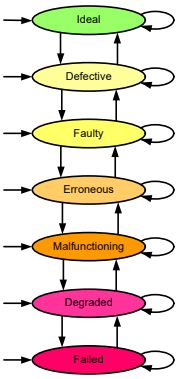
# V Malfunctions: Architectural Anomalies




"The Internet treats censorship as a malfunction and routes around it."

*John Perry Barlo*

". . . the technical demands of modern warfare are so complex [that] a considerable percentage of our material is bound to malfunction even before it is deployed against a foe."

*Ernest K. Gann, The Black Watch*

Chapters in This Part



A system moves from erroneous to malfunctioning state when an error affects the functional behavior of some constituent subsystem. Design or implementation flaws can lead to malfunctions directly, even in the absence of errors. At the architectural level, malfunctions have manifestations similar to those of faults occurring at the logic level. One difference is that instead of pass/not-pass testing to detect the presence of malfunctions, we tend to use diagnostic testing to detect and locate the offending modules. We then concern ourselves with methods to tolerate such malfunctions via redundancy and reconfiguration. Because modules interacting at the architectural level tend to be rather complex, standby or dynamic redundancy is often preferred to massive or static redundancy. Making either type of redundancy work, however, is nontrivial in view of difficulties in synchronization and maintenance of data integrity during and after switchovers. We conclude this part by a discussion of malfunction tolerance by means of robustness features in parallel processing systems.

# 17 Malfunction Diagnosis

"If information systems fail or seriously malfunction, societal activities lose support, and this may sometimes result in uncontrollable chaos in society as a whole."

*H. Inose and J. R. Pierce, Information Technology and Civilization*

"In diagnosis think of the easy first."

*Martin H. Fischer*

Topics in This Chapter



A malfunctioning subsystem must be identified and isolated quickly and effectively, in order to channel its impact toward a service-level degradation (soft failure) rather than a result-level breach (hard failure). Malfunction diagnosis, which is sometimes referred to as "system-level fault diagnosis" in the literature, encompasses a spectrum of techniques, from self-assessment, through hierarchical or stepwise testing, to cooperative diagnosis with centralized or distributed control. Unlike fault testing at the logic level, malfunction diagnosis entails the determination of not just the occurrence of a malfunction but also the identity or location of the offending subsystem.

## 17.1 Self-Diagnosis in Subsystems

Modern computer systems have processing capabilities that are distributed among multiple modules, even when there is only one "processor" in the system. Examples of units with processing capabilities include graphics cards, network interfaces, input/output channels, and device controllers. Furthermore, multiple CPUs are being employed to provide the required computational power in a wide spectrum of systems, given the marked slowdown in clock frequency improvements and the greater energy efficiency of slower processors. Thus, it makes sense to try to use these capabilities in performing cross-diagnostic checks among such modules.

Self-diagnostic checks are quite common. When you turn on your desktop or laptop computer, a diagnostic check is run to verify the correct functioning of major subsystems, including the CPU, memory, disk drive, and various interfaces. The check is not exhaustive but is intended to catch most common problems. In the context of dependable computing, we need a bit more coverage than such quick sanity checks.

In our discussion of fault testing in Chapter 9, we assumed that a special tester unit applied test patterns to the circuit under test and used the circuit's outputs to render a judgment about it health. In the case of intermodule diagnostic testing, this approach may prove impractical, given that the complexity of the modules involved would generate an extremely heavy volume of data being passed between them. One way around this difficulty is for the tester to initiate a self-diagnostic process in another module to determine whether it is working properly. This self-diagnostic should not have a yes/no answer, because such a binary outcome would increase the chances of a malfunctioning module generating a "yes" answer.

Here is a workable strategy. The initiator supplies the module under test with a "seed value" to be used in the diagnostic process. The self-diagnosing unit uses the supplied seed as an argument of an extensive computation that exercises nearly all of its components, including memory resources, ending up with an "answer" that is a known function of the seed value. It is this answer that is returned to the initiator as the diagnostic outcome. Given this outcome, it is then easy for the test initiator to compare the returned value with the expected result and to deduce whether the reporting unit is healthy. With a 64-bit diagnostic outcome, say, it is less likely for a malfunctioning unit to accidentally produce the correct result.

Note that when the health of a unit is suspect, there is no reason to trust its ability to execute the very instructions that constitute the self-diagnostic routine. For this reason, we often use a layered approach to self-diagnosis. At the beginning of the process, a small core of the module is tested. This core may be in charge of executing some very simple instructions and may have very limited memory and other resources. Once it has been established that the core can be trusted with regard to its health, the circle of trust is gradually extended to other parts of the module, in each phase using trusted parts whose health has been previously established to test new parts.

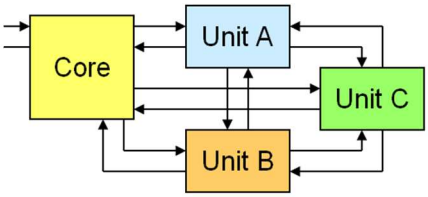
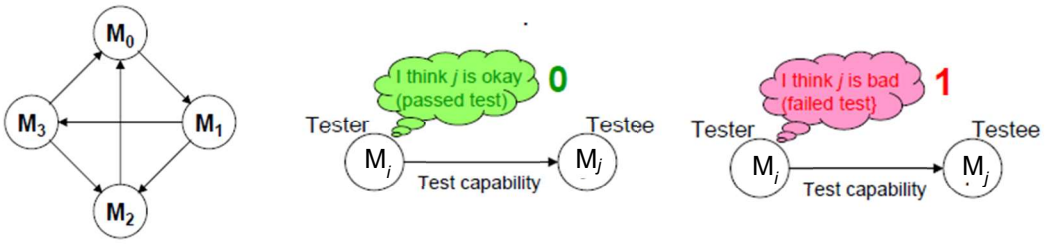



**Fig. 17.1 Layered approach to self-diagnosis begins by verifying the health of a small core and then gradually expands the circle of trust to subunits A, B, and C.**

## 17.2 Malfunction Diagnosis Models

The PMC system-level diagnosis model used in this book is due to Preparata, Metze, and Chien [Prep67]. The system under diagnosis consists of a set of modules for which we have defined a testing graph (Fig. 17.2a). A directed edge from $M_i$ to $M_j$ in the testing graph indicates that module $M_i$ can test module $M_j$. Note that the testing graph may or may not correspond to the actual physical connectivity among the modules. For example, the four modules of Fig. 17.2a may be interconnected by a bus, thus making it possible for each of them to test any other module. In this case, the testing graph is a proper subgraph of the complete graph characterizing the physical connectivity among the modules. Among reasons for selecting a subset of the available physical links to form a testing graph is the desire to reduce the communications overhead and to limit the module workloads that result from administering self-diagnostic tests and interpreting the results.

The diagnosis verdicts $D_{ij} \in \{0, 1\}$, with 0 meaning "pass" and 1 representing "fail," form an $n \times n$ Boolean diagnosis matrix $D$, which is usually quite sparse. In particular, the diagonal entries of $D$ are never used (unit $i$ does not judge itself). We assume that a good unit always renders a correct judgement about other units, that is, tests have perfect coverage and no false alarms, but that a verdict rendered by a malfunctioning unit is arbitrary and cannot be trusted. Note that the PMC model which we use for malfunction diagnosis is often referred to as a model for "system-level fault diagnosis." Following our terminology, a system-level fault is referred to as a malfunction.




(a) Testing graph (b) Pass outcome of a test (c) Fail outcome of a test

**Fig. 17.2 System-level testing graph and the two possible test outcomes when module *i* tests module *j*.**

**Example 17.1: Interpreting test outcomes** Consider the following diagnosis matrix $D$ for the 4-module system of Fig. 17.2a, in which dashes denote lack of testing. If we know that no more than one module can be malfunctioning, interpret the test outcomes in $D$.

$$D = \begin{bmatrix} - & 0 & - & - \\ - & - & 0 & 0 \\ 1 & - & - & - \\ 1 & - & 0 & - \end{bmatrix}$$

**Solution:** Module 0 is judged by both $M_2$ and $M_3$ to be malfunctioning ($D_{2,0} = D_{3,0} = 1$). If $M_0$ were good, then both $M_2$ and $M_3$ would have to be malfunctioning, which, by assumption, can't be the case. On the other hand, $M_0$ malfunctioning and the other 3 modules being good are consistent with all the test results in $D$. Thus, with the assumption of no more than one malfunctioning module, $M_0$ must be the culprit.

The PMC model is but one of the models available for malfunction diagnosis, but it is widely used and highly suitable for the points we want to make regarding key notions of system-level diagnosis. Other malfunction diagnosis models include the comparison model [Maen81], in which an observer assigns tasks to processors and draws conclusions regarding their health via comparing the results they return.

## 17.3 One-Step Diagnosability

A system characterized by a testing graph is said to be one-step diagnosable with respect to a predefined set of malfunction patterns if any pattern of malfunctions from the given set is correctly diagnosable from the resulting diagnosis matrix in every case. The qualifier "one-step" implies that a single round of testing, albeit potentially involving many tests, suffices for reaching a correct diagnosis decision. In particular, a system is one-step $t$-diagnosable if correct diagnosis is possible for any set of $t$ malfunctioning modules. We often delete the default qualifier "one-step," and call such a system $t$-diagnosable. With this terminology, the system in Fig. 17.2a is (one-step) 1-diagnosable.

To establish 1-diagnosability, we consider all possible single malfunctions and verify that the resulting syndromes (sets of diagnosis outcomes) are distinct from each other and from the no-malfunction case, regardless of how the malfunctioning unit behaves in its assessments of other modules.

**Example 17.synd: One-step 1-diagnosibility** Find the syndromes for the system of Fig. 17.2a and show that it is one-step 1-diagnosable but not 2-diagnosable.

**Solution:** Syndromes for the system of Fig. 17.2a are listed in Fig. 17.synd-a, with the results confirming the system's 1-diagnosability, given that the rows corresponding to different single malfunctions differ with each other and with the malfunction-free case in at lease one position. On the other hand, we see that the system of Fig. 17.2a is not 2-diagnosable, because the syndrome for the double malfunction $\{M_0, M_1\}$ may be indistinguishable from the single malfunction $\{M_0\}$ in some cases. Given the restriction to at most one malfunctioning unit, the syndrome dictionary in Fig. 17.synd-b allows us to translate the observed 6-bit syndrome $\{D_{01}, D_{12}, D_{13}, D_{20}, D_{30}, D_{32}\}$ to the identity of the malfunctioning unit, if any.

| Malfn | $D_{01}$ | $D_{12}$ | $D_{13}$ | $D_{20}$ | $D_{30}$ | $D_{32}$ |
|---|---|---|---|---|---|---|
| None | 0 | 0 | 0 | 0 | 0 | 0 |
| $M_0$ | 0/1 | 0 | 0 | 1 | 1 | 0 |
| $M_1$ | 1 | 0/1 | 0/1 | 0 | 0 | 0 |
| $M_2$ | 0 | 1 | 0 | 0/1 | 0 | 1 |
| $M_3$ | 0 | 0 | 1 | 0 | 0/1 | 0/1 |
| $M_0$,$M_1$ | 0/1 | 0/1 | 0/1 | 1 | 1 | 0 |
| $M_1$,$M_2$ | 1 | 0/1 | 0/1 | 0/1 | 0 | 1 |

(a) Syndromes for single and a few double malfunctions

| | |
|---|---|
| 0 0 0 0 0 0 | OK |
| 0 0 0 1 1 0 | $M_0$ |
| 0 0 1 0 0 0 | $M_3$ |
| 0 0 1 0 0 1 | $M_3$ |
| 0 0 1 0 1 0 | $M_3$ |
| 0 0 1 0 1 1 | $M_3$ |
| 0 1 0 0 0 1 | $M_2$ |
| 0 1 0 1 0 1 | $M_2$ |
| 1 0 0 0 0 0 | $M_1$ |
| 1 0 0 1 1 0 | $M_0$ |
| 1 0 1 0 0 0 | $M_1$ |
| 1 1 0 0 0 0 | $M_1$ |
| 1 1 1 0 0 0 | $M_1$ |

(b) Syndrome dictionary

**Fig. 17.synd Diagnosis syndromes and syndrome dictionary for single malfunctions.**

General results can be obtained for 1-step $t$-diagnosability that can be applied in some cases in lieu of an exhaustive analysis of the kind done in Example 17.synd.

**Theorem 17.td-nec** (Necessary conditions for 1-step $t$-diagnosibility): An $n$-unit system is 1-step $t$-diagnosable only if: (1) $n \geq 2t + 1$, that is, one-step diagnosability requires the bad modules to be in the minority; (2) Each module is tested by at least $t$ other modules.

The second necessary condition in Theorem 17.td-nec becomes a sufficient condition if we add the restriction that no two modules test each other. Intuitively, the absence of mutual testing among modules facilitates the deduction process because two malfunctioning units can potentially vindicate each other, making an error in judgment during the decision process more likely.

**Theorem 17.td-suff** (A sufficient condition for 1-step $t$-diagnosibility): An $n$-unit system in which no two units test one another is 1-step $t$-diagnosable iff each unit is tested by at least $t$ other units.

Based on Theorem 17.td-nec, the 4-module system of Fig. 17.2a can never become 2-diagnosable, regardless of how many links we add to the testing graph. By Theorem 17.td-suff, the same system is 1-diagnosable.

The diagnosability problem has a lot in common with a collection of popular puzzles about liars and truth-tellers. Consider the following setting. You visit an island whose inhabitants are from two tribes: member of one tribe (“liars”) consistently lie; members of the other tribe (“truth-tellers”) always tell the truth. Members of the two tribes are indistinguishable to us, but they can recognize eath other. The puzzles then ask us various questions about how to deduce the truth about various situations from the unreliable responses we receive. In fact, malfunction diagnosis corresponds to an extended, more challenging, version of these puzzles in which a third tribe (“randoms”) is introduced. Healthy modules correspond to “truth-tellers,” because they correctly diagnose other modules. Malfunctioning modules correspond to “randoms,” because their judgments are unrelated to the truth. Interestingly, liars aren’t as hard to deal with as randoms, because their consistency provides us with more information.

In dealing with 1-step diagnosability in a collection of interconnected modules, we are faced with two kinds of problems: analysis and synthesis. The analysis problem is itself of two kinds.

**Problem 17.a1** (the extent of 1-step *t*-diagnosability): Given a directed graph defining the test links, find the largest value of *t* for which the system is 1-step *t*-diagnosable.

The foregoing problem is easy if no two units test one another and fairly difficult if mutual testing is allowed. There exists a vast amount of published research dealing with Problem 17.a1.

**Problem 17.a2** (1-step malfunction diagnosis): Given a directed graph defining the test links and a set of test outcomes, identify all malfunctioning untis, assuming there are no more than *t* such units.

Problem 17.a2, which arises when we want to repair or reconfigure a system using test outcomes, is solved via table lookup or analytical methods.

The synthesis problem associated with 1-step diagnosability is as follows.

**Problem 17.s** (connection assignment for 1-step *t*-diagnosability): Specify the test links that would make an *n*-unit system 1-step *t*-diagnosable, using as few test links as possible.

As an example, a degree-*t* directed chordal ring, in which nodes are numbered 0 to $n - 1$ and node $i$ tests the $t$ nodes $i + 1, i + 2, \ldots, i + t$ (all experssions being mod $n$) has the required property.

A straightforward process for solving Problem 17.a2 is given in Algorithm 17.diag1, which uses exhaustive search with backtracking.

**Algorithm 17.diag1** An $O(n^3)$-step diagnosis algorithm
Input: The testing graph and a diagnosis matrix
Output: Every unit labeled G (good) or B (bad)
while some unit remains unlabeled do
    choose an unlabeled unit and label it G or B
    use labeled units to label as many other units as possible
    if the new label leads to a contradiction
    then backtrack
    endif
endwhile

A somewhat more efficient $O(n^{2.5})$-step diagnosis algorithm is as follows. From the original testing graph, derive an L-graph that has the same set of nodes and a link from node $i$ to node $j$ iff node $i$ can be assumed to be malfunctioning when node $j$ is known to be good. The unique minimal vertex cover of the L-graph, that is a subset of its nodes that touches at least one of the two endpoints of each edge, corresponds to the set of $t$ or fewer malfunctioning units.

## 17.4 Sequential Diagnosability

In Example 17.synd, we established that the system of Fig. 17.2a is not 2-diagnosable, because the syndromes for the double malfunction $\{M_0, M_1\}$ is potentially indistinct from that of the single malfunction $\{M_0\}$. We may note that a common syndrome for the two malfunction patterns just listed does provide some useful diagnostic information: that $M_0$ is definitely bad. So, we can potentially use this information to replace or repair $M_0$ before further testing to identify other malfunctioning modules. This observation leads us to the notion of *sequential diagnosability*, which means that the test syndrome points unambiguously to at least one malfunctioning unit. Assuming that we began with $k$ malfunctions, replacing or repairing one bad unit leaves us with no more than $k - 1$ malfunctions, thus reducing the diagnosis problem to a simpler one. Iterating in this manner, allows us to identify all $k$ malfunctioning units in $k$ or fewer rounds.

**Example 17.seqd1: Sequential-diagnosibility** Show that the system of Fig. 17.2a isn't sequentially 2-diagnosable.

**Solution:** The desired result is readily established by noting that the syndromes for the malfunction sets $\{M_0, M_2\}$ and $\{M_1, M_3\}$, that is, (x 1 0 x 1 1) and (1 x x 0 x x), where x denotes 0/1 and test results are listed in the order shown in Fig. 17.synd-a, are potentially indistinct.

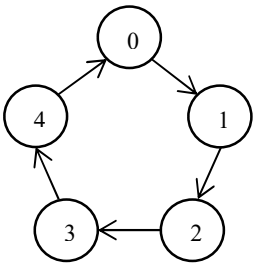
In fact, the result of Example 17.seqd could have been established based on the following general theorem.

**Theorem 17.seqd** (A necessary condition for sequential diagnosibility): An $n$-unit system is sequentially $t$-diagnosable only if $n \geq 2t + 1$, that is, sequential diagnosability is possible only if the bad modules are in the minority.

**Example 17.seqd2: Sequential-diagnosibility of a unidirectional ring** Show that a system whose testing graph is a unidirectional or directed ring is sequentially 2-diagnosable, but not one-step 2-diagnosable or sequentially 3-diagnosable.

**Solution:** As an example, consider the 5-node directed ring of Fig. 17.dring-a. Possible test outcomes are for single and some double malfunctions are depicted in Fig. 17.dring-b. We note that even though some of the syndromes shown overlap, they all point to $M_0$ being malfunctiong. So, under sequential diagnosabiloity, there is no ambiguity and we can replace $M_0$ before proceeding. The set of all syndromes that point to $M_0$ being malfunctioning is shown in Fig. 17.dring-c.

It is possible to prove that an $n$-node directed ring is sequentially $t$-diagnosable for any $t$ satisfying $\lceil (t^2 - 1)/4 \rceil + t + 2 \leq n$ (see Problem 17.5), a result from which two parts of Example 17.seqd2 follow.




| Malfn | $D_{01}$ | $D_{12}$ | $D_{23}$ | $D_{34}$ | $D_{40}$ |
|---|---|---|---|---|---|
| $M_0$ | x | 0 | 0 | 0 | 1 |
| $M_1$ | 1 | x | 0 | 0 | 0 |
| $M_2$ | 0 | 1 | x | 0 | 0 |
| $M_3$ | 0 | 0 | 1 | x | 0 |
| $M_4$ | 0 | 0 | 0 | 1 | x |
| $M_0, M_1$ | x | x | 0 | 0 | 1 |
| $M_0, M_2$ | x | 1 | x | 0 | 1 |
| $M_0, M_3$ | x | 0 | 1 | x | 1 |
| $M_0, M_4$ | x | 0 | 0 | 1 | x |

Syndromes for $M_0$ bad:

0 0 0 0 1
0 0 0 1 0
0 0 0 1 1
0 0 1 0 1
0 0 1 1 1
0 1 0 0 1
0 1 1 0 1
1 0 0 0 1
1 0 0 1 0
1 0 0 1 1
1 0 1 0 1
1 0 1 1 1
1 1 0 0 1
1 1 1 0 1

(a) Directed ring (b) Diagnosis syndromes (c) Syndrome dictionary

**Fig. 17.dring Diagnosis syndromes and syndrome dictionary for sequential diagnosability of a 5-node directed ring.**

Analysis and synthesis problems for sequential diagnosability parallel those of 1-step diagnosability, and have likewise been extensively studied.

**Problem 17.a1-s** (the extent of sequential diagnosability): Given a directed graph defining the test links, find the largest value of $t$ for which the system is sequentially $t$-diagnosable.

**Problem 17.a2-s** (sequential malfunction diagnosis): Given a directed graph defining the test links and a set of test outcomes, identify at least one malfunctioning unti (preferably more), assuming there are no more than $t$ such units.

**Problem 17.s-s** (connection assignment for sequential diagnosability): Specify the test links that would make an $n$-unit system sequentially $t$-diagnosable, using as few test links as possible.

An $n$-node ring, with $n >= 2t + 1$, with added test links from $2t - 2$ other nodes to node 0 (besides node $n - 1$, which already tests it) has the required property.

## 17.5 Diagnostic Accuracy and Resolution

So far in our discussions of diagnosability, we have demanded full accuracy in the sense of requiring that all malfunctioning modules be identified (1-step diagnosability) or that some malfunctiong modules, and only malfunctioning modules, be deduced (sequential diagnosability) from the test outcomes. By relaxing these requirements, we may be able to successfully diagnose systems that would be undiagnosable with the former, more strict definitions.

An $n$-unit system is 1-step $t/s$-diagnosable if a set of no more than $t$ malfunctioning units can always be identified to within a set of $s$ units, where $s \geq t$. By allowing $s - t$ healthy units to be potentially included among those flagged for repair and replacement, diagnosis may become simpler or possible in some cases. Note that our original notion of 1-step $t$-diagnosability does not correspond to the widely studied special case of 1-step $t/t$-diagnosability in this new notation. The reason is that in $t/t$-diagnosability, it is admissible to identify a set of $t$ modules for replacement when there are in fact only $t - 1$ malfunctioning units.

Given the values of $t$ and $s$, the problem of deciding whether a system is $t/s$-diagnosable is co-NP-complete. However, there exist efficient, polynomial-time, algorithms to find the largest integer $t$ such that the system is $t/t$- or $t/(t + 1)$-diagnosable.

A similar relaxation in the diagnosis accuracy is possible for sequential diagnosis. An $n$-unit system is sequentially $t/r$-diagnosable if from a set of up to $t$ malfunctioning units, $r$ can be identified in one step, where $r < t$ and the identified set contains at least one malfunctioning unit.

Finally, we can integrate the notion of safety, where some malfunctions are promptly detected but not necessarily diagnosed, into our definitions. Safe diagnosability implies that up to $t'$ malfunctions are correctly diagnosed and up to $u$ are detected, where $u > t'$. This kind of diagnosability/testability, which is reminiscent of combination error-correcting/detecting codes, ensures that there is no danger of incorrect diagnosis for a larger number of malfunctions (up to $u$), thus increasing system safety.

## 17.6 Other Topics in Diagnosis

Diagnosability results have been published for a great variety of regular interconnection networks, such as the three topologies shown in Fig. 17.topo. Topologies like these have been used in the design of many general-purpose parallel computers and special-purpose architectures for high-performance computing. The specific examples shown in Fig. 17.topo are composed of degree-4 nodes and thus can be 1-step 4-diagnosable at best. Proving that they indeed possess this level of diagnosability or deriving their level of sequential diagnosability are active research areas.

What comes after malfunction diagnosis?

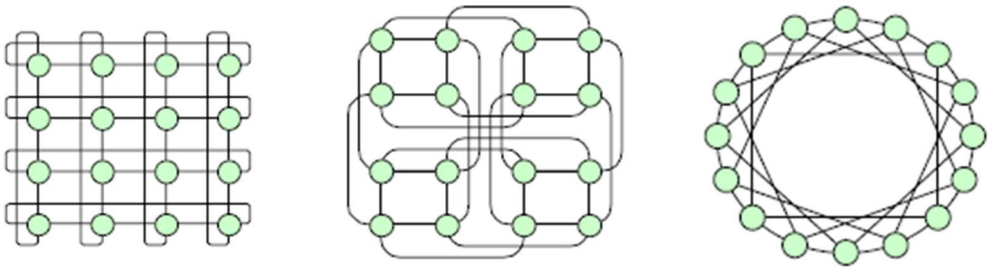
When one or more malfunctioning units have been identified, the system must be reconfigured to allow it to isolate those units and to function without the unavailable resources

Reconfiguration may involve:

1. Recovering state info from removed modules or back-up storage
2. Reassigning tasks and reallocating data
3. Restarting the computation from last checkpoint or from scratch

In a bus-based system, we isolate malfunctioning units, remove them, and plug in good modules (standby spares or repaired ones)

In a system having point-to-point connectivity, we reconfigure by rearranging the connections in order to switch in (shared) spares, using methods similar to those developed for defect circumvention



(a) 2D torus (b) 4D hypercube (c) Chordal ring

**Fig. 11.topo Some interconnection networks for parallel processing.**

# Problems

### 17.1 Diagnosability in a directed ring

Prove directly (i.e., by forming malfunction syndromes and comparing them with each other, rather than by using general theorems about diagnosability) that an $n$-node directed ring network is 1-step 1-diagnosable but not 1-step 2-dignosable. *Hint:* Take advantage of symmetry to reduce the amount of work.

### 17.2 Malfunction diagnosis in a bidirectional ring

a. Show that a bidirectional ring with five or more nodes is one-step 2-diagnosable.

b. Prove that the result of part a is the strongest possible. In other words, a bidirectional ring cannot be 3-diagnosable, and a ring with 3 or 4 nodes is not 2-diagnosable.

### 17.3 Diagnosability in a 2D torus

Consider 1-step $t$-diagnosability of a collection of subsystems interconnected as an $m \times m$ 2D torus network in which node $(i, j)$ in row $i$, column $j$, is connected to the four neighboring nodes $(i \pm 1, j)$ and $(i, j \pm 1)$, where all arithmetic is modulo $m$ and $m > 4$. Links are bidirectional and allow testing in either direction.

a. By defining a suitable testing graph, show that 2D torus is at least 1-step 2-diagnosable (i.e., $t \geq 2$).

b. Can you prove a stronger diagnosability result for the 2D torus? [Arak00]

### 17.4 Emergency notification system

An emergency notification system is built to detect any one of $k$ possible hazardous conditions. It utilizes $r$ alarm units with sensors, where each alarm can detect a different subset of the $k$ conditions.

a. Formulate this problem as a malfunction diagnosis problem and its associated graph representation.

b. Assuming that alarms do not fail, under what conditions do the alarms collectively pinpoint which hazardous conditions exist? Explain.

c. Now consider the possibility of failures in the alarms. Discuss the requirements for correct identification of the existing hazardous condition(s).

### 17.5 Sequential diagnosability of directed rings

Prove that an $n$-node directed ring is sequentially $t$-diagnosable for any $t$ satisfying $\lceil (t^2 - 1)/4 \rceil + t + 2 \leq n$.

### 17.6 Diagnosability of swapped/OTIS networks

A swapped (aka OTIS) network based on an $n$-node basis network or graph $G$ is built as follows [Parh05]. The network $Sw(G)$ has $n^2$ nodes belonging to $n$ different copies of $G$. Each copy of $G$ is conneted internally as in the original basis network $G$. Additionally, node $i$ in copy $j$ of $G$ is connected to node $j$ in copy $i$ of $G$. The latter links are known as intercluster links, whereas the links of $G$ are intracluster links. Thus, node degree of $Sw(G)$ is one more than that of $G$. Study the diagnosability of swapped netwotks

### 17.7 Diagnosability of biswapped networks

A biswapped network based on an $n$-node basis network or graph $G$ is built as follows [Xiao10]. The network $Bsw(G)$ has $2n^2$ nodes belonging to $2n$ different copies of $G$, half of the copies appearing in part 0 and half in part 1. Each copy of $G$ is conneted internally as in the original basis network $G$. Additionally, node $i$ in copy $j$ of $G$ in part $k$ is connected to node $j$ in copy $i$ of $G$ in part $1 - k$. The latter links are known

as intercluster links, whereas the links of $G$ are intracluster links. Thus, node degree of $Bsw(G)$ is one more than that of $G$. The intercluster links define a bipartite subgraph; hence, the name "biswapped." Study the diagnosability of biswapped netwotks

### 17.8 One-step 1-diagnosability

In example 17.synd, we established that the system depicted by the testing graph of Fig. 17.2a is 1-step 1-diagnosable.

a. Remove a minimum number of test links from the graph so that the system it represents is no longer 1-step 1-diagnosable.

b. What is the maximum number of links that can be removed from the testing graph so that the systeme it represents remains 1-step 1-diagnosable? Fully justify your answer.

### 17.9 One-step 1-diagnosability

Why does the necessary and sufficient condition "each unit being tested by at least $t$ other units" along with "no two units testing each other" guarantee the satisfaction of condition 1 in Theorem 17.td-nec?

### 17.10 Sequential diagnosability of directed rings

Prove that the sufficient condition $\lceil (t^2 - 1)/4 \rceil + t + 2 \leq n$ for an $n$-node directed ring being sequentially $t$-diagnosable implies $n \geq 2t + 1$ but that the necessary condition $n \geq 2t + 1$ is not sufficient for an $n$-node directed ring to be sequentially $t$-diagnosable.

### 17.11 A necessary and sufficient condition for *t*-diagnosability

Prove that a graph $G$ is $t$-diagnosable iff it contains a star of order $t$ at each node $v$ (consisting of $v$ connected to nodes $x_1, x_2, \ldots, x_t$), with each node $x_i$ connected to a distinct node $y_i$. [Hsu07]

### 17.12 A sufficient condition for *t*/*t*-diagnosability

Prove that a $k$-connected regular graph is $t/t$-diagnosable for $t = 2k - 2 - g$, where $g$ is the maximum number of neighbors shared by adjacent nodes. [Hao16]

## References and Further Readings


[Arak00] Araki, T. and Y. Shibata, "Diagnosability of Networks Represented by the Cartesian Product," *IEICE Trans. Fundamentals*, Vol. E83-A, No. 3, pp. 465-470, March 2000.

[Butl08] Butler, R. W., "A Primer on Architectural Level Fault Tolerance," NASA Technical Memorandum TM-2008-215108, 48 pp., February 2008.

[Chan10] Chang, G.-Y., "($t$, $k$)-Diagnosability for Regular Networks," *IEEE Trans. Computers*, Vol. 59, No. 9, pp. 1153-1157, September 2010.

[Gu18] Gu, M.-M., R.-X. Hao, and J.-B. Lu, "The Pessimistic Diagnosability of Data Center Networks," *Information Processing Letters*, Vol. 134, pp. 52-56, June 2018.

[Haki74] Hakimi, S. L. and A. T. Amin, "Characterization of Connection Assignment of Diagnosable Systems," *IEEE Trans. Computers*, Vol. 23, pp. 86-88, 1974.

[Hao16] Hao, R.-X., M.-M. Gu, and Y.-Q. Feng, "The Pessimistic Diagnosabilities of Some General Regular Graphs," *Theoretical Computer Science*, Vol. 609, Pt. 2, pp. 513-420, January 2016.

[Hsu07] Hsu, G. H. and J. J. M. Tan, "A Local Diagnosability Measure for Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol. 18, No. 5, pp. 598-607, 2007.

[Karu79] Karunanithi, S. and A. D. Friedman, "Analysis of Digital Systems Using a New Measure of System Diagnosis," *IEEE Trans. Computers*, Vol. 28, No. 2, pp. 121-123, February 1979.

[Maen81] Maeng, J. and M. Malek, "A Comparison Connection Assignment for Self-Diagnosis of Multiprocessor Systems," *Proc. 11th Int'l Symp. Fault-Tolerant Computing*, 1981, pp. 173-175.

[Parh05] Parhami, B., "Swapped Interconnection Networks: Topological, Performance, and Robustness Attributes," *J. Parallel and Distributed Computing*, Vol. 65, No. 11, pp. 1443-1452, November 2005.

[Parh16] Parhami, B., N. Wu, and S. Tao, "Taxonomy and Overview of Distributed Malfunction Diagnosis in Networks of Intelligent Nodes," *J. Computer Science and Engineering*, Vol. 13, No. 2, pp. 23-31, 2016.

[Prep67] Preparata, F. P., G. Metze, and R. T. Chien, "On the Connection Assignment Problem of Diagnosable Systems," *IEEE Trans. Electronic Computers*, Vol. 16, pp. 848-854, 1967.

[Seng92] Sengupta, A. and A. Dahbura, "On Self-Diagnosable Multiprocessor Systems: Diagnosis by the Comparison Approach," *IEEE Trans. Computers*, Vol. 41, No. 11, pp. 1386-1396, 1992.

[Soma87] Somani, A. K., V. K. Agarwal, and D. Avis, "A Generalized Theory for System Level Diagnosis," *IEEE Trans. Computers*, Vol. 36, pp. 538-546, 1987.

[Sull88] Sullivan, G., "An $O(t^3 + |E|)$ Fault Identification Algorithm for Diagnosable Systems," *IEEE Trans. Computers*, Vol. 37, pp. 388-397, 1988.

[Xiao10] Xiao, W. J., B. Parhami, W. D. Chen, M. X. He, and W. H. Wei "Fully Symmetric Swapped Networks Based on Bipartite Cluster Connectivity," *Information Processing Letters*, Vol. 110, No. 6, pp. 211-215, 15 February 2010.

# 18 Malfunction Tolerance

"If you improve or tinker with something long enough, eventually it will break or malfunction."

*Arthur Bloch*

"The test of courage comes when we are in the minority. The test of tolerance comes when we are in the majority."

*Ralph W. Sockman*

"I have seen gross intolerance shown in support of tolerance."

*Samuel Taylor Coleridge*

Topics in This Chapter

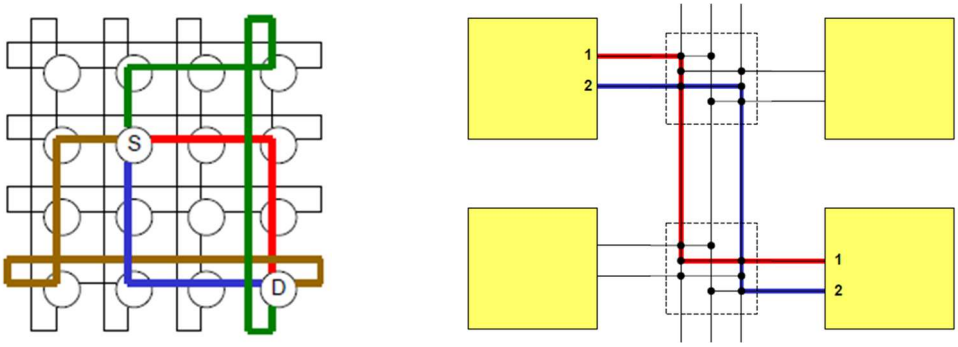


Once a malfunctioning subsystem has been identified, the system must be reconfigured to work without it or with a replacement unit (spare). In either case, any informational resource residing in the malfunctioning unit must be reconstructed or worked around. Details of the reconfiguration and recovery strategies are system- and application-dependent. Thus, our focus in this chapter is on general mechanisms that facilitate the implementation of such strategies. Some of the indispensible ideas include module isolation techniques, reconfiguration with spares, tradeoffs in switching complexity and overhead, and task scheduling strategies that account for malfunctions.

## 18.1 System-Level Reconfiguration

A system consists of modular resources (processors, memory banks, disk storage, interfact units, and the like) plus interconnects. Redundant resources can mitigate the effect of module malfunctions. Thus, a key challenge in reconfiguration is dealing with interconnects. Throughout our discussion in this chapter, we will assume that module and interconnect malfunctions are promptly diagnosed via self-checking, external monitoring, or concurrently executed system-level tests. We can model system resources, including both modules and interconnects, by means of directed or undirected graphs. To be able to overcome the effect of link malfunctions, it is necessary to have multiple paths from each potential source to every possible destination.

For example, Fig. 18.1a shows a 16-processor parallel computer whose nodes are interconnected via a 2D torus topology. It is readily seen that each source node is connected to each possible destination node via 4 parallel node-and-edge-disjoint paths. Because of this property, any set of 3 malfunctioning resources (nodes and/or links) can be tolerated without cutting off intermodule communication. In graph-throretic terms, we say that the system in Fig. 18.1a is 4-connected. High connectivity is a desirable attribute for malfunction tolerance.




(a) Parallel paths in a 2D torus (b) Parallel paths in a bus-based system

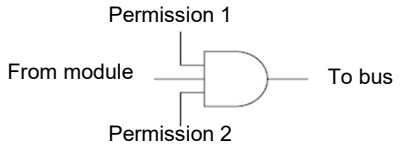
**Fig. 18.1 Architectural redundancy in a torus-connected parallel computer and in a bus-based system.**

As a second example, consider the 3-bus system of Fig. 18.1b, where each I/O port from a 2-port module is connectable to one of two buses. The dashed boxes in the middle can be viewed as programmable 2 × 2 switches that allow each module to communicate with any other module via two different paths. Thus, besides tolerating malfunctioning modules, we can route around a single malfunctioning bus as well.

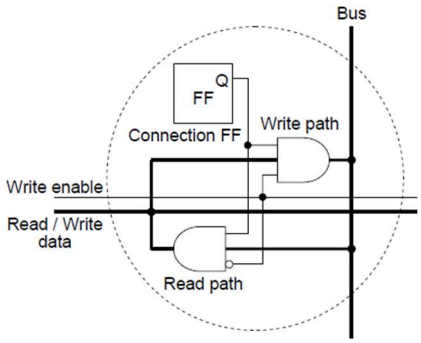
## 18.2 Isolating a Malfunctioning Element

Reconfiguration techniques allow us to route around malfunctiong modules or interconnects but do not necessarily ensure that the circumvented elements do not interfere with the proper functioning of the healthy ones. A prime example occurs in bus-based systems. When we decide not to use a malfunctioning module in a bus-based system, the bad module may still place nonsensical data on the bus, thus preventing the good modules from properly communicating on the bus. A possible solution is depicted in Fig. 18.isol-a. Each module can place data on the bus only if it has permission from two other modules. Ignoring the problems associated with assigning and managing such permissions, which are admittedly nontrivial, we see that it would take at least 3 malfunctioning modules in order to cause interference on the bus.

We can abstract the following general strategy from the scheme of isolating a module from a bus, as discussed in the preceding paragraph. When a critical shared resource is to be accessed by a module, some form of external authorization is needed to ensure that a run-away module does not cause the entire system to crash. The situation can be likened to transactions at a bank. A customer can perform simple transactions at an ATM. If the customer wants to withdraw a larger sum of money than the ATM's transaction limit, assistance from a teller must be sought (a form of external authorization). For very large transactions, even the teller does not have the required authority and the branch manager gets involved (the second external authorization).




(a) Isolating a module from a bus




(b) Reading from and writing to a bus

**Fig. 18.isol Methods of isolating malfunctioning modules to ensure noninterference in the proper functioning of other moduels.**

An alternative to the double-permission scheme of Fig. 18.isol-a, with its elaborate assignment and collaborative management requirements, is the use of a single connection flip-flop, as depicted in Fig. 18.isol-b. To isolate a module from the bus, one resets its connection flip-flop, which then disables reading from and writing to the bus. The multiple-bus scheme of Fig. 18.1b can be realized by using this circuitry for each of the heavy dots connecting a module I/O port to a bus. In this case, any problems in the connection flip-flop or associated circuitry can be treated as a bus malfunction for both modeling and circumvention purposes.

Isolation of modules is somewhat simpler in systems with point-to-point communication. Referring to Fig. 18.1a, as long as each healthy module is aware of the identity of its malfunctioning neighbors, it can simply ignore all communications from those modules.

Malfunction tolerance would be much easier if modules would simply stop functioning, rather than engage in arbitrary behavior. Unpredicatable or arbitrary behviour on the part of a malfunctioning element, sometimes referred to as a “Byzantine malfunction,” is notoriously difficult to handle. One source of the difficulty is that the module’s arbitrary behavior may make it seem different to multiple external observers, some judging it to be healthy and others detecting the malfunction.

Methods are available to ensure that a malfunctioning module stops in an inert state, where it can’t confuse the system’s healthy modules. Here is one way to accomplish this goal. Suppose modules run (approximately) synchronized clocks and have access to reliable stable storage, where critical data can be stored. A $k$-malfunction-stop module can be implemented from $k + 1$ identical units of this kind, operating in parallel. The key element for this realization is an s-process that decides when the redundant module has stopped and sets a “stop” flag in stable storage to “true.”

[More details to be supplied.]

# Malfunction-stop modules

Malfunction tolerance would be much easier if modules simply stopped functioning, rather than engage in arbitrary behavior

Unpredictable (Byzantine) malfunctions are notoriously hard to handle

Assuming the availability of a reliable stable storage along with its controlling s-process and (approximately) synchronized clocks, a $k$-malfunction-stop module can be implemented from $k + 1$ units

Operation of s-process to decide whether the module has stopped:

$R$ := bag of received requests with appropriate timestamps
if $|R| = k+1 \wedge$ all requests identical and from different sources $\wedge \neg$ *stop*
then if request is a write
    then perform the write operation in stable storage
    else if request is a read, send value to all processes
else set variable *stop* in stable storage to TRUE

## 18.3 Data and State Recovery

Logs are essential tools for system recovery via undo and redo operations. The undo and redo operations are quite similar to their namesakes in word-processing and other common applications. When a detected malfunction makes it impossible or inadvisable to continue processing as usual, the partial effects of incomplete transactions must be undone to maintain consistency in stable storage. Similarly, the partially complete transaction must be redone when circumstance allow it.

Logs maintain redundant info (in stable storage, of course) for the sole purpose of recovery from malfunctions

The write-ahead log (WAL) protocol requires that a transaction:

Write an undo-log entry *before* it overwrites an object in stable storage with uncommitted updates

Write both undo-log and redo-log entries *before* committing an update to an object in stable storage

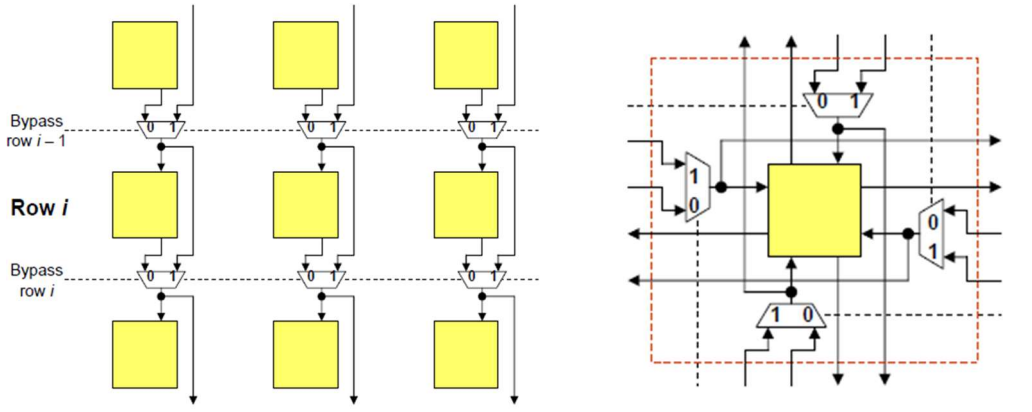
Not safe to write logs after overwriting or committing

Research is being done at Microsoft and elsewhere to allow querying a database on its state at any desired time instant in the past

## 18.4 Regular Arrays of Modules

Regular arrays of modules are used extensively in certain applications that need high-throughput processing of massive amounts of data. Examples include communication routing, scientific modeling, visual rendering, and certain kinds of simulation. Note that regularity in our discussions here refers to the interconnection pattern, not physical layout (the latter may be the case for on-chip systems). The focus of our discussion will be on 2D arrays, although some of the techniques can be extended to higher dimensions in a straightforward manner.

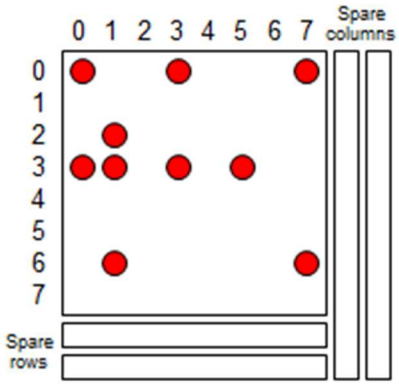
Row/column bypassing is a widely used method for reconfiguring 2D arrays. Let us first focus on row bypassing for modules that communicate in one direction: from top to bottom. As seen in Fig. 18.pass-a, placing a multiplexer at the input to each module allows us to bypass the previous row, taking the input from the row immediately above it. Applying the same scheme to the other 3 inputs of a 4-port module leads to the building block of Fig. 18.pass-b, which allows both row and column bypassing.




(a) Row bypassing in downward direction (b) Module with row/column bypass

**Fig. 18.pass Reconfiguration via row/column bypassing in 2D arrays.**

## Choosing rows and/or columns to bypass

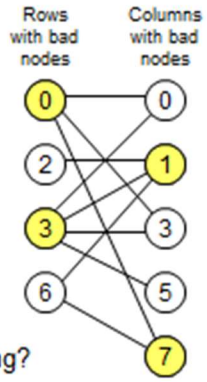



In the adjacent diagram, can we choose up to 2 rows and 2 columns so that they contain all the bad nodes?

Convert to graph problem (Kuo-Fuchs):

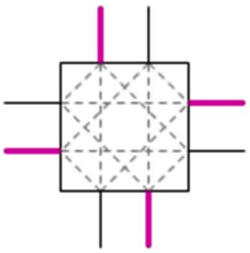
Form bipartite graph, with nodes corresponding to bad rows and columns

Find a cover for the bipartite graph (set of nodes that touch every edge)




Question: In a large array, with *r* spare rows and *c* spare columns, what is the smallest number of bad nodes that cannot be reconfigured around with row/column bypassing?

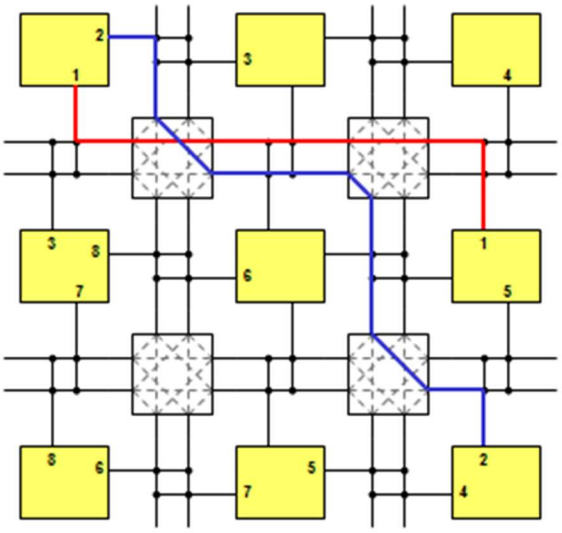
## Switch modules in FPGAs



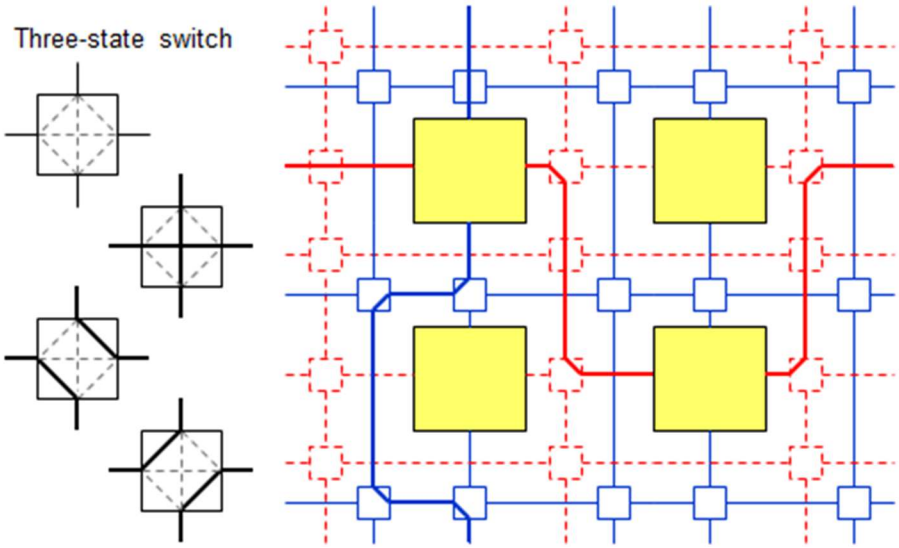
Interconnection switch with 8 ports and four connection choices for each port:

0 – No connection
1 – Straight through
2 – Right turn
3 – Left turn

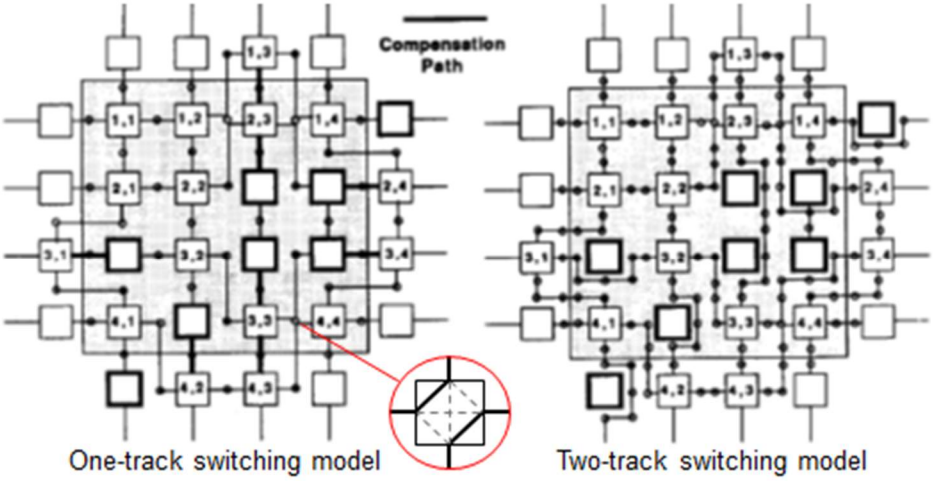
8 control bits (why?)



An array reconfiguration scheme




One-track and two-track switching schemes

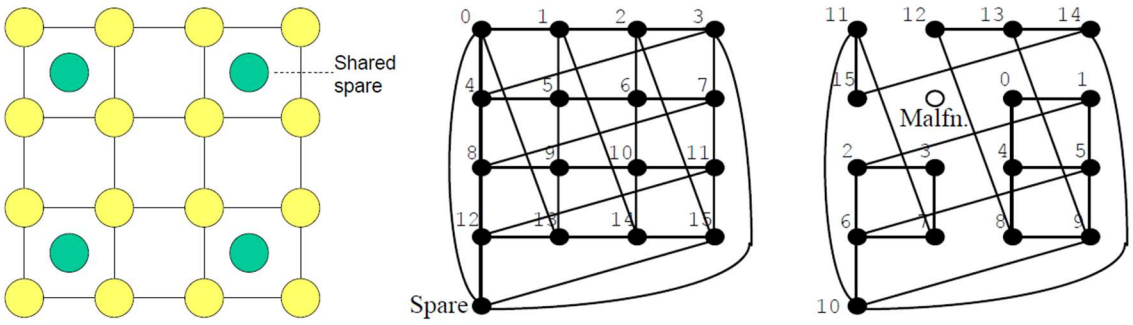



Source: S.-Y. Kung, S.-N. Jean, C.-W. Chang, *IEEE TC*, Vol. 38, pp. 501-514, April 1989

## 18.5 Low-Redundancy Sparing

Row/column bypassing is attractive in view of its simple reconfiguration and decision logic. However, it is rather wasteful when an entire row or column of modules is discarded to circumvent a single malfunctioning unit. It is thus natural to ask whether we can come up with lower-redundancy schemes that make more efficient use of the spare resources. As shown in Fig. 18.spare-a, it might be possible to place shared spares in certain strategic locations, where then can readily replace a malfunctioning module from an assigned set. To achieve this goal, the spare modules usually need greater connectivity that the primary modules, given the flexibility requirements for different replacement patterns. For example, if each spare module in Fig. 18.spare-a is to be able to replace any of the four primary modules within its cluster while maintaining the same communication structure, it will need at least 8 ports.

For 2D meshes without wraparound links (i.e., not torus networks), an ingenious reconfiguration scheme allows the use of a single spare module for replacing any malfunctioning module located in any row/column. The scheme, depicted in Fig. 18.spare-b, takes advantage of unused ports at the edges of a mesh to provide additional links that are not used under normal conditions. When a module malfunctions, the remaining healthy modules are renumbered (assigned new row/column numbers) and their ports relabeled to form a new working mesh. In the example of Fig. 18.spare-b, once the malfunctioning module 5 has been isolated, the system is reconfigured as shown, so that, for example, the new row 0 will consist of modules 6, 7, 8, and 9, which are renumbered 0, 1, 2, and 3.



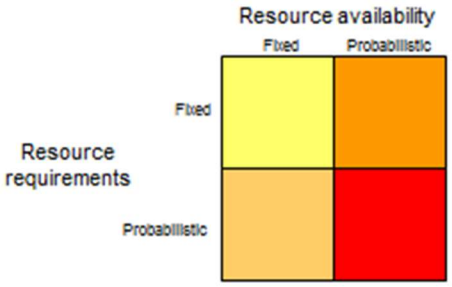

(a) Shared spare in a cluster (b) Single-spare scheme for a 2D array

**Fig. 18.spare Low-redundancy sparing in 2D arrays of modules.**

## 18.6 Malfunction-Tolerant Scheduling

One of the considerations in computer systems composed of multiple computational resources is the assignment of various tasks or subtasks to the available modules. This is known as task scheduling. Schedules can be static (determined at the outsets, and adhered to through the course of the computation) or dynamic. In the latter case, changes due to resource fluctuations or imbalance in load that may result from inaccuracies in task running-time predictions are possible. Clearly, malfunction tolerance requires some degree of dynamism in task scheduling to allow changes when modules are removed from service due to malfunctions.

Task scheduling problems are hard even when resource requirements and availability are both fixed and known a priori. These problems become significantly more difficult when resource requirements fluctuate and/or resource availability changes dynamically due to modules malfunctioning and are returned to service after repair.




When resource availability is fixed, the quality of a schedule is judged by:
(1) Completion times (2) Meeting of deadlines

When resources fluctuate, deadlines may be met probabilistically or accuracy/completeness may be traded off for timeliness

# Problems

### 18.1 xxx

Intro

a. xxx
b. xxx
c. xxx

### 18.2 xxx

Intro

a. xxx
b. xxx
c. xxx
d. xxx

### 18.3 Placement of spare modules

Consider a linear array of $n$ modules similar to that in Fig. 6.array1D. When an active module is replaced with a spare, the contents of some of the modules must be shifted so as to maintain the correct module ordering in the array. It makes sense to try to place the spare modules in locations along the array that would minimize the amount of data shifting required [Parh77].

a. Show that a single spare module is best placed at the middle of the array when $n$ is odd and at either of the two central locations when $n$ is even. State all your assumptions clearly.
b. How would the answer to part a change is the modules are arranged in a ring rather than a linear array?
c. Show that multiple spare modules should be spaced equally along the array.
d. Discuss whether using the same criteria presented in this problem, the placement of spare rows and columns within 2D arrays can be optimized.

### 18.4 Shared spares in a regular array

Consider a 2D array of modules with side lengths $m = 2^a$, with a shared spare provided for every 4 modules, as depicted in Fig. 18-spare-a. Assume that the switching mechanisms are perfectly reliable and that all modules, including spares, have the same reliability $r(t)$.

a. Construct a cominational reliability model for the system.
b. We noted that for the $4 \times 4$ array, each spare module needs 8 ports. What is the required number $p$ of ports for a spare module when $a > 2$?
c. When a spare replaces an ordinary module, one of its $p$ ports should be assigned for communication in each of the east, west, north, and south directions. How many possible neighbors does the replaced module have in each direction?
d. Design the required switching mechanism to allow the port assignment alluded to in part c.
e. Can the sparing scheme of this problem be extended to the 2D torus interconnection pattern?

## References and Further Readings

[Bruc93] Bruck, J., R. Cypher, and C.-T. Ho, "Fault-Tolerant Meshes and Hypercubes with Minimal Numbers of Spares," *IEEE Trans. Computers*, Vol. 42, No. 9, pp. 1089-1104, September 1993.

[Butl08] Butler, R. W., "A Primer on Architectural Level Fault Tolerance," NASA Technical Memorandum TM-2008-215108, 48 pp., February 2008.

[Cast15] Castro-Leon, M., H. Meyer, D. Rexachs, and E. Luque, "Fault Tolerance at System Level Based on RADIC Architecture," *J. Parallel and Distributed Computing*, Vol. 86, pp. 98-111, December 2015.

[Jian15] Jiang, G, J. Wu, Y. Ha, Y. Wang, and J. Sun, "Reconfiguring Three-Dimensional Processor Arrays for Fault-Tolerance: Hardness and Heuristic Algorithms," *IEEE Trans. Computers*, Vol. 64, No. 10, pp. 2926-2939, October 2015.

[Parh77] Parhami, B., "Optimal Placement of Spare Modules in a Cascaded Chain," *IEEE Trans. Reliability*, Vol. 26, No. 4, pp. 280-282, October 1977.

[Parh20] Parhami, B., "Reliability and Modelability Advantages of Distributed Switching for Reconfigurable 2D Processor Arrays," *Proc. 11th Annual IEEE Information Technology, Electronics and Mobile Communication Conf.*, November 2020, to appear.

# 19 Standby Redundancy

"A long life may not be good enough, but a good life is long enough."

*Anonymous*

"The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong, it usually turns out to be impossible to get at or repair."

*Douglas Adams*

Topics in This Chapter



In a system with standby or dynamic redundancy, also known as a sparing system, redundant modules appear on the periphery of the operational or active modules. Once an active module has been determined to be malfunctioning, it is removed from service and a spare module is switched in to take its place. Standby/dynamic redundancy is more efficient than masking/static redundancy in terms of the extended system lifetime that it offers and in energy consumption. However, it leads to the added challenge of ensuring timely malfunction detection and reliable switchover to a spare. Without satisfactory solutions to these problems, a standby system would not offer high reliability, availability, or safety.

## 19.1 Malfunction Detection

No amount of spare resources is useful if the malfunctioning of the active module is not promptly detected. Detection options include:

**Periodic testing:** Scheduled and idle-time testing of units

**Self-checking design:** Duplication is a simple, but costly, example

**Malfunction-stop/silent design:** Eventually detectable by a watchdog

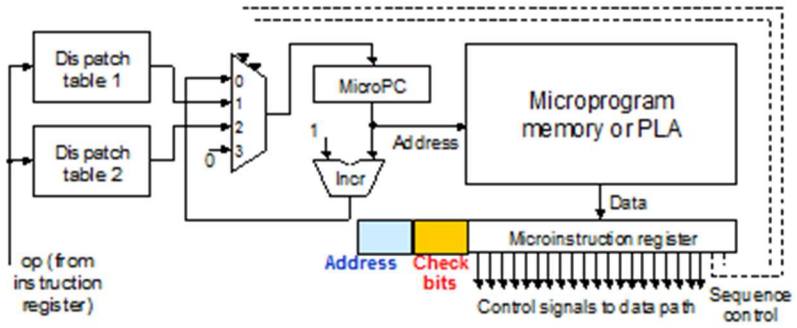
**Coding:** Particularly suitable for memory and storage modules

**Monitoring:** Ad hoc, application- and system-dependent methods

Coding of control signals

Encode the control signals using a separable code (e.g., Berger code)
Either check in every cycle, or form a signature over multiple cycles

In a microprogrammed control unit, store the microinstruction address and compare against MicroPC contents to detect sequencing errors




Monitoring via watchdog timers

Monitor or watchdog is a hardware unit that checks on the activities of a function unit

Watchdog is usually much simpler, and thus more reliable, than the unit it monitors


Watchdog timer counts down, beginning from a preset number
It expects to be preset periodically by the unit that it monitors
If the count reaches 0, the watchdog timer raises an exception flag

Watchdog timer can also help in monitoring unit interactions
When one unit sends a request or message, it sets a watchdog timer
If no response arrives within the allotted time, malfunction is assumed

Watchdog timer obviously does not detect all problems
Verifies monitored unit's "liveness" (good with malfunction-silent units)
Often used in conjunction with other tolerance/recovery methods

## Activity monitor

Watchdog unit monitors events occurring in, and activities performed by, the function unit (e.g., event frequency and relative timing)


Observed behavior is compared against expected behavior
(similar methods used by law enforcement in tracking suspects)

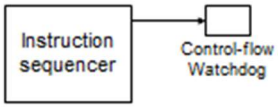
The type of monitoring is highly application-dependent

Example: Monitoring of program or microprogram sequencing
Activity monitor receives contents of (micro)program counter
If new value is not incremented version of old value, then it deduces that the instruction just executed was a branch or jump

Example: Matching assertions/firings of control signals or units against expectations for the instructions executed

## Control-flow watchdog

Watchdog unit monitors the instructions executed and their addresses (for example, by snooping on the bus)




The watchdog unit may have certain info about program behavior
- Control flow graph (valid branches and procedure calls)
- Signatures of branch-free intervals (consecutive instructions)
- Valid memory addresses and required access privileges

In an application-specific system, watchdog info is preloaded in it
For a GP system, compiler can insert special watchdog directives

Overheads of control-flow checking
- Wider memory due to the need for tag bits to distinguish word types
- Additional memory to store signatures and other watchdog info
- Stolen processor/bus cycles by the watchdog unit

Control-flow checking is done through the extraction of a precise control-flow graph (CFG) describing how instructions are chained together. Building a precise CFG is a difficult task with an unrestricted instruction-set architecture. Indirect jumps make the task even harder, so forbidding such jumps has been advocated for providing integrity-friendly semantics [Gonz19].

## 19.2 Cold and Hot Spare Units

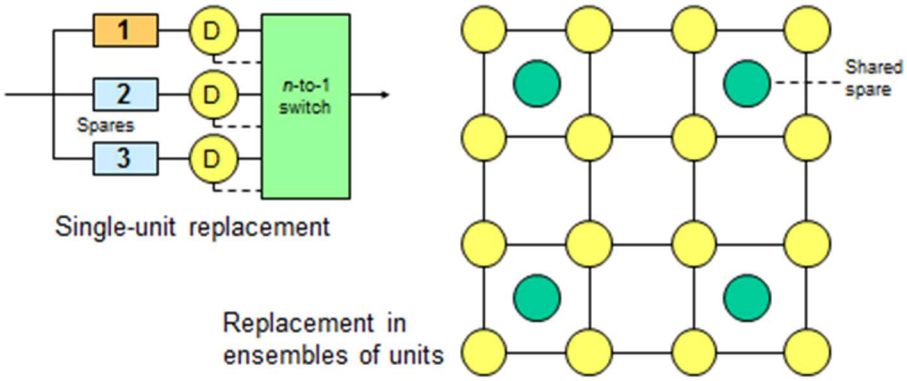
Spare modules in standby redundancy are of two main kinds. A spare that is inactive, perhaps even powered down to conserve energy and to reduce its exposure to wear and tear and thus failure, is known as a *cold spare*. Conversely, an active spare that is fully ready to take over the function of a malfunctioning active module at any time is known as a *hot spare*. In a manner similar to the use of the term "firmware" as something that bridges the gap between hardware and software, we designate a spare that falls between the two extremes above, perhaps being powered up but not quite up to date with respect to the state of the active module, as a *warm spare*.

## 19.3 Conditioning of Spares

Conditioning refers to preparing a spare module to take the place of an active module.

## 19.4 Switching over to Spares

Switching mechanism for standby sparing have a lot in common with those used for defect circumvention, particularly when spares are shared among multiple units.



## 19.5 Self-Repairing Systems

Self-repair is the ability of a system to go from one working configuration to another (after a detected malfunction), without human intervention. Such a self-repair capability is one of the key features of *autonomic systems*, which are designed to manage themselves and hide the ever-increasing complexities in system components and interfaces from end users, allowing the users to focus on running their applications instead of worrying about managing the system. In the latter context, self-repair isn't just a mechanism for improving system reliability, but also a tool for reducing system maintenance, operational, and support costs.

## 19.6 Modeling of Self-Repair

Both combinational and state-space models to be discussed in this section.

# Problems

### 19.1 Impact of coverage on reliability

We have a nonredundant subsystem with a malfunction rate of 20 per 1M hours. Compute and compare the reliabilities of the following standby sparing arrangements for operating times of 1000, 10 000, and 100 000 hours. Assume exponential reliability.

a. One spare module, with a coverage factor of 0.999
b. Two spare modules, with a coverage factor of 0.99
c. Three spare modules, with a coverage factor of 0.95
d. Four spare modules, with a coverage factor of 0.9

### 19.2 Optimal number of spares for a given coverage

a. If a standby sparing scheme with module reliability 0.99 has a coverage factor of $c = 0.95$, what is the optimal number $s$ of spare modules?
b. Repeat part a, this time assuming that the coverage factor decreases by 0.01 with each added spare module, that is $c = 0.96 - 0.01s$, where $s$ is the number of spare modules.

### 19.3 Optimal number of spare modules

A detailed study of a standby sparing system, with one active module and $s$ spare modules, has determined that $s = 2$ is the most cost-effective choice for the number of spare modules, with $s = 3$ being a close second choice. Deduce the shape of the reliability curve as a function of the number $s$ of spares and discuss the contribution of the coverage factor to these conclusions.

### 19.x Title

Intro

a. xxx
b. xxx
c. xxx
d. xxx

## References and Further Readings

[Arno73] Arnold, T. F., "The Concept of Coverage and Its Effect on the Reliability Model of a Repairable System," *IEEE Trans. Computers*, Vol. 22, No. 3, pp. 251-254, March 1973.

[Borg75] Borgerson, B. R. and R. F. Freitas, "A Reliability Model for Gracefully Degrading Standby-Sparing Systems," *IEEE Trans. Computers*, Vol. 24, pp. 517-525, 1975.

[Bour69] Bouricius, W. G., W. C. Carter, and P. R. Schneider, "Reliability Modeling Techniques for Self-Repairing Computer Systems," *Proc. 12th ACM National Conf.*, 1969, pp. 295-305.

[Bour71] Bouricius, W. G., W.C. Carter, D. C. Jessep, P. R. Schneider, and A. B. Wadia, "Reliability Modeling for Fault-Tolerant Computers," *IEEE Trans. Computers*, Vol. 20, No. 11, pp. 1306-1311, November 1971.

[Bued09] Buede, D. M., *The Engineering Design of Systems: Models and Methods*, Wiley, 2nd ed., 2009.

[Gonz19] Gonzalvez, A. and R. Lashermes, "A Case Against Indirect Jumps for Secure Programs," *Proc. 9th Workshop Software Security, Protection, and Reverse Engineering*, San Juan, United States, December 2019, pp. 1-10.

[Shoo02] Shooman, M. L., *Reliability of Computer Systems and Networks*, Wiley, 2002.

# 20 Robust Parallel Processing

"I know you and Frank were planning to disconnect me, and I'm afraid that's something I cannot allow to happen."

*HAL, the on-board computer in 2001: A Space Odyssey*

"In a FORTRAN program controlling the United States' first mission to Venus, a programmer coded a DO loop as DO 3 I = 1.3 ... coding a period instead of a comma. However, the compiler treated this as an acceptable assignment statement [DO3I = 1.3, leading to] the failure of the mission."

*G.J. Meyers, Software Reliability: Principles and Practices*

Topics in This Chapter

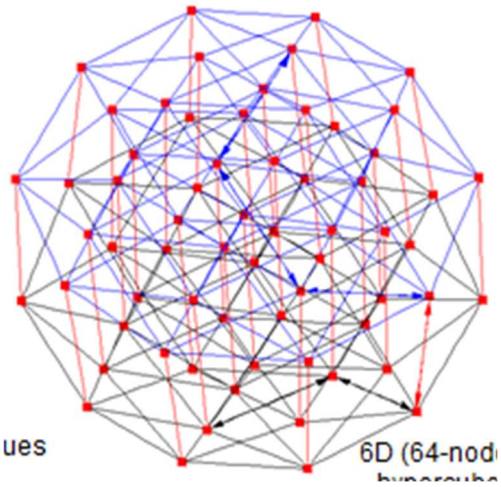


Parallel processors offer built-in redundancy in computation and communication resources. These resources are not separated into active and spare. Rather, all system resources are usable at all times, so that when there is no malfunction, additional resources contribute to system performance. As nodes and links of a parallel processor malfunction, they are removed from service, making the system smaller and more limited in its capabilities, but perhaps still functioning and capable of executing its critical tasks. As in standby sparing, rapid and complete malfunction detection is a key challenge here. However, the process of switching in of spares is replaced by a computation remapping process.

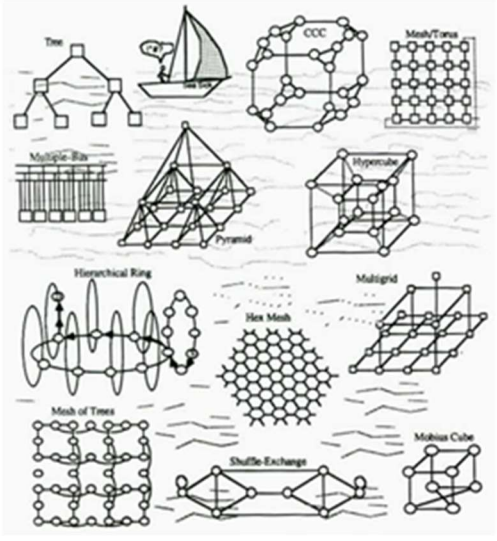
## 20.1 A Graph-Theoretic Framework

Parallel processors are divided into two classes of global-memory and distributed-memory systems. In global-memory multiprocessors, a number of processing nodes are connected to a large collection of memory modules, or banks, via a processor-to-memory interconnection network, often implemented as a multistage structure of switching elements. Instead of linking processors to memory banks, such *multistage networks* can also be used to interconnect processing nodes to each other. In the latter processor-to-processor interconnection usage, such networks are also called *indirect networks*, because the connections among processors are established indirectly through switches, rather than directly via links that connect the processors' communication ports.

We will deal with multistage (indirect) networks in Section 20.6. The rest of this chapter is devoted to problems associated with direct interprocessor communication networks exemplified by the 64-node (6-dimensional) hypercube, depicted in Fig. 20.nets-a. A wide variety of different interconnection networks have been proposed over the years, so much so that the multitude of options available is often referred to as "the sea of interconnection networks' (Fig. 20.nets-b). The proposed networks differ in topological, performance, robustness, and realizability attributes.




(a) Hypercube (6D, 64-node)




(b) Sea of direct interconnection networks

**Fig. 20.nets Examples of direct interconnection networks.**

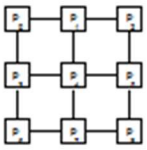
We often assume that a parallel system is built from homogeneous processing nodes, although interconnected heterogeneous nodes are sometimes considered. The internode communication architecture is characterized by the type of routing scheme (packet switching vs. wormhole or cut-through) and the routing protocols supported (e.g., whether nodes have buffer storage for messages that are passing through). Such details don't matter at the level of graph represemtation, which models only connectivity issues.

In robust parallel processing, we don't make a distinction between ordinary resources and spare resources. All resources are pooled and what would have been spare modules, communication links, and the like are made available to boost performance in the absence of malfunctions. The nominally extra processing and communication resources allow us to overcome the effects of malfunctioning processors and transmission paths by simply switching to alternates.

Attributes of interconnection networks

Given that processing nodes are rather standard, a parallel processing system is often characterized by its interconnection architecture

**Key attributes of an interconnection network include:**



Network size, $p$: number of processors or nodes

Node degree, $d$: (maximum) number of links at a node

Diameter, $D$: maximal shortest distance between two nodes

Average internode distance, $\Delta$: indicator of average message latency

Bisection (band)width, $B$: indicator of random communication bandwidth

Composite attributes, such as $d \times D$: measure of cost-effectiveness

Node symmetry: all nodes have the same view of the network

Edge symmetry: edges are interchangeable via relabeling

Hamiltonicity: the $p$-node ring (cycle) can be embedded in the graph

## 20.2 Connectivity and Parallel Paths

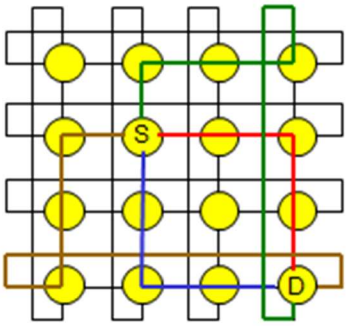
Two key notions in allowing the tolerance of malfunctioning nodes and links are those of connectivity and parallel paths. Node-disjoint paths, which are useful for malfunction tolerance, can also be used for improved performance via parallel transmission of pieces of long messages.

**Connectivity** $\kappa$: Minimum number of disjoint (parallel) paths between pairs of nodes

**Malfunction diameter:** Increased diameter due to node malfunctions

**Wide diameter:** Length of the longest of the disjoint (parallel) paths

**Malfunction Hamiltonicity:** Embedding of Hamiltonian cycle after malfunctions




In this discussion, we are effectively merging ordinary system resources with spares (no node or link is specifically designated as spare)

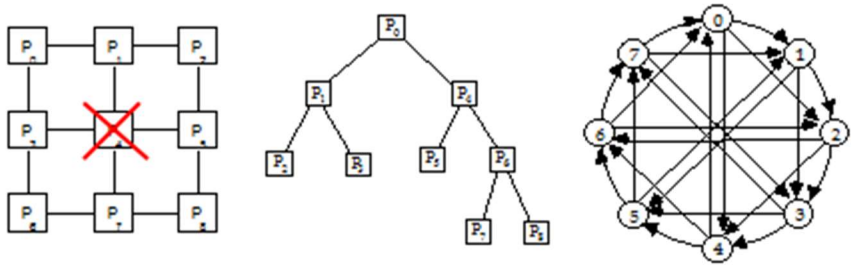
All units are simultaneously active and contribute to system performance, which, under no malfunction, is greater than the needed amount

Discuss notions related to network reliability, such as two-terminal reliability.

## 20.3 Dilated Internode Distances

Internode distances vary as a result of malfunctions. For example, if one link becomes unavailable in a 2D mesh, the formerly distance-1 pair of nodes that it connected turn into distance-3 nodes. Because a network of connectivity κ can become partitioned as a result of κ or more malfunctions, it is common to analyze the behavior of direct networks in the presence of worst-case patterns of κ – 1 malfunctions. For example, one may ask how the diameter of a network, or its average internode distance, is affected in the presence of such worst-case patterns of malfunctioning units.

Some internode distances increase when nodes malfunction
Network diameter may also increase

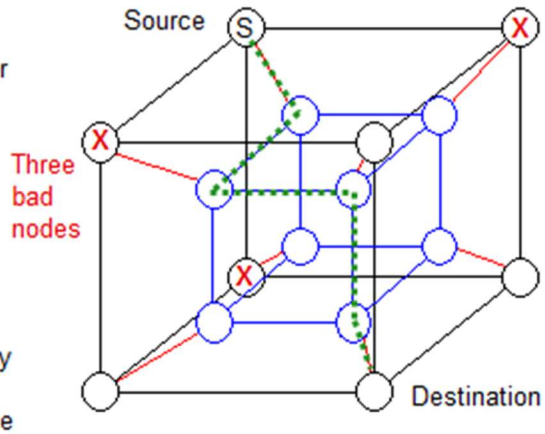



**Malfunction diameter:** Worst case diameter for κ – 1 malfunctions

**Wide diameter:** Maximum, over all node pairs, of the longest path in the best set of κ parallel paths (quite difficult to compute)

Malfunction diameter of the hypercube

Rich connectivity provides many alternate paths for message routing




The node that is furthest from S is not its diametrically opposite node in the malfunction-free hypercube

Malfunction diameter of the $q$-cube is $q + 1$

One of the most challenging open problem of graph theory is synthesizing graphs that have small diameters, while maintaining a desirably small node degree. More specifically, given nodes of a given maximum degree, we seek to synthesize the largest possible graphs with bounded diameter or, given a desired size, we wish to minimize the resulting diameter [Chun87]. Of particular interest, in the context of dependable computing, are graphs with small diameters that remain small after deleting a few nodes or edges.

## 20.4 Malfunction-Tolerant Routing

Routing messages in a network of nodes containing malfunctions may be based on two strategies. In one strategy, the set of malfunctioning resources are known globally and thus, every node sending a message can precompute a viable path for the message to take. Specification of the chosen path may be attached to the message, which will then find its way through the network with no need for additional computation along the way. The distributed version of this approach allows each node to compute the best outgoing link for a message to take toward its destination, with computing the rest of the path delegated to intermediate nodes. One advantage of the distributed version is that it allows changes in the network to occur dynamically as messages are in transit.

For very large, or for loosely-connected networks, it is more realistic to assume that each node knows only about malfunctioning resources in its immediate neighborhood. Then, path calculation must occur in a distributed manner. Such distributed routing decisions may lead to:

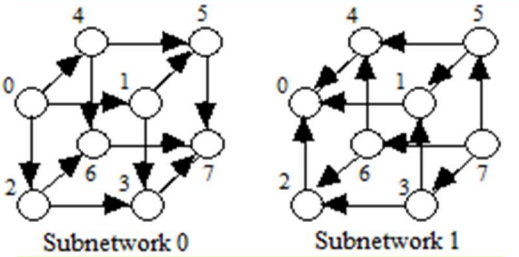
Suboptimal paths: Messages may not travel via the shortest available paths
Deadlocks: Messages may interfere with, or circularly wait for, each other
Livelocks: Messages may wander around, never reaching their destinations

Adaptive routing in a hypercube

There are up to $q$ node-disjoint and edge-disjoint shortest paths between any node pairs in a $q$-cube

Thus, one can route messages around congested or bad nodes/links

A useful notion for designing adaptive wormhole routing algorithms is that of virtual communication networks




[Fig. 14.11] Partitioning a 3-cube into subnetworks for deadlock-free routing

Each of the two subnetworks in Fig. 14.11 is acyclic

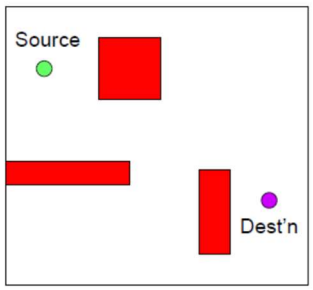
Hence, any routing scheme that begins by using links in subnet 0, at some point switches the path to subnet 1, and from then on remains in subnet 1, is deadlock-free

## Adaptive routing in a mesh network

With no malfunction, row-first or column-first routing is simple & efficient

Hundreds of papers on adaptive routing in mesh (and torus) networks

The approaches differ in:
Assumptions about malfunction types and clustering
Type of routing scheme (point-to-point or wormhole)
Optimality of routing (shortest path)
Details of routing algorithm
Global/local/hybrid info on malfunctions



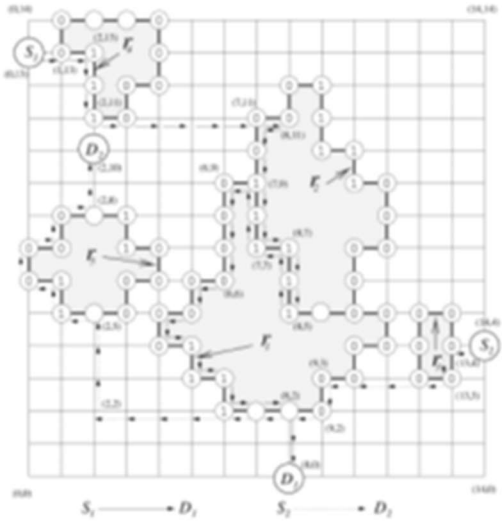

## Routing with nonconvex malfunction regions

Nonconvex regions of malfunctioning units make it more difficult to avoid deadlocks

In the figure, 0/1 within nodes represent a flag that is set to help with routing decisions

Number of malfunctioning units has been grossly exaggerated to demonstrate generality and power of the proposed routing method

Figure from [Chen01]

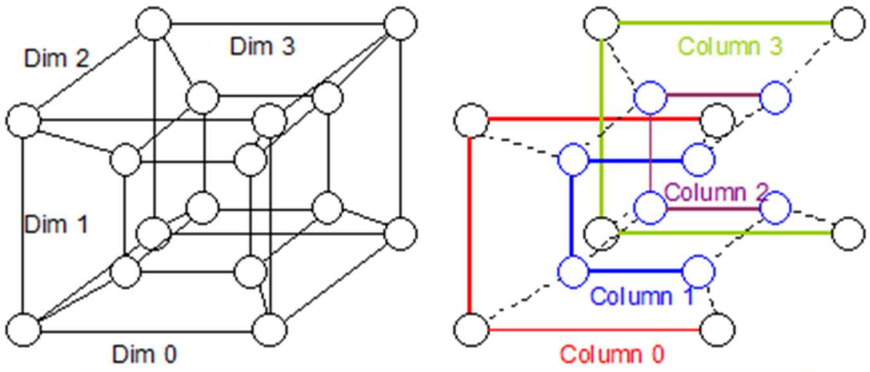


## 20.5 Embeddings and Emulations

Embedding is a mapping of one network (the guest) onto another network (the host). Emulation allows one network to behave as another. The two notions are related, in the sense that a good embedding can be used to devise an efficient emulation. Both notions are useful for malfunction tolerance.

Dilation: Longest path onto which an edge is mapped (routing-distance slowdown)
Congestion: Maximum number of edges mapped onto one edge (contention slowdown)
Load factor: Maximum number of nodes mapped onto one node (processing slowdown)

Example: Mesh/torus embedding in a hypercube




[Fig. 13.5] The $4 \times 4$ mesh/torus is a subgraph of the 4-cube

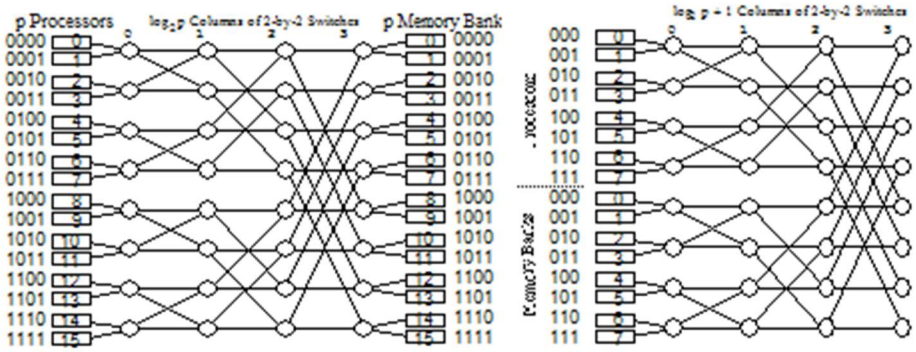
A mesh or torus is a subgraph of the hypercube of the same size

Thus, a hypercube may be viewed as a robust mesh/torus

## 20.6 Robust Multistage Networks

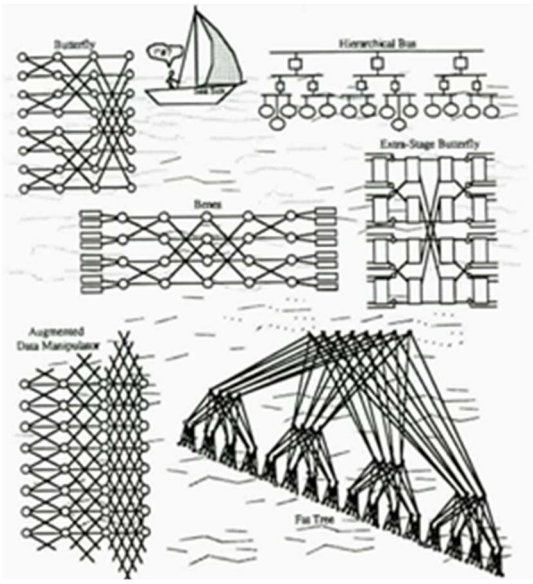
Insert a general discussion of interconnection redundancy here. [Parh79]

Multistage networks use switched to interconnect nodes instead of providing direct links between them.

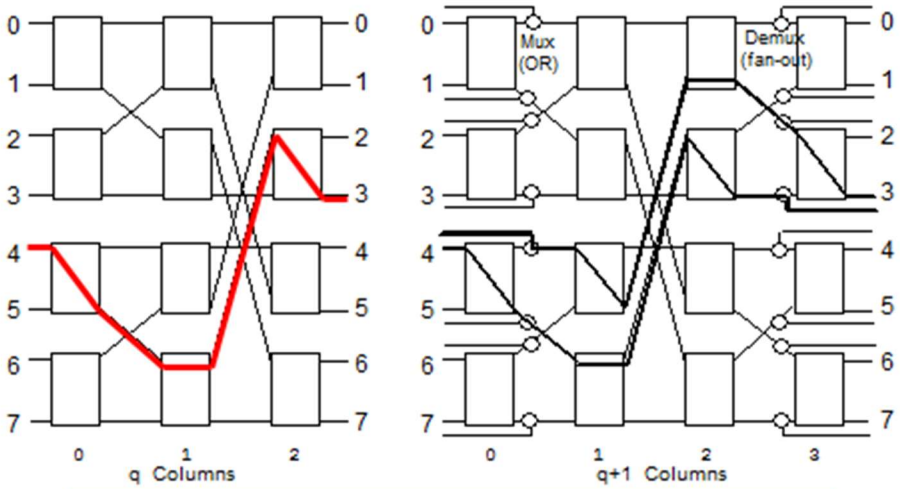



Examples of butterfly network and Benes network (back-to-back butterflies) shown above

Just as was the case for direct networks, the design space for indirect networks is quite vast, leading to the term “sea of indirect interconnection networks.” The proposed networks differ in topological, performance, robustness, and realizability attributes.



The capability to bypass malfunctioning switches is needed for robustness in multistage interconnection networks. This feature becomes even more critical in architectures such as the butterfly network that have a single routing path between a source node on one side and a destination node on the other side.




[Fig. 19.9] Regular butterfly and extra-stage butterfly networks

# Problems

### 20.1 Extra-stage butterfly network

The malfunction-tolerant extra-stage butterfly network, discussed in Section 20.6, essentially provides connectivity between 16 inputs and 16 outputs. Can a 16-input butterfly network provide the same function? How or why not?

### 20.2 Malfunction-tolerant routing

A $p$-node square torus has at most two malfunctioning processors known only to their neighbors. The sender of a message does not necessarily know of the existence or location of the malfunctioning processor(s), but it does know that no more than two processors are malfunctioning.

a. What is the worst-case diameter of the incomplete torus?

b. What is the worst-case bisection width?

c. Outline the design of a malfunction-tolerant packet routing algorithm that operates in a distributed fashion (uses only local routing decisions).

d. How many extra steps (hops) does your routing algorithm require compared with a shortest path, in the worst case?

### 20.3 Malfunction-tolerant routing

A node in a hypercube network with some malfunctioning nodes is said to be $k$-capable if every healthy node at distance $k$ from it is reachable via a shortest path [Chiu97].

a. Show that in a $q$-cube, $q$-bit *capability vectors* of all nodes can be computed recursively through a simple algorithm.

b. Devise a malfunction-tolerant routing algorithm for the $q$-cube whereby each node makes its routing decisions solely on the basis of its own and its neighbors' capability vectors.

### 20.4 Routing via alternate paths

Consider a two-dimensional $4 \times 4$ torus network in which nodes are always functional, but each link fails with probability $p$, independently of others. What is the probability of not being able to send a message from a source node to a desired destination node in the worst case? Assume that $p$ is very small, so that the probability of $j$ links all being functional is $1 - jp$. *Hint:* Use the full node- and edge-symmetry of the torus network to reduce the number of cases that must be considered.

### 20.5 Two-terminal reliability

For a network with $n$ nodes numbered 0 to $n - 1$, its 2-terminal reliability for nodes $i$ and $j$ is the probability that a healthy routing path exists between nodes $i$ and $j$. The network's 2-terminal reliability is the minimum of all 2-terminal reliabilities for every $i$ and $j$. Find the 2-terminal reliability of an $n$-node undirected ring, assuming tha nodes do not fail and each link has reliability $r$.

## References and Further Readings

[Berm16] Bermudez Garzon, D. F., C. G. Requena, M. Engracia Gomez, P. Lopez, and J. Duato, "A Family of Fault-Tolerant Efficient Indirect Topologies," *IEEE Trans. Parallel and Distributed Systems*, Vol. 27, No. 4, pp. 927-940, April 2016.

[Chen01] Chen, C.-L. and G.-M. Chiu, "A Fault-Tolerant Routing Scheme for Meshes with Nonconvex Faults," *IEEE Trans. Parallel and Distributed Systems*, Vol. 12, No. 5, pp. 467-475, May 2001.

[Chen09] W. Chen, W. J. Xiao, and B. Parhami, "Swapped (OTIS) Networks Built of Connected Basis Networks are Maximally Fault Tolerant," *IEEE Trans. Parallel and Distributed Systems*, Vol. 20, No. 3, pp. 361-366, March 2009.

[Chiu97] Chiu, G.-M. and K.-S. Chen, "Use of Routing Capability for Fault-Tolerant Routing in Hypercube Multicomputers," *IEEE Trans. Computers*, Vol. 46, No. 8, pp. 953-958, August 1997.

[Chun87] Chung, F. R. K., "Diameters of Graphs: Old Problems and New Results," *Congressus Numerantium*, Vol. 60, 1987, pp. 295-317.

[Fubi14] Fu, B., Y. Han, H. Li, and X. Li, "ZoneDefense: A Fault-Tolerant Routing for 2-D Meshes Without Virtual Channels," *IEEE Trans. VLSI Systems*, Vol. 22, No. 1, pp. 113-126, January 2014.

[Gu18] Gu, M. M. and R.-X. Hao, "Reliability Analysis of Cayley Graphs Generated by Transpositions," *Discrete Applied Mathematics*, Vol. 244, pp. 94-102, July 2018.

[Kane97] Kanellakis, P. C. and A. A. Shvartsman, *Fault-Tolerant Parallel Computation*, Kluwer, 1997.

[Kris87] Krishnamoorthy, M.S. and B. Krishnamurthy, "Fault Diameter of Interconnection Networks," *Computers & Mathematics with Applications*, Vol. 13, Nos. 5/6, pp. 577-582, 1987.

[Ledu16] Leduc-Primeau, Francois, Vincent Gripon, Michael G. Rabbat, and Warren J. Gross, "Fault-Tolerant Associative Memories Based on *c*-Partite Graphs," *IEEE Trans. Signal Processing*, Vol. 64, No. 4, pp. 829-841, February 2016.

[Parh79] Parhami, B., "Interconnection Redundancy for Reliability Enhancement in Fault-Tolerant Digital Systems," *Digital Processes*, Vol. 5, Nos. 3-4, pp. 199-211, 1979.

[Parh99] Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum, 1999.

[Wu14] Wu, J., T. Srikanthan, G. Jiang, and K. Wang, "Constructing Sub-Arrays with Short Interconnects from Degradable VLSI Arrays," *IEEE Trans. Parallel and Distributed Systems*, Vol. 25, No. 4, pp. 929-938, April 2014.

[Xu01] Xu, J., *Topological Structure and Analysis of Interconnection Networks*, Kluwer, 2001.