare nodes. Therefore, faulty modules are tolerated until there is no spare node. Firstly, number of modules and spare nodes are taken by the user. There is a class that contains two functions. Init function returns result and calls NMR function. It has five arguments as shown in Figure 1. ‘code’ is defined as original module, ‘module_count’ is the number of NMR modules, ‘code_spare’ is spare nodes, ‘spare_node_count’ is the number of spare nodes.

```
1   import numpy as np
2   class hybrid:
3       def _init_(self,code,module_count,spare_node_count,code_spare):
4           self.code=code
5           self.module_count=module_count
6           self.spare_node_count=spare_node_count
7           self.code_spare=code_spare
8           result=self.NMR(code,module_count,spare_node_count)
```

Figure 1: Init function

Second function is NMR function is the one where the whole process takes place. Arguments of this function is the same as the init function with one extra input which is the original module. Initially, empty arrays are generated called ‘results’ and ‘count’ as shown in Figure 2. Results array contains results of the modules and count array defines the number of modules with the same result.

```
9       def NMR(self,code,modul_count,spare_node_count,code_spare,*argv):
10          results=np.empty(modul_count+1)
11          count=np.empty(modul_count+1)
```

Figure 2: NMR function

Then, an integer called ‘result’ is defined which represents correct result found by calculating the majority value as shown in Figure 3. It assigned to -1000 initially to mark the faulty module. Plus, an integer called ‘count’ in order to increment each time correct result is found.

```
12              result=-1000
13              max_count=0
```

Figure 3:

Afterwards, in the first for loop, results of N-modular redundancy from original modules are stored in ‘results[]’ array as shown in Figure 4. In the second for loop, these results are compared with each other and the ‘count’ array is incremented by one each time a correct result is returned. The number of identical results is stored in ‘count’ array. In order to compute the majority, integer ‘max_count’ is defined which is the maximum value of the count array. It also corresponds to the maximum number of identical results.

```
13          max_count=0
14          for i in range(1,modul_count+1):
15              results[i] = code(*argv)
16          for i in range(1,modul_count+1):
17              for j in range(1,modul_count+1):
18                  if(results[j] == results[i]):
19                      count[j] += 1
20          max_count=max(count)
```

Figure 4: Finding maximum count

For the next loop, there is a condition checks whether the number of common results is greater than half of the total number of modules. If the condition satisfies, in the next for loop, result of the maximum value of count array is assigned as the result as shown in Figure 5.

```
21      if(max_count > (modul_count / 2)):
22              for i in range(1,modul_count+1):
23                  if (count[i] == max_count):
24                      result = results[i]
```

Figure 5: Assigning results of majority

Then, results that do not provide majority replaced by spare nodes until there is no more spare node as shown in Figure 6. It is performed by checking whether the ‘results’ array element is equal to the majority of the results. If they are not equal and there is enough spare node, that element is replaced by the spare node. How many spare modules are left and if not, is printed on the screen.

```
25              for i in range(1, modul_count + 1):
26                  if(results[i]!=result and spare_node_count!=0):
27                      results[i] = code_spare(*argv)
28                      spare_node_count = spare_node_count - 1
29                      print('Spare node left:')
30                      print (spare_node_count)
31                  elif(spare_node_count==0):
32                      print('There is no spare node left')
33                      break;
```

Figure 6: Replacement of spare nodes.

If the majority result is not assigned to the ‘results’ integer, it is equal to its initial value which is -1000 and prints an error. Otherwise, majority result is returned as shown in Figure 7.

```
34          if(result==-1000):
35              print('Error')
36          else:
37              print(result)
38              return result
```

Figure 7: Printing results

In Figure 8, example code is shown. First definition is for original module. The second one is for spare node. Then, hybrid class and NMR function is called. Arguments for NMR function is added. In this example, there are 7 modules and 2 spare nodes. This example code runs for input 5 and the correct result is 125.

```
from embeded_proje import hybrid
def code(a):
    return a**3
def code2(b):
    return b*b*b
p1=hybrid()
x=p1.NMR(code,7,2,code2,5)
```

Figure 8: Example code for NMR+K

III. CONCLUSION

N-Modular Redundancy+K compares the output of the voter with the output of the active modules and then replaces modules whose output disagree with the output of the voter with spares. This technique has the highest reliablitiy compared to the other techniques. However it is the most expensive one because of the amount of hardware required to implement a system [4].

To make clear all the steps disscussed, example code gives outputs as follows. When one of the argument from 'results[]' array is different from the others, code output is as shown in Figure 9.

```
Spare node left:
1
125.0
```

Figure 9: Output of example code with one different output.

If another argument is made different too, output is as shown in Figure 10.

```
Spare node left:
1
Spare node left:
0
There is no spare node left
125.0
```

Figure 10: Output with two different arguments.

When there is no majority, output is as shown in Figure 11.

```
Error

Process finished with exit code 0
```

Figure 11: Output of example code with no majority.

IV. REFERENCES

[1] L. -C. Chu ve B. -W. Wah, «Fault tolerant neural networks with hybrid redundancy,» IEEE, 1990.

[2] K.-C. Lee, S. Kim ve H. Man, *Predictive hybrid redundancy using exponential smoothing method for safety critical systems*, International Journal of Control Automation and Systems, 2008.

[3] M. Hafezparast, *FAULT-TOLERANT HARDWARE DESIGNS*, Department of Electrical and Electronics Engineering,, 1990.

[4] D. P. Siewiorek ve R. S. Swarz, *Redundancy in fault tolerant computing*, Reliable Computer Systems, 1992.

APPENDIX

```
import numpy as np

class hybrid:

    def _init_(self,code,module_count,spare_node_count,code_spare):
        self.code=code
        self.module_count=module_count
        self.spare_node_count=spare_node_count
        self.code_spare=code_spare
        result=self.NMR(code,module_count,spare_node_count)

    def NMR(self,code,modul_count,spare_node_count,code_spare,*argv):
        results=np.empty(modul_count+1)
        count=np.empty(modul_count+1)
        result=-1000
        max_count=0

        for i in range(1,modul_count+1):
            results[i] = code(*argv)
        for i in range(1,modul_count+1):
            for j in range(1,modul_count+1):
                if(results[j] == results[i]):
                    count[j] += 1
        max_count=max(count)
        if(max_count > (modul_count / 2)):
            for i in range(1,modul_count+1):
                if (count[i] == max_count):
                    result = results[i]
            for i in range(1, modul_count + 1):
                if(results[i]!=result and spare_node_count!=0):
                    results[i] = code_spare(*argv)
                    spare_node_count = spare_node_count - 1
                    print('Spare node left:')
                    print (spare_node_count)
                elif(spare_node_count==0):
                    print('There is no spare node left')
                    break;

        if(result==-1000):
            print('Error')
        else:
            print(result)
            return result
```

Example code:

```
from embeded_proje import hybrid

def code(a):
    return a**3
def code2(b):
    return b*b*b

p1=hybrid()

x=p1.NMR(code,7,2,code2,5)
```