BI-LEVEL RECONFIGURATIONS OF FAULT TOLERANT ARRAYS IN BI-MODAL COMPUTATIONAL ENVIRONMENTS

Rami G. Melhem

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT

Fault-tolerant architectures and algorithms are studied for processor arrays which are subject to computational loads that alternate between two phases. A *strict* phase which is characterized by a heavy load and strict constraints on response time, and a *relaxed* phase which is characterized by a light load and relatively relaxed constraints on response time. Under this type of loads, a bi-level algorithm may be applied to reconfigure the system after faults. Specifically, at one level, called the fast response level, a local distributed fault-tolerant algorithm is used during strict phases to achieve fast fault recovery at the expense of possible rapid degradation in the potential to tolerate future faults. In order to minimize the effect of this degradation, a second level, called the optimization level, is added. At that level, a global, relatively slow, reorganization algorithm is applied during relaxed phases to restore the system into a shape that ensures adequate fault tolerance capability in the remaining part of the system's mission.

Three examples are given for bi-level reconfiguration algorithms which emphasize three different restoration criteria. In the first example, the goal of the optimization level algorithm is to maximize the probability of survival of the system. In the second example, the goal is to maximize the potential of fault detection and correction in the system, and in the third example, the goal is to guarantee the locality of fault coverage during subsequent strict phases of the system's operation.

1. INTRODUCTION

Computing arrays that are composed of a large number of processors provide a computational power which is comparable to, and potentially larger than, the computational power of supercomputers. One way of providing such arrays with adequate fault tolerance capabilities is to include in the array a number of spare processors that are capable of taking over the tasks of processors that fault at run-time. Many algorithms have been suggested for the run-time reconfiguration of different processor arrays architectures. These algorithms may be classified according to the amount of knowledge needed to recover from a fault. On one extreme, some algorithms require a knowledge of the state of the entire system in order to successfully recover from a fault, while, on the other extreme, some algorithms require only a knowledge of the states of the faulty processor and its immediate neighbors. In many cases, the acquisition of more knowledge about the system leads to more robust algorithms and more efficient utilization of the available redundancy.

In addition to the above classification, it is possible to classify array reconfiguration algorithms according to the changes in the configuration required to recover from faults. In order to allow for fast reconfiguration with very little interruption in service, it is desirable that the coverage of a faulty node causes very few changes in the system interconnections and in the assignment of tasks to nodes. The modular approach [5, 18] and the roving approach [17], for example, require extremely local reconfiguration and very few node reassignment. Only the faulty node and its immediate neighbors need to be aware of any change in the configuration. However, in such systems, the nodes are typically partitioned into groups, and each group is assigned some specific number of spares. A spare may only be used to cover for a node in its assigned group or at most in a neighboring group (as in [18]). This inflexibility reduces the fault coverage capability and thus the reliability of the entire system.

In this paper, two types of algorithms will be considered. Namely *local algorithms* and *global algorithms*. In a local algorithm, no processors need to know the status of all other processors in the system. The recovery process is distributed among the processors with each processor using extremely local knowledge. Moreover, only the neighbors of a faulty processor need to take some corrective actions. With these properties, the reconfiguration algorithm may achieve fast recovery and real time response, but may sacrifice the optimal use of redundancy. In contrast, the goal of a global algorithm is to optimize the use of redundancy with respect to some fault tolerance criteria. This, however, requires global knowledge about other processors in the system and often necessitates extensive changes in the configuration of the system.

For unmaintained, long-life systems, local fault tolerance algorithms have the advantages of fast recovery and real time response. Global fault tolerance algorithms, on the other hand, provide better reliability and longer life expectancy, but often cannot meet real time requirements. Fortunately, under certain conditions, it is possible to combine the advantages of the two types of algorithms. These conditions are described in the next Section.

2. BI-MODAL COMPUTATIONAL ENVIRONMENTS

As suggested by its name, a bi-modal computational environment is defined to be one which alternates between two modes that have different characteristics and different computational requirements. The two modes considered in this paper are called the "*strict mode*" and the "*relaxed mode*". The *strict mode* is characterized by a heavy computational load and a demand for fast response within severe time constraints, while the *relaxed mode* is characterized by a light computational load with relatively relaxed constraints on the response time. A computing system subject to such a bi-modal load, thus, alternates between strict phases and

0731-3071/89/0000/0488$01.00 © 1989 IEEE

relaxed phases, and it is reasonable to assume that the duration of each of the two phases is not fixed or predictable. For example, an advanced radar system may be considered in a relaxed phase as long as the system does not detect any target within its range. However, at any instant, a detected target may carry the system into a strict phase in which the nature of the target as well as its speed, direction and destination should be determined as quickly and reliably as possible.

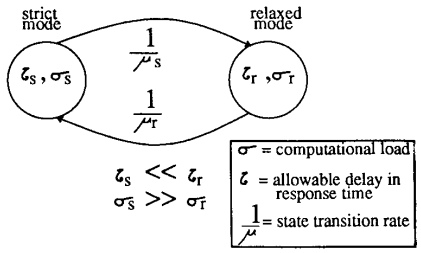


Figure 1 - A state diagram model for bi-modal computational environments

A state transition diagram for bi-modal computational environments is shown in Figure 1. A crucial distinction between the two modes is that the computational load for the strict mode, σ_s, is much larger than the load for the relaxed mode, σ_r (σ_s and σ_r are measured in some appropriate units, as for example MFLOPS or MIPS). Another distinction concerns the allowable delay in response time, τ. Specifically, it is assumed that the computing system is subject to run-time faults and that such faults are reparable on-line. However, repairing a fault is assumed to consume some time during which the system is not available to advance the status of the computation. In other words, the response time of the system is delayed, for each fault, by the time needed to repair that fault. In the model of Figure 1, it is assumed that allowable delays in response times are much smaller in strict phases than in relaxed phases. Note that it is possible, though not necessary, for τ and σ to be related in the sense that larger σ implies smaller τ. For example, in air traffic controller systems, heavier traffic necessitates faster responses because of the higher accuracy and more timely information required to manage the traffic.

The average durations of the strict modes and the relaxed modes are assumed to be μ_1 and μ_2, respectively. That is, the average transition rates from the strict mode to the relaxed mode, and from the relaxed mode to the strict mode are assumed to be $1/\mu_1$ and $1/\mu_2$, respectively. No assumptions are made about the values of μ_1 and μ_2 or about the ratio μ_1/μ_2.

Computing systems emphasized in this paper consist of ensembles of computing elements where some elements are initially designated as primary elements that contribute to the progress of the computation and other elements are designated as spares for the primary elements. If, during the system's mission, a primary element becomes faulty, then an on-line reconfiguration algorithm is invoked to replace the faulty element by a spare. Fault-tolerant two-dimensional arrays [8, 9, 12], tree architectures [6, 10] and augmented hypercube architectures [3, 14], are examples of such systems.

Let n be the number of primary elements in the computing system and assume that the reliability of an element is given by $e^{-\lambda t}$. Hence, if faults in elements are independent of each others, then faults in primary elements will occur at a rate of $\Lambda=n\lambda$ and, on the average, the system will need reconfiguration every $1/\Lambda$ time units. In systems subject to bi_modal computational environments, it is possible for Λ to be larger during strict modes than during relaxed modes. This may be due to harsher operating conditions (as in systems subject to military attacks), or to high utilization (which may lead to the emergence of hidden software and hardware errors). For this reason, Λ_s and Λ_r will be used for the fault rates during strict and relaxed phases, respectively.

3. BI-LEVEL RECONFIGURATION ALGORITHMS

As mentioned earlier, a fault in a primary element of a computing array causes the invocation of an on-line reconfiguration algorithm which disables the array temporarily. The state of the system during reconfiguration is modeled in Figure 2 by the two states denoted "reconfig. phase". The transition rates from the strict mode and the relaxed mode to the reconfiguration phase are Λ_s and Λ_r, respectively. The rate of transition out of the reconfiguration phase is $1/\rho$, where ρ is the average time needed for reconfiguration.

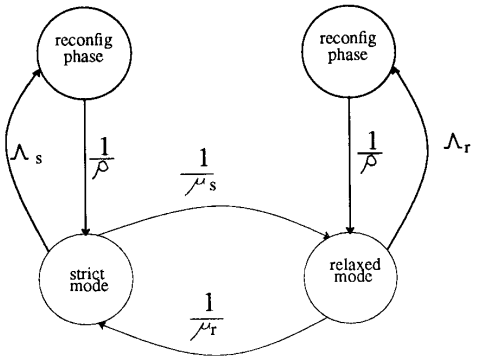


Figure 2 - Array reconfiguration in bi-modal environments

Clearly, the time, ρ, consumed by the reconfiguration algorithm should be of the order of τ_s to meet the response time constraints during the strict phases of the computation. Such a demand for fast and correct responses may require the application of extremely local and distributed reconfiguration strategies because global and centralized strategies may not meet the time constraints imposed on the system. However, local/distributed reconfiguration algorithms do not take into consideration the state of all the elements in the system, and thus, in many cases, do not use the redundancy of the system in the most efficient way. Such underutilization of redundancy may be the cause for rapid reliability degradation and reduced life expectancy.

The basic idea in bi-level reconfiguration algorithms is to apply some fast, local, distributed algorithm when the system is in strict modes. In this mode, the reconfiguration time, ρ_L, is constrained by τ_s. Whenever the system enters a relaxed mode, a global, centralized reorganization (clean-up) algorithm is applied to restore the system into a shape that ensures adequate fault tolerance capability in the remaining portion of its mission. The global reconfiguration time, ρ_G, can be afforded during relaxed phases because it is constrained by τ_r which is much larger than τ_s.

Global algorithms applied during relaxed modes, however, should not make the system un-available (or reduce drasticly its computational capability) for prolonged period of times. This is

because, at any instant, the system may be transferred from a relaxed mode to a strict mode. Hence, global reorganization should proceed in short incremental steps, each meeting the strict mode constraint τ_s. Between any two steps, the system should be in a consistent state and be ready to resume computation at full capacity. For example, if a centralized controller decides to reassign the task of each processor node in the system to a different node, then reading the state of all the nodes and then writing them back to the reassigned nodes may not be acceptable because it may incapacitate the system for an extended period of time. An alternative strategy is to complete the reassignment by performing, if possible, a sequence of pair-wise node interchanges. Such strategy will be acceptable if the time for each interchange is bound by τ_s and after each interchange, the system is ready to resume computation.

Given that the average durations of strict phases is μ_s and that the average fault rate during strict modes is Λ_s, the average number of faults that occurs during a strict phase is $\phi_s = \mu_s \Lambda_s$. Similarly, the average number of faults that occur during a relaxed phase is $\phi_r = \mu_r \Lambda_r$. The values of ϕ_s and ϕ_r may be used to approximate the frequency of global reconfiguration in bi-modal algorithm. Specifically, if $f_1, f_2, f_3, \cdots$ is a sequence of faults and $\phi = \phi_s + \phi_r$, then it may be assumed that global reconfiguration is invoked after each fault f_k, where $k = i\phi + \phi_s, i\phi + \phi_s + 1, \ldots, (i+1)\phi$, for $i = 0,1,\ldots$. For example, if $\phi_s = 3$ and $\phi_r = 2$, then global reconfiguration is invoked after f_3, f_4, f_5, f_8, f_9, f_{10}, f_{13}, and so on. In the following sections, simulation studies will be simplified by assuming that the frequency of global reconfiguration, denoted by RF, is constant for the life-time of the system. Such an assumption may be justified if $\Lambda_r \ll \Lambda_s$ or $\mu_r \ll \mu_s$, thus leading to $RF \approx \phi_s$. If this is not the case, then a constant RF may be regarded as the average frequency of global reconfiguration, namely $(\phi_s + \phi_r)/(\phi_r + 1)$. It should be also noted that if $\phi_s \leqslant 1$ and $\phi_r \leqslant 1$, then $RF = 1$.

In the next sections, we present three examples of fault tolerant systems that may benefit from bi-level reconfiguration algorithms. The examples emphasize three different optimization criteria and illustrate the advantages of the bi-level concept

EXAMPLE 1: MAXIMIZING LIFE-TIME IN FAULT TOLERANT HYPERCUBE ARCHITECTURES

Hypercube multiprocessor systems are being used for a variety of scientific applications and dedicated real time systems. However, as the number of processors in a hypercube system increases, the complexity of the system increases, thus leading to possible high failure rates. For applications where degraded performance is not allowed it is necessary to reassign the tasks of the failing processors to some spare processors without destroying the logical hypercube interconnection among the functioning processors. One approach to achieve fault tolerance in hypercubes is to initially designate only some of the processors as active and designate the rest as spares that may cover for faulty active processors [13]. Another approach is to decompose the hypercube structure hierarchically and add redundancy at several levels [4, 14, 15]. In this section we consider a yet different approach in which an N node hypercube is augmented with E extra nodes that are to be used as spares [2, 3]. From the application point of view, only N nodes are used and messages between them are interchanged by using the usual $\log N$ hypercube addresses. When a node fails, one of the spare nodes takes over its task and inherits its address. From this point on, messages addressed to the failed node should be routed to the spare node that replaces it.

For example, consider a 4-dimensional binary cube which is augmented with 8 spare nodes such that each spare is added to some face (4 nodes) of the cube and is connected to all the nodes in that face. Such an augmented cube is shown in Figure 3 where Figure 3(a) shows the interconnections among the primary nodes and among the spare nodes, and Figure 3(b) shows the interconnections between the primary and the spare nodes (spare nodes are drawn using bold lines). In accordance with Figure 3, a primary node will be referred to by a tuple (i,j), where $1 \leqslant i,j \leqslant 4$, and a spare node will be referred to by $(0,j)$, or $(i,0)$. The 24 nodes form a fault tolerant basic block (FTBB) in which each primary node (i,j) may be covered by one of two spares, namely $(0,j)$ or $(i,0)$.

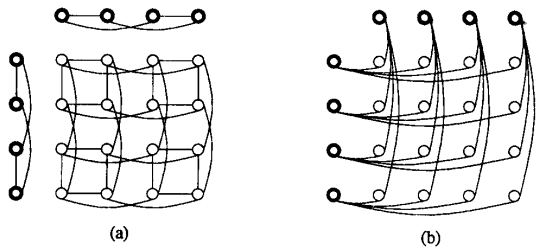


Figure 3 - A 4-dimension augmented hypercube

An n-dimensional hypercube may be constructed from $n-4$ FTBB's by connecting, for every i and j, the corresponding $n-4$ nodes (i,j) as a hypercube. Message routing in such augmented hypercube architectures [2] will not be discussed here. Rather, the primary concern will be the reliability of the system. Specifically, system failure is defined as the failure of a node (i,j) while neither spare $(i,0)$ nor spare $(0,j)$ is available to cover for (i,j).

A local and distributed replacement policy in an FTBB has to attempt to replace a failing node by one of its two spares in some order. For example, if (i,j) fails then the replacement policy may be to first try to replace (i,j) with $(i,0)$ and, only if $(i,0)$ is not available, try to replace (i,j) with $(0,j)$. Alternate policies are to chose the order in some arbitrary fashion, or to try $(i,0)$ first if $i<j$, and try $(j,0)$ first otherwise. All these policies are fast to implement because they do not have to investigate the status of all the other nodes. However, there are cases where, by looking at the global status of the FTBB, an intelligent decision may be made regarding the replacement which increases the probability of survival of the system.

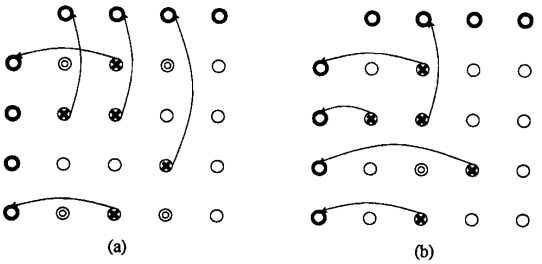


Figure 4 - (a) Local Vs. (b) global fault coverage in the cube of Figure 3

We clarify this by an example. Assume that the nodes (2,1), (2,2), (3,3), (1,2) and (4,2) failed in that order. If spare $(i,0)$ is always tried before $(0,j)$, then the configuration after the 5 faults is as shown in Figure 4(a). By studying that figure more closely we find that 4 of the 11 non-faulty primary nodes have both their

spares unavailable. Such nodes are called uncovered nodes and are denoted in Figure 4 by double circles. In other words, if each of the 11 non-faulty primary nodes have equal probability of failure, then the probability that the system will survive one more fault in a primary node is 7/11. In Figure 4(b), an alternative coverage is shown where only one node is left uncovered. This means that, with the reconfiguration of Figure 4(b), the probability of surviving one more fault in a primary node is 10/11.

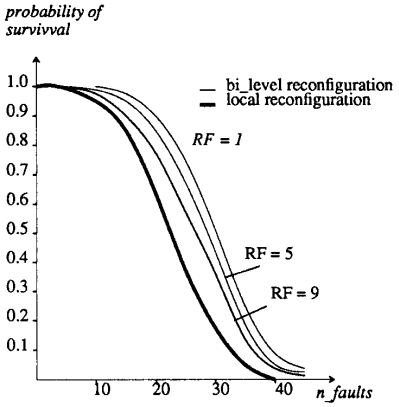


Figure 5 - Probabilities of survival of 7-dimensional augmented hypercubes

It is straight forward to write a recursive algorithm which, given a certain fault configuration, exhaustively finds all possible fault-to-spare assignments and choses the one that minimizes the number of uncovered nodes. The effect of such global reconfiguration on the life-time of the system have been studied by considering two d-dimensional augmented cubes (each having 2^d primary nodes and 2^{d-1} spares) and simulating both systems under randomly generated sequences of faults. Local reconfiguration have been applied to both systems after each fault, but global reconfiguration have only been applied to the second system every RF faults (stands for *reconfiguration frequency*). For each sequence of faults, the number of node failures that causes system failure may be considered as an indication of the life-time of the system. By repeating the above experiment for 1000 different sequences, an estimate have been obtained for the probability of each system to survive a specific number of faults, *n_faults*. The results of such simulation are shown in Figure 5 for 7-dimensional augmented cubes assuming RF = 1, 5 and 9 and following the local policy of trying spare $(i,0)$ or $(0,j)$ first depending on whether $i<j$ or $j \leqslant i$, respectively. Clearly, a remarkable improvement in the life-time of the system results from bi-level reconfiguration.

In the above discussion, a spare was not allowed to fail, an assumption which is not realistic. In order to allow for spare failure, one more spare is added to each FTBB. This spare is referred to by (0,0) and is used to cover for the failure of any one of the nodes $(k,0)$ and $(0,k)$, k=1,..,4. Simulations similar to the one described above was conducted for d-dimensional systems (in this case the number of spares is $2^{d-1}+d-4$). The results of these simulations are shown in Figure 6 for d=7. In this figure, the probability of survival is given in terms of the number of faults *n_faults*. This number is related to the time t by *n_faults* = Λt, where $\Lambda=2^d\lambda$.

Finally, we note that the goal of the global reconfiguration described in this section is to minimize the number of uncovered nodes in the systems. Exhaustive search techniques were applied to

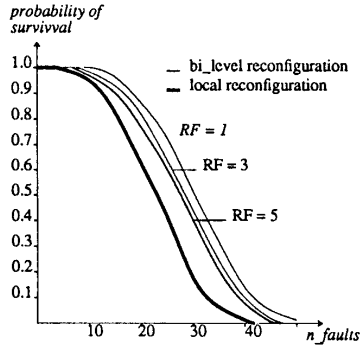
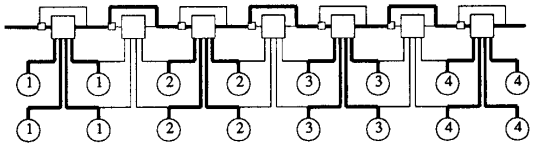


Figure 6 - Reliability of 7-dimension hypercubes with one additional spare per FTBB

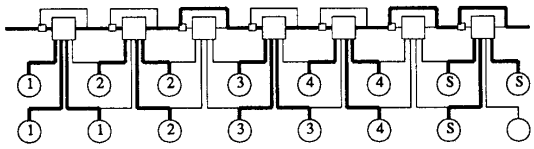
solve this minimization problem. It is not quite obvious that faster solutions do not exist for this problem. Moreover, it is interesting to study fast heuristics that may approximate the solution. Specifically, any algorithm that decreases the number of uncovered nodes is expected to improve the reliability of the system.

EXAMPLE 2: RESTORING ERROR CORRECTION CAPABILITY IN NMR LINEAR ARRAYS

Most research in fault tolerant computational arrays have concentrated on fault coverage (see for e.g. [1, 12, 16]) and only few research efforts have been directed toward fault detection and correction in such arrays. The roving spare technique [17] and the weighted checksum coding [7] are two approaches that may be used to detect (and in some cases correct) transient or permanent faults. However, in both cases, a latency period may elapse before faulty processors are detected. If the processor array is subject to severe recovery time constraints, or if the production of faulty results (due to delayed error detection) may be disastrous, then the use of N modular redundancy (NMR) seems to be appropriate.



(a) With 4 module redundancy



(b) With TMR and a spare

Figure 7 - A 4-node linear array

In the context of computational arrays, NMR may be achieved by replicating each node in the array N times (each replica will be called a module). The input to a node is directed to its N modules, and the output of the node is computed by taking a majority vote among the outputs of the N modules. For example,

In Fig 7(a), we show a linear array of 4 nodes, each composed of 4 modules. The array is constructed by embedding it into a reconfigurable architecture in which every four modules are connected to a switching element that may perform majority voting. Bold and non-bold lines are used in Figure 7 to indicate active and non-active connections, respectively. With a 4-way comparison at the switches, it is possible to locate any single faulty module in a node and to exclude it from that node. Clearly, a node with three modules may still detect and correct a fault in any of its modules. However, a node with two modules may only detect a fault. Because of its inability to correct a detected fault, such a node is declared faulty and consequently the entire system is declared faulty.

We have just described a *local redundancy* method for increasing the reliability of the array by using redundant hardware to increase the reliability of each node. For instance, in Fig 7(a), a failure of two of the 4 modules constituting a node may be tolerated as long as failures do not occur simultaneously. Assuming that the reliabilities of the modules are independent and the reliability of each module is given by $r(t)=e^{-\lambda t}$, then the reliability of an m-node array is given by

$$R_m(\lambda t) = \left(\sum_{i=2}^{4} \binom{4}{i} r^i \, (1-r)^{4-i} \right)^m \qquad (1)$$

where λt is used as the independent variable to avoid separate specification of the time t and the module fault rate λ. If $k=4m\lambda t$, where $4m$ is the total number of modules, then $R_m(\lambda t)$ is the probability of system survival after k faults. Note here that system failure is declared as soon as any node looses its ability to detect faults. This coincides with the detection of an un-correctable fault.

The local redundancy technique described above is not the only way of allocating redundant hardware in a computational array. Another technique, called *horizontal redundancy*, is to use only three modules for each node and to employ the remaining modules to form s spare nodes, each of which is also composed of three modules. Spare nodes are to be used at run time to replace failing nodes. For example, the system of Fig 7(a) may be reconfigured as shown in Fig 7(b) where a spare node is available to replace any node that becomes faulty at run time. The reliability of the array in such case is given by:

$$\hat{R}_{m,s}(\lambda t) = \sum_{i=m}^{m+s} \binom{m+s}{i} p_{2,3}^i \, (1-p_{2,3})^{m+s-i} \qquad (2)$$

where

$$p_{2,3}(\lambda t) = \sum_{i=2}^{3} \binom{3}{i} r^i \, (1-r)^{3-i}$$

If the total number of modules in the above two redundancy techniques is approximately the same, that is if $4m=3(m+s) \pm 2$, then the two reliability formulas (1) and (2) can be compared. For example, for $m=4$ and $s=1$, it may be shown that $R_m(\lambda t) < \hat{R}_{m,s}(\lambda t)$ if $\lambda t \leq 0.35$, and $R_m(\lambda t) > \hat{R}_{m,s}(\lambda t)$, otherwise.

Although horizontal redundancy may, in some cases, improve the overall reliability of the array, its implementation may require global reallocation of modules after a fault. In strict operational modes, such global reallocation and any associated transfer of programs and data may be prohibited. The local redundancy technique does not suffer from such a drawback. It only requires the reconfiguration of the switch associated with the faulty node. As will be described next, it is possible to design a bi-level reconfiguration algorithm which takes advantage of local redundancy during strict phases and applies a global optimization during relaxed phases to redistribute the redundant modules uniformly among the nodes of the array. This increases the reliability of the entire array.

Bi-modal reconfiguration of NMR processor arrays.

Given $4m$ modules, the basic idea in the bi-modal reconfiguration presented in this section is to initially configure an m-node array by grouping every four modules into a node in a manner similar to that shown in Figure 7(a). As described earlier, faulty modules are excluded from the nodes as soon as they are detected. However, when a node is left with only two non-faulty modules, that node looses its fault correction capability and thus leaves the entire system susceptible to failure. In this case, the array is globally reconfigured such that, if possible, at least three non-faulty modules contribute to each node. The global reconfiguration should be performed when the system operates in a relaxed mode. Consider for example, the same array shown in Figure 7. If the 3 modules marked by an X in Figure 8(a) fail, then the second node in the array will have only two non-faulty modules. A global reconfiguration of the array is shown in Figure 8(b), where each node consists of three non-faulty modules, and thus may correct any future fault.

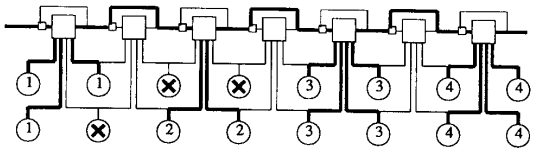


Figure 8(a) - The array of Figure 7 after 3 faults

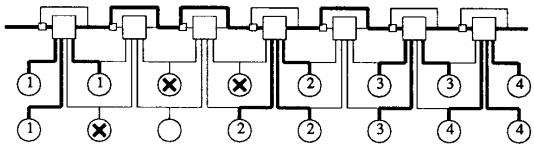


Figure 8(b) - Global reconfiguration of the array of Figure 8(a)

Let τ be the time at which the global reconfiguration takes place. Because of the memory-less property of exponential distributions, the reliability of any non-faulty module at time $t \geq \tau$ is given by $r_\tau(t) = e^{-(t-\tau)\lambda}$. Hence, given the configuration of the system at τ, its reliability at any time t, $t \geq \tau$, may be calculated. Specifically, for the system of Figure 8(a), that reliability is

$$R_{m,\tau}(\lambda t) = p_{2,2} \; p_{2,3} \; p_{2,4}^2 \qquad (3)$$

where

$$p_{k,q} = \sum_{i=k}^{q} \binom{q}{i} r_\tau^i \, (1-r_\tau)^{q-k}$$

is the probability of having at least k non-faulty modules in a node composed of q modules. Similarly, for the globally reconfigured system of Figure 8(b), the reliability is

$$\bar{R}_{m,\tau}(\lambda t) = p_{2,3}^4 \qquad (4)$$

From (3) and (4), it is straight forward to check that the system of Figure 8(b) is more reliable than that of Figure 8(a). In fact, it may be shown that for any distribution of faults in an m-node

array, it is very unlikely that the restoration of the fault correction capability (the assurance that each node has at least three modules) will lead to decreased reliability at any future time t. In order to be more specific, let $n\_fault=4m\lambda\tau$ be the number of faulty modules in the array at time $t=\tau$ and assume that global reconfiguration is applied at time τ. If $\bar{R}_{m,\tau}(\lambda t)$ and $R_{m,\tau}(\lambda t)$ are the reliabilities of the array with and without global reconfiguration, respectively, then the following may be proved [11]:

Lemma: There exists a t_c such that

$$\bar{R}_{m,\tau}(\lambda t) > R_{m,\tau}(\lambda t) \qquad \tau \leqslant t \leqslant \tau + t_c$$

Moreover, $\lambda t_c > 0.35$ and, at $t = \tau + t_c$, $R_{m,\tau}(\lambda t) < 0.4$.

We have written an exhaustive search algorithm for the global reconfiguration of any m-node array which consists of $4m$ modules connected by switches as shown in Figure 7. Given the restrictions imposed by the interconnection and the possible switch settings, the objective of the algorithm is to maximize the error correction capability, that is, to minimize the number of two-module nodes in the array. This optimization level algorithm was used to answer two basic questions. First, what is the probability of successfully restoring the error correction capability in an NMR linear array when this property is lost?, and second, how is the reliability of an NMR array affected when bi-level reconfiguration is applied with the objective of maximizing the error correction capability.

n_faults	$m = 10$		$m = 12$		$m = 16$	
	PL	*PR*	*PL*	*PR*	*PL*	*PR*
1	0	0	0	0	0	0
2	112	112	65	65	58	58
3	253	253	189	189	137	137
4	356	356	342	342	247	247
5	624	624	541	541	435	435
6	753	736	675	674	587	587
7	934	717	822	815	702	702
8	950	353	914	792	765	765
9	1000	84	973	508	874	869
10	1000	15	991	238	966	937
11	1000	0	999	41	983	795
12	1000	0	999	6	994	503
13	1000	0	1000	0	998	221
14	1000	0	1000	0	998	67
15	1000	0	1000	0	1000	6
16	1000	0	1000	0	1000	0

Table 1 - Restoration of error correction for 1000 random configurations of *n_faults* faults (PL = property lost, PR = property restored).

In order to answer the first question for a specific number of faults, *n_faults*, and number of nodes, m, *n_faults* $\leqslant m$, a 1000 different distributions of *n_faults* faulty modules was generated randomly. For each fault distribution, the reconfiguration algorithm was invoked if some node in the array had more than one faulty module, and the reconfiguration was called successful if it produced an array with at least three modules per node. The results of the experiments are reported in Table 1 for m =10, 12 and 16. For each value of m and *n_faults*, two numbers are reported. The first, *PL* (*property lost*), is the number of cases (out of 1000) in which the error correction capability were lost and reconfiguration were invoked. The second number, *PR* (*property restored*), is the number of cases (out of *property lost*) in which the reconfiguration was successful. For low values of *n_faults*, reconfiguration was always successful when called, and, for *n_faults* $> m$, successful reconfiguration is not possible because The number of non-faulty modules in this case is less than $3m$.

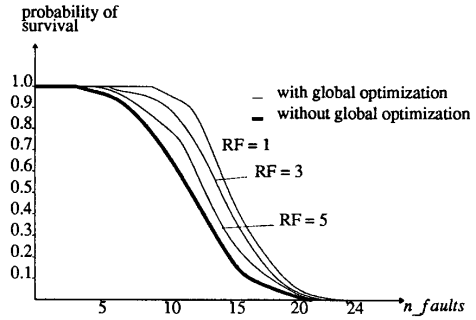


Figure 9 - life-time of a 12-node NMR array

The effect of bi-level reconfiguration on the reliability of the array have been studied via a discrete event simulation similar to the one used in Section 4. Specifically, the effect of sequences of random faults on NMR arrays with and without global reconfiguration was observed and, for each sequence, the number of faults that caused the system to fail (loose its fault detection capability) was recorded. These numbers was then used to compute the probabilities of the systems to survive specific number of faults. Figure 9 shows the probability of surviving *n_faults* faults for m=12, where as before, the parameter *RF* represents the frequency of global reconfiguration. As expected, bi-level reconfiguration is more effective when global optimization is invoked frequently. As *RF* increases, the performance of the bi-level reconfiguration approaches that of the local reconfiguration.

EXAMPLE 3: MINIMIZING FUTURE RECONFIGURATION DELAYS

The problem that we consider in this section is the embedding of a ring (or a linear array) into an n-dimensional hypercube such that adjacent nodes in the ring are mapped to neighboring nodes in the hypercube. We are interested in finding an embedding E that does not occupy all of the hypercube nodes, thus designating some nodes as spares. For fast fault coverage, we require that if ν_{j-1}, ν_j and ν_{j+1} are three consecutive nodes in the ring (+ and - are modulo the size of the ring), then these nodes are mapped to three neighboring nodes $E(\nu_{j-1})$, $E(\nu_j)$, $E(\nu_{j+1})$ in the cube such that there exist a spare node, s_j, in the cube that is a neighbor to both $E(\nu_{j-1})$ and $E(\nu_{j+1})$. This ensures that if node $E(\nu_j)$ becomes faulty, then it may be locally (and thus quickly) covered by s_j. Only $E(\nu_j)$ need to be remapped to s_j and only $E(\nu_{j-1})$ and $E(\nu_{j+1})$ need to know about the fault. This property is called the local coverage property. Along the same line of reasoning, the node $E(\nu_j)$ is said to enjoy the quasi-local coverage property if the recovery of a fault in $E(\nu_j)$ requires that both $E(\nu_j)$ and one of its neighbors, say $E(\nu_{j+1})$, be remapped to two new nodes. In this case, only nodes $E(\nu_{j-1})$ and $E(\nu_{j+2})$ need to be aware of the fault.

Let each of the 2^n nodes in the hypercube be given an n-bit address such that the addresses of any two neighboring nodes across dimension i, i=1,...,n, differ only in the i^{th} bit. The embedding of a ring (or a linear array) of length l may be given in terms of a sequence $\{\nu_1, \cdots, \nu_l\}$ of l nodes, where each two nodes in the sequence, as well as ν_l and ν_1, are neighbors across a given

dimension. Alternatively, the embedding may be specified in terms of a sequence $\{d_1, \cdots, d_l\}$ of dimensions, such that nodes ν_i and ν_{i+1} (modulo l) are neighbors across dimension d_i. This second specification sequence, called from now on the embedding sequence, is more general than the first because it leaves the starting node ν_1 unspecified. For example, the ring shown in Fig 10(a) has the following embedding sequence {1,2,3,4,1,2,3,4}.

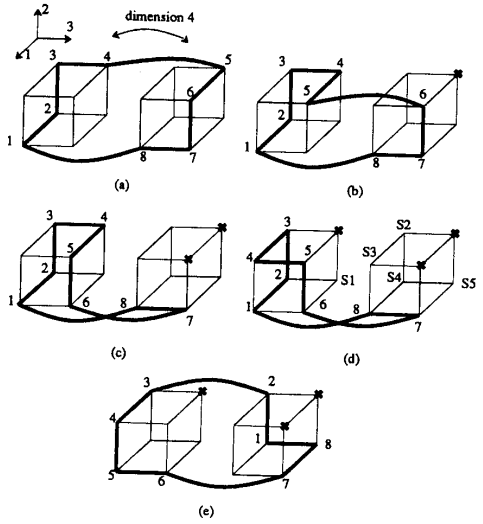


Fig 10 - Fault tolerant embedding of an 8-node ring into a 16-node hypercube

The number of faults that may be tolerated in the system depends on the number of nodes in the cube designated as spares. For example, it may be shown [13] that in order to tolerate any single fault (and many multiple fault configurations) in the manner described above, 25 percent of the nodes in the cube should be designated as spares. Similarly, in order to tolerate any combination of two faults, half of the nodes in the cube should be designated as spares. Specifically, consider the embedding sequence generated from the following algorithm in which the notation $x \mid y$ denotes the sequence resulting from the concatenation of the two sequences x and y:

$x = \{1,2,3\}$
FOR $i = 4,\ldots,n$ DO
    $x = x \mid \{i\} \mid x$
$x = x \mid \{n\}$

It may be proved that the embedding derived from this sequence may tolerate any single fault with local coverage and any combination of two faults (and many multiple faults configurations) with at worst quasi-local coverage.

The 8-node ring shown in Fig 10(a) is embedded in a 4-dimension cube using the above method. The local coverage property is demonstrated in Fig 10(b), Fig 10(c) and Fig 10(d), where the reconfigured ring is shown after the failure of nodes 5, 6 and 4, respectively. By inspecting Fig 10(d) it is easy to see that the remaining five spares, $S1,\ldots,S5$, are not uniformly distributed in the cube, and that none of the eight nodes constituting the ring in Figure 10(d) enjoys the local coverage property. It is possible, however, through a global reassignment of the nodes, to recover the local coverage property for seven of the eight ring nodes. Specifically, if nodes 1, 8 and 2 are reassigned to $S4$, $S5$ and $S2$, respectively, and then node 5 is reassigned to the old location of node 1, then every node in the ring, except node 1, will have a spare that may cover it locally (see Fig 10(e)). Note that the restoration of the local coverage property requires the non-local relocation of four nodes. This may only be afforded if the system is in a relaxed mode. Moreover, the four relocations may be performed sequentially, and after each relocation, the system is consistent and ready to enter a strict mode if necessary.

Hence, in a bi-level faults coverage algorithm for this example, the goal of the global reconfiguration may be to minimize (and possibly reduce to zero) the number of nodes that do not enjoy the local coverage property. An exhaustive search algorithm may be easily designed to solve this problem. Specifically, given a fault configuration and a corresponding current embedding, the algorithm finds all the embeddings that minimize the number of locally uncovered nodes and, among these, choses the one that may be reached from the current embedding with the least number of node remappings. However, because of the huge number of possible embeddings of rings in faulty hypercubes, the exhaustive search technique may not be practically applied to reasonable size hypercubes. A fast heuristic is definitely needed in this case to find a near optimal configuration. Such a heuristics is given in [11].

CONCLUDING REMARKS

The concept of bi-level reconfiguration is a new concept that applies, whenever possible, a global optimization algorithm to optimize some fault tolerance criteria in the computing array. In contrast with reconfiguration algorithms that only aim at the restoration of functionality, global optimization takes into consideration possible distributions of future faults. The goal is to reconfigure the array in a way that ensures optimal fault tolerance capabilities in the remaining portion of the system's life-time.

We have presented examples of three criteria that may be considered during the global reorganization of computing arrays. Namely, the expected system's life-time, the error detection capability in the array, and the locality of fault recovery. Idealy, the reorganization should minimize a closed-form cost function based on these criteria. For instance, the cost functions in the examples of Sections 4 and 5 are the number of uncovered nodes, and the number of nodes with less than three modules, respectively. However, if more than one factor or criteria are to be considered, then a closed-form cost function may not be easily, or even precisely, defined. For example, in Section 6, the goal of the reorganization is to minimize the number of nodes with no local coverage, and in the same time minimize the reorganization effort. Also, in that same section, a more complex cost function may be defined if, as in Section 4, multiple modules are used for each node, and thus, non-faulty nodes may have different probabilities of survival depending on the number of non-faulty modules in each node.

In the three examples of this paper, we assumed that exhaustive search is used to minimize the cost function. In some cases, such a search is feasible. For instance, in Sections 4 and 5, the complexity of the search algorithms are $O(\frac{n}{25} 2^{16}) = O(n)$ and $O(2^n)$, respectively, where n is the number of nodes in the systems. In other cases, however, exhaustive search is too costly. For instance, in Section 6, the complexity of the search is $O((logn)^n)$. In all cases, it should be possible to design heuristic algorithms to approximately solve the minimization problems. In general, approximate solutions are acceptable in the context of bi-level algorithms because any reconfiguration which reduces the cost function is beneficial.

A different area that have not been investigated is the possible anomalies that may result if the global reconfiguration is forced to terminate prematurely due to the start of a strict mode. Specifically, when the global reconfiguration is completed in incremental steps, it is only required that after each step, the system be in a consistent state from which it may start computing. It is not required, however, that the cost function does not increase after each incremental step. In general, guaranteeing such a monotonic decrease in the cost function seems to severly restrict the class of applicable reconfigurations.

Finally, the idea of bi-level algorithms may be extended to areas other than fault tolerance. In memory management, for example, it is possible to implement bi-level garbage collection; a fast algorithm may be applied to quickly retrieve part of the unreferenced memory if the system is in a strict computational mode, and an efficient algorithm may be applied to clean up memory when the system in in a relaxed mode.

References

1. J. Abraham, P. Banerjee, C. Chen, W. Fuchs, S.Y. Kuo, and A. Reddy, "Fault Tolerance Techniques for Systolic Arrays," *IEEE Computer*, pp. 65-74, July 1987.
2. M. Alam and R. Melhem, "Fault Tolerant Augmented Hypercube Architectures," *Proc. of the IEEE Phoenix Comference on Computers and Communications*, March 1989.
3. P. Banerjee, J. Rahmesh, C. Stunkel, V. Nair, K. Roy, and J. Abraham, "An Evaluation of System-Level Fault Tolerance on the Intel Hypercube Multiprocessor," *Proc. of the 18th International Symposium on Fault-Tolerant Computing*, pp. 362-367, 1988.
4. W. Bouricious, W. Carter, D. Jessep, P. Schneider, and A. Wadia, "Reliability Modeling for Fault Tolerant Computer," *IEEE Transactions on Computers*, pp. 1306-1311, Nov. 1971.
5. A. Hassan and V. Agarwal, "A Fault Tolerant Modular Architecture for Binary Trees," *IEEE Trans. on Computers*, vol. C-35, no. 4, pp. 356-361.
6. S. Hosseini, J. Kuhl, and S. Reddy, "Distributed Fault Tolerance of Tree Structures," *IEEE Transactions on Computers*, vol. C-36, no. 11, pp. 1378-1382, November 1987.
7. K. Huang and J. Abraham, "Algorithm-based Fault-tolerance for MAtrix Operations," *IEEE Trans. on Computers*, vol. C-36, no. 6, pp. 518-528, June 1984.
8. I. Koren, "A Reconfigurable and Fault-tolerant VLSI Multiprocessor Array," *Proc. of the 8th Symposium on Computer Architecture*, pp. 425-442, 1981.
9. S.Y. Kung, C.W. Chang, and C.W. Jen, "Fault-Toelrance Design in Real-Time VLSI Array Processors," *Proc. of the IEEE Phoenix Comference on Computers and Communications*, pp. 110-115, 1987.
10. M. Lowrie and K. Fuchs, "Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance," *IEEE Trans. on Computers*, vol. C-36, no. 10, pp. 1172-1182, 1987.
11. R. Melhem, "Bi-level Reconfiguration of Fault Tolerant Arrays," Tech. Report 89-4. Department of Computer Science, The University of Pittsburgh.
12. R. Negrini, M. Sami, and R. Stefanelli, "Fault Tolerance Techniques for Array Structures used in Supercomputing," *IEEE Computer*, pp. 78-87, Feb. 1986.
13. F. Provost and R. Melhem, "Distributed Fault Tolerant Embedding of Binary Trees and Ring in Hypercubes," *Proceedings of the International Workshop on Defect and Fault Tolerance in VLSI Systems*, Oct. 1988.
14. D. Rennels, "On Implementing Fault-Tolerance in Binary Hypercubes," *Proc. of the sixteenth Symposium on Fault Tolerant Computing Systems*, pp. 344-349, 1986.
15. D. A. Rennels, "Fault Tolerant Computing: Concepts and Examples," *IEEE Trans. Computers*, pp. 1116-1129, Dec 1984.
16. A. Rosenberg, "The Diogenes Approach to Testable Fault-Tolerant Array Processors," *IEEE Trans. on Computers*, vol. C-32, no. 10, pp. 902-910, 1983.
17. L. Shombert and D. Siewiorek, "Using Redundancy for Concurrent Testing and Repairing Systolic Arrays," *The Seventeenth Int. Symp. on Fault-Tolerant Computing*, pp. 244-249, 1987.
18. A. Singh, "A Reconfigurable Modular Fault Tolerant Binary Tree Architecture," *Proc. of the Seventeenth Symp. on Fault Tolerant Computing*, pp. 298-304, 1987.