A Uniform Approach to Software and Hardware Fault Tolerance

J. Wu, Y. Wang and E. B. Fernandez
Department of Computer Science and Engineering
Florida Atlantic Unversity
Boca Raton, FL 33431

Abstract

In recent years, various attempts have been made to combine software and hardware fault tolerance in critical computer systems. In these systems, software and hardware faults may occur in many different sequences. The problem of how to incorporate these sequences in a fault-tolerant system design has been largely neglected in previous work. In this paper, a uniform software-based fault tolerance method is proposed to distinguish and tolerate various sequences of software and hardware faults. This method can be based on the recovery block or the n-version programming scheme, two fundamental software fault tolerance schemes. The concept of fault identification and system reconfiguration (FISR) tree is proposed to represent the procedure of fault identification and system reconfiguration in a systematic way.

1 Introduction

During the past years, a good deal of progress has been made in the development of fault tolerance techniques that improve the ability of computer systems to cope with software or hardware failures separately. Two of the most widely used techniques for software fault tolerance are *recovery block* [2] and *n-version programming* [1] which have been applied to several situations [8]. These two mechanisms assume that the processor will not fail when executing software or hardware faults will be handled by a complementary mechanism.

A recovery block (RB) consists of a set of versions and an acceptance test. The acceptance test is applied to the result of the primary version. Control exits from the recovery block if the execution result passes the acceptance test, otherwise the next alternate version is executed. This process is repeated until some version passes the acceptance test or all versions fail. A possible situation is that all alternate versions cannot pass these acceptance tests due to faulty hardware and this fault may be erroneously identified as a software fault. N-version programming (NVP) also uses design redundancy, but all versions of an algorithm execute in parallel on different processors. The result of each version is then sent to a decision mechanism which outputs the final result. A generally used decision method is majority voting, which can mask erroneous results from some versions provided that results from correct versions constitute a majority. As a matter of fact, an erroneous result may be either caused by a fault in a software version or by a hardware fault. The identification of the cause of the fault is needed when replacement of faulty components and reconfiguration are performed in order to provide continuous availability of the system.

Moreover, there are many other possible cases when software and hardware faults occur in the same processor, and also faults may appear in any sequences and combinations. This can make fault tolerance quite complex. If software and hardware fault tolerance techniques are only used separately, it may not be possible to reach a high level of reliability. There is therefore a need to develop a unified method for tolerating both software and hardware faults in combination. Some attempts have been made to deal with this problem in [9], [10], and [13]. But different sequences and combinations of software and hardware faults were not addressed, and no systematic approach for their combined detection and handling was presented.

In this paper, we first define a *fault pattern* as a set of possible fault combinations and then propose a uniform method for detecting, locating and tolerating predefined fault patterns by repeated execution and reconfiguration actions. The method is developed by using RB or NVP methods for software fault tolerance and by using comparison and voting for hardware fault tolerance. In addition, a *fault identification and system reconfiguration* (FISR) tree is proposed to illustrate the procedure of detecting and locating fault patterns, reconfiguring the processors and changing software assignments to increase availability of the system. In a FISR tree, each internal node represents a system state which may indicate that some faults exist in the system. Each leaf in the FISR tree represents a system state interpretation in which fault-free results are obtained, or where a fault pattern is identified. Possible reconfiguration actions are associated with edges in each path from the root to one of the leaves. Each fault pattern can be tolerated by following the path from the root to the corresponding leaf.

1066-6192/92 $3.00 © 1992 IEEE

An application can be considered as a sequence of system reconfiguration periods such that faulty software versions and processors are identified and removed (either physically or by reconfiguration of the system). In case no fault occurs, each system reconfiguration period corresponds to a regular execution of a program unit or a frame in a real-time system. A FISR tree represents a system reconfiguration period and the depth of the FISR tree represents the maximum reconfiguration steps within the period. To simplify the discussions, we assume that the execution of the FISR tree is fault-free. Although creating such an environment is not a trivial task, it is beyond the scope of this paper.

In general, the ability of a system for fault detection and location is based on the available set of fault tolerance or fault detection methods. Different fault tolerance methods may generate different FISR trees. Two general questions can be addressed, although they are not considered here. One is how to find a minimum (or convenient) set of fault tolerance methods for a given set of fault patterns such that all the fault patterns can be identified and tolerated. The other one is how to find a minimum set of fault tolerance mechanisms such that all the faults can be identified and tolerated in a given number of (reconfiguration) steps. In this paper, we concentrate only on the construction and analysis of the FISR tree as well as necessary actions to be performed based on the given set of fault patterns under the given fault tolerance methods. To make our discussion more general, the selection of fault patterns and fault tolerance methods is not targeted to any particular application, e.g. real-time applications. However, the scheme can be tailored to specific applications by selecting an appropriate set of fault patterns and fault tolerance methods.

The rest of the paper is structured as follows: In Section 2, basic notation, definitions and assumptions are given. In Section 3, an RB-based method is proposed, while in Section 4 an NVP-based method is discussed. An analysis of the system based on FISR trees is discussed in Section 5. Some conclusions are stated in Section 6.

2 Notation and Assumptions

For simplicity we make the following assumptions:

- A software fault is permanent, which makes a software version fail for some specific inputs. A hardware fault may be either transient (denoted as H_s) or permanent (denoted as H_h).
- Coincident errors may occur among software versions, thus several software versions fail at the same time under the same input. But they are assumed not to generate identical erroneous results, that is, there are no *correlated errors* [6]. Similarly, two faulty processors will not produce correlated errors either.
- There are enough software versions and processors in the backup pool when replacement and reconfiguration actions are performed.

The first assumption has to do with failure semantics [5], that is, the type of faults to be covered. A software version or a processor with permanent faults should be made passive, i.e., removed from the system once identified. Although our methods can be extended to cover faulty software versions or processors which generate correlated errors, it will be more expensive to implement the scheme since more software versions, processors and time are required [3], [14]. The selection of fault models to be included is dependent on the applications. In general, the stronger a specified failure semantics is, the more expensive and complex their recovery. In our discussions, a hardware fault is treated as permanent if it occurs in two consecutive executions, although it may actually be a transient fault.

Table 1 lists possible fault sequences assuming a maximum of two faults occurring within a reconfiguration period.

Table 1: Possible fault sequences

Notation	*Meaning*
(F_1)	one fault
(F_1, F_2)	two consecutive faults
$(F_1 \& F_2)$	two concurrent faults

In Table 1, F_1 and F_2 can be either a software fault (S) or a hardware fault (H). For example, (F_1, F_2) actually represents four fault patterns: $(S,H),(H,S),(H,H)$, and (S,S). Two faults in each pattern occurs in consecutive execution periods. Two concurrent faults in $(F_1 \& F_2)$ occur in the same execution period.

A fault pattern is said to be *fully covered* by a method if it can be distinguished from other patterns and can be tolerated by using the method. A fault pattern is said to be *partially covered* if it can be tolerated but can not be distinguished from other fault patterns. In some real-time applications only partial covering of fault patterns may be needed as long as a correct result is provided because of time and cost constraints. But the designers of fault-tolerant systems need to locate which software versions or processors are faulty, so that they can reconfigure the system by replacing faulty components. Note that the time of delivery of a result normally occurs before the time of completing a diagnosis or a reconfiguration. This fact will be illustrated in Sections 3 and 4.

A *fault set* is defined as a set of fault patterns. For simplicity, only some special types of fault sets are considered in this paper. In the first type of fault set, the maximum number of hardware and software

faults in each fault pattern is limited. In the second type the maximum number of total faults in each pattern is limited, but any possible sequence of software and hardware faults is included. The notation (m/n) is used for the first type of fault set which stands for a maximum of m software faults and n hardware faults within a reconfiguration period, e.g. (1/1) indicates a fault set $\{(S,H),(H,S),(S\&H)\}$. The notation (m) represents the second type of fault set in which the maximum number of faults is m, e.g. the fault set (2) includes fault patterns: $(S),(H),(S,H),(H,S),(S,S),(H,H),(S\&S),(H\&H)$ and $(S\&H)$. In addition, these faults may be distributed on several processors or in several software versions. Clearly fault set (2) includes fault set (1/1).

Three fault detection methods are used in the approach: *acceptance tests, result comparison* and *voting*. The selection of fault detection mechanisms here is by no means optimal but this use is just to illustrate our approach. In general the selection of fault detection mechanisms depends not only on the selected fault tolerance scheme but also on the underlying architecture. For example, in the *extended distributed recovery block* scheme [13], two fault detection methods are used. One is the acceptance test used to detect software faults. The other one is *heartbeats* which check the operational node pair to detect hardware faults with the help of a central supervisory node.

In general the acceptance test is a boolean function without side effects which can test whether the result satisfies the specification and is assumed to have a perfect fault coverage. (While in general this is not the case, it is however, a reasonable starting approximation.) A comparison checks whether two results produced by the same software version running on different processors match each other. A voting mechanism called *double voting* (denoted as R_d) is defined in terms of *majority voting* (denoted as R_m). V_{ij} denotes the result produced by software version V_i executing on processor PE_j. The result delivered by the majority voting on (..., V_{ij}, ...) with more than three elements is represented as:

$$R_m = majority(..., V_{ij}, ...)$$

If R_m produces a result this must be correct because no correlated faults are assumed to occur. Otherwise, no result is produced (denoted as "-"). A *double voting* is defined by the tuple $(V_{ix}, V_{iy}, V_{jz}, V_{kw})$ where version V_i is assigned to two PEs (PE_x and PE_y), and versions V_j and V_k are assigned to PE_z and PE_w respectively. Assuming that R_{m1}= majority (V_{ix}, V_{jz}, V_{kw}) and R_{m2} = majority (V_{iy}, V_{jz}, V_{kw}), the result delivered by *double voting* R_d is defined in Table 2.

Clearly, for the fault sets (1/1) and (2), the situation that $(R_{m1} \neq \text{"}-\text{"}) \neq (R_{m2} \neq \text{"}-\text{"})$ will never happen, therefore whenever there is a result in R_d, it must be a correct one.

Table 2: Definition of double voting

R_{m1} *vs.* R_{m2}	*Result of Double Voting*
$R_{m1} = R_{m2}$	$R_d = R_{m1} = R_{m2}$
R_{m1} = "-"	$R_d = R_{m2}$
R_{m2} = "-"	$R_d = R_{m1}$

X-Y(FS) represents the fact that methods X and Y are used for software and hardware fault tolerance respectively to identify and tolerate the fault set FS. In this paper, we consider the following four types of X-Y(FS):

- RB-C(1/1)
- RB-C(2)
- NVP-C(1/1)
- NVP-C(2)

Here RB stands for recovery block, C for comparison, and NVP for n-version programming. Note that the RB scheme uses acceptance test and the NVP scheme uses majority voting and double voting. In the proposed method, whenever a faulty software version or a permanently faulty PE is made passive after its identification one should assign new software versions or reconfigure PEs in order to make the system be able to tolerate subsequent faults and thus provide continuous availability. For this purpose, the following actions are used:

- *passive(PE or V)*: The specified processor PE or software version V is made passive. If PE or V is unspecified, the selection is based on the result of the acceptance test or voting.
- *exec(..., V_{ij}, ...)*: The execution of V_i on PE_j respectively. Any PE_j can be missing which is denoted by "-" and means that it is not active in the execution.

Each software version V or processor PE is in one of two states: active or passive. We assume that the system can be changed only through the above defined actions. In order to have a convenient representation for combinations of fault detection actions followed by reconfiguration actions we use a FISR tree which is a special tree where each internal node represents a system state, determined by acceptance test, result comparison, voting or their combinations on software versions executing on different PEs. In general, reconfiguration actions must be applied to those internal nodes which indicate that faults may exist in order to make the system reach one of leaves.

Based on the fully covering or partial covering of fault patterns, the leaves in the FISR tree are divided into two groups. Each leaf in the first group is unable to distinguish fault patterns. It may (represented by a bold square) or may not (represented by a regular square) be able to deliver a correct result. Each leaf in the second group (represented by a circle) is able to

provide a correct result as well as distinguish the fault patterns. If the system reaches a square leaf, it is still in an inconsistent system state, otherwise it is in a consistent system state. All the internal nodes are in inconsistent system states, although they may (represented by a bold rectangle) or may not (represented by a regular rectangle) be able to deliver a correct result.

In the proposed method, three types of fault sets can be identified based on the degree of difficulty in fault identification. The first type represents that all fault patterns can be identified and a correct result is delivered within a given number of reconfiguration steps (or executions). The second type is similar to the first one but some fault patterns in the set cannot be identified in a given number of reconfiguration steps. However, a correct result can still be reached for each fault pattern in the set. The worst case (the third type) is that some fault patterns cannot be distinguished and also no correct result is obtained. The above classification is dependent on the fault identification method used. In general, the fault identification method should be chosen such that there will be no third type of fault set. Depending on the nature of an application, e.g., a real-time application, where it is too costly to fully cover all fault patterns, sometimes fault patterns are only partially covered, although the method has the ability to fully cover these fault patterns by increasing the number of reconfiguration steps. The main sequence of fault identification and reconfiguration steps is defined in the following way:

repeat
  execute versions;
  obtain and analyze system state;
  reconfigure system;
until (correct results are delivered or
  fault patterns are identified or
  time constraints are exceeded).

Normally, the above procedure stops when correct results are delivered, but the fault pattern may or may not be fully covered, that is each leaf in a FISR tree is a bold square or a circle. When time constraints are exceeded, neither correct results nor fault pattern identification can be guaranteed.

A path from the root to one leaf in the FISR tree is composed of several loops of the above procedure. The time constraints used in the procedure intend to prevent performance degradation, which is especially critical in real-time systems [11]. We assume that the FISR tree is executed on a hard core mechanism which is fault free. This hard core collects execution results, performs results checking by some error detection method and perform necessary configurations. The implementation of this hard core is beyond the scope of the paper. The next two sections describe the RB-C(1/1) and NVP-C(2). The detailed discussion on RB-C(2) and NVP-C(1/1) can be found in [12].

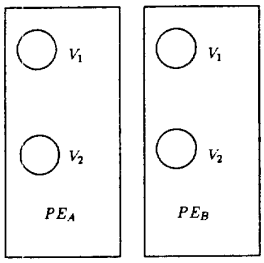

Figure 1: System architecture for fault set (1/1)

3 The RB-C(1/1) Method

In this section, we study fault identification and system reconfiguration using recovery blocks for the fault set (1/1). The identification and reconfiguration steps are given using the concept of FISR tree defined in the last section. A system state is denoted as *(iCP, jAP)* where *i* is the number of matching results of comparisons and j is the number of versions that pass the acceptance test. The supporting architecture for this method is shown in Figure 1. This architecture represents the minimum number of versions and PEs that can be used to cover the fault patterns in fault set (1/1).

Table 3: System states for the $RB-C(1/1)$ method

System States	*Fault Patterns*
(2CP, 2AT)	no error
(2CP, 0AT)	(S)
(0CP, 1AT)	(H)
(0CP, 0AT)	(S & H)

Initially version V_1 is executed on both PEs. Then a comparison takes place between the two results provided by both PEs. If there is a fault in the version, the comparison will not detect this fault but the acceptance tests will fail in both results. If a hardware fault occurs in one of the PEs, the comparison and one of the acceptance tests will fail. Finally, simultaneous hardware and software faults will make both the comparison and the acceptance test fail. Therefore, we can distinguish a hardware fault from a software fault. Table 3 shows a list of possible system states, while Figure 2 shows the FISR tree.

The state (0CP, 0AT) indicates a fault pattern *(S&H)*, which has two possible fault distributions: the software fault and the hardware fault can be at the same PE or at different PEs. In the RB-based method, these two distributions produce the same effect because identical software versions are executed on all PEs.

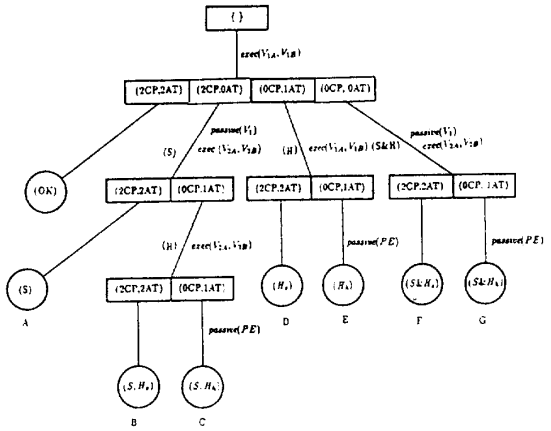

Figure 2: FISR tree for RB-C(1/1)

The FISR for this architecture is shown in Figure 2, where the root represents a consistent system state. Every internal node represents a system state after an execution of the version. The path from the root to a leaf is a sequence of diagnosis and system reconfiguration actions. Partial diagnosis are shown at the left of edges, while reconfiguration actions are shown to the right of edges.

In the FISR tree in Figure 2, when the state (2CP, 0AT) is identified (which indicates a software fault), the second version V_2 is executed in both PEs: *exec*(V_{2A}, V_{2B}). If the system state (2CP, 2AT) appears after the execution of V_2, the system reaches leaf A with a final diagnosis result of no hardware fault and one software fault (in V_1). If the system state (0CP, 1AT) appears after the execution of V_2, a hardware fault is identified which occurs before or during the execution of V_2 but after the execution of V_1. Because there will be no software fault due to the assumption of fault set (1/1), the second round execution of V_2 is activated to determine the nature of the hardware fault, which is either transient or permanent. After this execution, if the system state (2CP, 2AT) appears, the leaf B is then reached which indicates the final diagnosis result: a software fault (in V_1) and then a transient hardware fault. Otherwise if the system state (0CP, 1AT) appears, a permanent hardware fault is identified and the corresponding PE is made passive. A similar analysis can be applied to the other leaves in the FISR tree. All the fault patterns in set (1/1) are fully covered in Figure 2, thus all the leaves (represented by circles) indicate that the system is in a consistent state.

4 The NVP-C(2) Method

In this section we consider the identification of fault patterns and system reconfiguration based on NVP for fault set (2) together with a minimum supporting architecture.

This approach is more complex than the method discussed in Section 3 because of the following reasons. In the NVP method a result is considered correct when the majority of the results agree. However, an erroneous result may be either due to a fault in a software version or to a fault on the PE where a version executes. Therefore, additional means are needed to determine the source of an error. The second reason is that when fault pattern ($S\&H$) appears, the software and hardware faults may occur on the same PE or on the different PEs. These two fault distributions do not appear at the same time as in the RB-based method. The third reason is that several software faults may occur concurrently in different versions executed on different PEs while there is also no such case in the RB-based method.

When several different software versions are executing in parallel on different PEs, the PE or the executing software version is considered suspicious if the result delivered by this PE is different from the result of the majority. As shown later, by assigning two identical suspicious software versions to two PEs and by repeated execution, the cause of erroneous results can be identified as software or hardware faults. As before, reconfiguration of new software versions and PEs is performed in order to maintain the ability to tolerate subsequent faults.

The architecture needed for NVP-C(2) (also for NVP-C(1/1)) is shown in Figure 3. Again this architecture represents the minimum number of versions and PEs that can be used to cover the fault patterns in NVP(2) [12]. Software versions are assigned in such a way that after a software version is identified as faulty, it is made passive and a back-up software version is made active on two PEs, thus double voting can still be performed in repeated executions. In this architecture, four software versions and four PEs are used. On each PE, three versions are back-up versions and all PEs are assigned the same set of versions. The order of activation of each versions on each PE is based on different execution results of versions at each PE. Therefore, the ordering in Figure 3, except for the primary ones (first set of versions), is irrelevant. In Figure 3, the ordering is based on the increasing order of their subscripts. Every time only one version is executed on each PE, while the other versions remain idle until they are activated when necessary. Of the four executing software versions, two of them are identical (called duplicate version, e.g., in Figure 3, initially V_1 is duplicate). After one round of execution, each PE will produce a result. By using the double voting scheme and comparing the result of double voting with the result of each version, the system state can be obtained. The correct result can be obtained if $R_d \neq$ "-". According to the system state some faults can be identified immediately, but to identify other faults

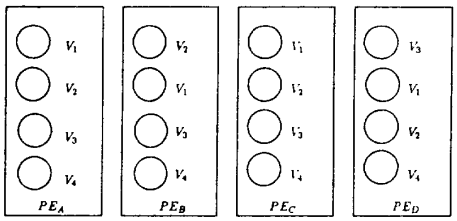

Figure 3: System architecture for NVP-C(2) and NVP-C(1/1)

it requires further reconfiguration of software versions or PEs and repeated execution. Reconfiguration and repeated execution are based on the following rules: (1) If two copies of a duplicate version produce identical results which are different from the result of double voting, R_d, the duplicate version is identified as faulty. (2) If two copies of a duplicate software version and their PEs are suspicious, replace the duplicate version by a new software version selected from the backup pool, and execute the new version on the same PE. (3) If only one copy of a duplicate version and its PE are suspicious, repeat the execution using the same configuration. (4) If a non-duplicate software version and its PE are suspicious, assign also this version to a non-suspicious PE. Thus the suspicious version becomes a duplicate, then execute both copies of the version.

The actions following these rules are repeated until a fault pattern is identified or a correct result is delivered where only partial cover of a fault pattern is needed. The FISR tree for NVP-C(2) is shown in Figure 4.

There are two main differences between the NVP-C(1/1) and the NVP-C(2) methods. In the NVP-C(2) method, the fault patterns (S, S), $(S\&S)$, (H, H), and $(H\&H)$ need to be covered. In addition, whenever a correct result is obtained, no further reconfiguration steps are activated to identify each fault pattern in order to simplify the FISR tree for NVP-C(2) or due to cost and time constraints. If there is a requirement to identify each fault pattern, additional reconfiguration steps can be applied.

In Figure 4 eight cases of fault patterns are possible if the state $(V_{1A} = V_{1C}) \neq V_{2B} \neq V_{3D}$ appears after the execution of $exec(V_{1A}, V_{2B}, V_{1C}, V_{3D})$. These fault patterns cannot be distinguished and no correct result is obtained at this stage. If $exec(V_{1A}, V_{4B}, V_{4C}, V_{3D})$ is applied at the next step, a correct result is delivered. Two fault patterns can be fully covered (leaves K and N) and four fault patterns are partially covered (leaves L and M). However, if the state $(V_{4B} = V_{4C}) \neq V_{1A} \neq V_{3D}$ appears after $exec(V_{1A}, V_{4B}, V_{4C}, V_{3D})$, two fault patterns $(V_1 \& PE_D)$ and $(V_1 \& V_3)$ still cannot be distinguished and no correct result is delivered. So by further applying $exec(V_{2A}, V_{4B}, V_{3C}, -)$ after V_1 and

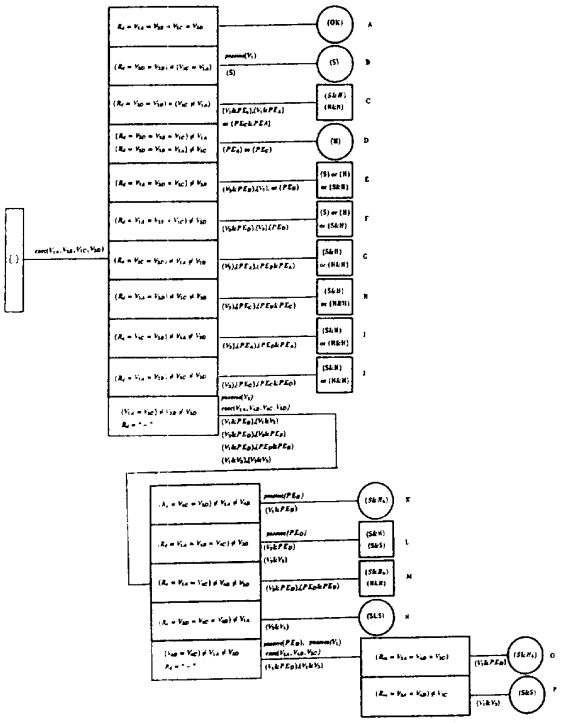

Figure 4: FISR tree for NVP-C(2)

PE_D are made passive, these two fault patterns can be distinguished and we can also deliver a correct result. In Figure 4, the leaves C, E, F, G, H, I, and J are also partially covered because each of them includes two fault patterns which are not distinguished by the steps taken before. So the NVP-C method only partially covers the fault set (2). However the correct result is guaranteed to be obtainable under every different fault distribution.

5 Analysis and Evaluation

The dependability analysis of a system includes concepts such as reliability, maintainability and availability. It serves, along with considerations of cost and performance, as a major system selection criterion. In general, the effects of a fault tolerant design strategy on system reliability can be expressed as follows [15]:

$$R_{system} =$$
$$Pr\{\text{no fault}\} + Pr\{\text{fault pattern/fault}\} * Pr\{\text{fault}\}$$

where the effect of fault tolerance on reliability is represented by the second term. Markov and Markov reward models are commonly used for quantitative reliability evaluation and have been extensively studied [7], [4]. We can concentrate here the analysis based on

the FISR tree, and study the effects on dependability of model parameters such as failure rate λ when applying the proposed method.

5.1 System Reconfiguration Steps (SRS)

We define *system reconfiguration steps* (SRS) as a measure of the number of steps or executions needed to distinguish fault patterns and reconfigure the system to a consistent state when faults occur. The maximum number of SRS is equivalent to the depth of the FISR tree minus one.

$$\text{Max SRS} = depth(FISR) - 1$$

If the probability of fault pattern occurrence can be determined we can also determine the average SRS. Let λ_{FP} be the probability of the occurrence of a fault pattern and $\lambda_{FP_i} = \Pr\{FP_i/FP\}$ be the probability of fault pattern FP_i when a fault pattern occurs. It is clear that (1- λ_{FP}), where $\lambda_{FP} = \sum \lambda_{FP_i}$, is the probability of no fault. Since a fault pattern corresponds to a combination of software and hardware faults, therefore each λ_{FP_i} is a function of the failure rate of each software version and processor. Let *pathlength* (FP_i) be the length of the path in the FISR that leads to a leaf corresponding to FP_i and let *noFP* correspond to the fault free case. Then, we have the following average SRS:

$$\begin{aligned} AveSRS &= (1-\lambda_{FP}) * (pathlength(noFP) - 1) \\ &\quad + \Sigma\lambda_{FP_i} * (pathlength(FP_i) - 1) \end{aligned}$$

Since *pathlength (noFP)* = 1 which represents one execution, therefore there is no reconfiguration step. *Pathlength* (FP_i) - 1 stands for the number of reconfiguration steps of FP_i required to detect fault patterns which is denoted as *reconfigsteps* (FP_i). We have then

$$AveSRS = \Sigma\lambda_{FP_i} * reconfigsteps(FP_i)$$

For example, suppose we have the distribution of failure rate (Table 4) for the FISR tree of RB-C(1/1) shown in Figure 2. We assume that the rate of transient faults is higher than the ra te of permanent faults. The *pathlength* (PL) and *system reconfiguration steps* (SRS) are derived directly from this FISR tree in the figure.

Table 4: Reconfiguration steps of RB-C(1/1)

FP	λ	*PL*	*SRS*
A	0.0001	2	1
B	0.0012	3	2
C	0.0009	3	2
D	0.0005	2	1
E	0.0004	2	1
F	0.0011	2	1
G	0.0007	2	1
OK	0.9966	1	0

The average SRS then can be computed in the following way:

$$\begin{aligned} AveSRS &= .0001*1 + .0012*2 + .0009*2 + \\ &\quad .0005*1 + .0004*1 + .0011*1 \\ &\quad + .0007*1 + .9966*1 = .007 \end{aligned}$$

If all the fault patterns have equal probability of occurrence, the average SRS can be given by:

$$\begin{aligned} \text{Ave SRS} &= \Sigma\lambda_{FP_i} * reconfigsteps(FP_i) \\ &= (\Sigma reconfigsteps(FP_i)) * \lambda_{FP}/\text{total} \\ &\quad \text{number of fault patterns} \end{aligned}$$

In this case the average SRS of the above example is calculated as (suppose λ_{FP} is 0.0034).

$$\text{Ave SRS} = (1+2+2+1+1+1+1)*0.0034/7 = .004$$

5.2 Scheme and Parameter Selection

In order to use the proposed model one must first select the related parameters. These parameters can be grouped into the following tuple:

$$(S, C, M, F)$$

where S stands for scheme, C for fault set coverage, M for fault detection mechanisms and F for fault set. The structure of the FISR tree depends strongly on the selection of the above four parameters. The selection of a specific parameter in general depends on the selection of the remaining parameters. For example, if S, C, and F are given, how one finds a minimum or cheapest M ? Here we focus only in the selection of the parameters in F. As defined in Section 2, (m/n) stands for a maximum m software and n hardware faults within a reconfiguration period, while (m) stands for a maximum of m faults (either software or hardware) within a reconfiguration period. A reconfiguration period can be measured by the longest time used from the root of the FISR to one of the leaves. A misestimation of the m and n in F would have the following effects:

- An underestimation of m and n will reduce the fault coverage and fault diagnosability of the system. Some faults that occur within a reconfiguration period will remain undetected and propagate to the next reconfiguration period, which will further reduce the fault coverage and fault diagnosability of the system.
- An overestimation of m and n will increase the cost. More resources (either hardware or software) will be used than necessary.

In general, the values of m and n are proportional to the components' failure rates λ, and to the length

of each reconfiguration period, the depth of the FISR tree (which in turn depends on the execution time of the application), repair rates, etc.

Note that throughout this paper, we only concentrate on fault patterns with a maximum of two faults within a reconfiguration period. Theoretically, the identification and reconfiguration procedure can be applied to any number of faults and the FISR tree can be used to analyze the procedure. However, because of time and cost constraints, especially in real-time applications, the proposed approach is efficient only when a small number of faults is considered. This problem can be alleviated by increasing the rate of reconfiguration period, and therefore potentially the system can tolerate more faults. In addition, if the fault pattern cannot be covered within the time limit, system failure occurs. In this case other mechanisms may be used to recover the system which are beyond the scope of the proposed scheme. The FISR tree itself can become very large if a larger number of faults are involved.

6 Conclusions

We have presented a uniform software-based fault tolerance method to distinguish and tolerate various sequences of software and hardware faults. The method can be based on recovery blocks or n-version programming. A special tree called fault identification and system reconfiguration (FISR) tree was proposed to represent the procedure of fault identification and system reconfiguration.

There are applications that require a very high level of reliability, e.g., flight control systems, nuclear reactor controls, and others. This approach should be valuable as another tool for the design of ultra-reliable systems. As a general research direction this approach is also valuable in showing how hardware and software fault tolerance mechanisms can be combined in a harmonious way.

The problems of how to consider correlated errors among software versions, acceptance tests with no perfect coverage, and how to find a general method to reconfigure the system still needs to be studied. However we believe this is a step in the direction of solving the complex problem of combined software/hardware faults.

References

[1] A.Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. on Software Engineering.* 11, (12), Dec.1985, 1491-1501.

[2] B.Randell. System structure for software fault tolerance. *IEEE Trans. on Software Enginerring.* 1, (2), June 1975, 220-232.

[3] D.E.Eckhardt and L.D.Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Trans. on Software Engineering.* 11, (12), Dec. 1985, 1511-1517.

[4] D.I.Heimann, N. Mittal, and K.S. Trivedi. Availability and reliability modeling for computer systems. *Advances in Computers.* 31, edited by M.C.Yovits, Academic Press, 1990, 692-702.

[5] F.Cristian. Understanding fault-tolerant distributed systems. *Communication of ACM.* 34, (2), Feb. 1991, 57-78.

[6] G.Pucci. A new approach to the modeling of recovery block structures. *IEEE Trans. on Software Engineering.* 18, (2), Feb. 1992, 159-167.

[7] J.Arlat, K.Kanoun, and J.C.Laprie. Dependability modeling and evaluation of software fault-tolerant systems. *IEEE Trans. on Computers.* 39, (4), April 1990, 504-512.

[8] J.C.Knight and N.G.Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Trans. on software Engineering.* 12, (1), Jan. 1986, 96-106.

[9] J.C.Laprie, J. Arlat, C. Beounes, and K. Kanoun. Definition and analysis of hardware and software fault-tolerant architectures. *IEEE Computer.* 23, (7), July 1990, 39-51.

[10] J.H.Lala and L.S.Alger. Hardware and software fault tolerance: A unified architectural approach. *Proc. of 18th FTCS.* 1988, 240-245.

[11] J.Kelly, T.McVittie, and W.Yamamoto. Implementing design diversity to achieve fault tolerance. *IEEE Software.* July 1991, 61-71.

[12] J.Wu, Y.W.Wang, and E.B.Fernandez. A uniform approach to software and hardware fault tolerance. *TR-CSE-91-1, Department of Computer Science and Engineering.* Florida Atlantic University, 1991.

[13] M.Hecht, J.Agron, H.Hecht, and K.H.Kim. A distributed fault tolerant archecture for nuclear reactor and other critical process control applications. *Proc. of 21th FTCS.* 1991, 462-470.

[14] D. Tang and R.K.Iyer. Analysis and modeling of correlated failures in multicomputer systems. *IEEE Trans. on Computers.* 41, (5), May 1992, 567-577.

[15] V.P.Nelson. Fault-tolerant computing: fundamental concepts. *IEEE Computer.* 23, (7), July 1990, 19-25.