Dependable Computing

A Multilevel Approach

Behrooz Parhami

University of California, Santa Barbara

STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals / Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods / Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods / Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods / Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods / Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Robust Parallel Processing
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods / Examples	21. Degradation Allowance 22. Degradation Management 23 Resilient Algorithms 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods / Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

Appendix: Past, Present, and Future

About This Presentation

This presentation is intended to support the use of the textbook *Dependable Computing: A Multilevel Approach* (traditional print or on-line open publication, TBD). It is updated regularly by the author as part of his teaching of the graduate course ECE 257A, Fault-Tolerant Computing, at Univ. of California, Santa Barbara. Instructors can use these slides freely in classroom teaching or for other educational purposes. Unauthorized uses, including distribution for profit, are strictly prohibited.

Edition	Released	Revised	Revised	Revised	Revised
First	Sep. 2006	Oct. 2007	Nov. 2009	Nov. 2012	Nov. 2013
		Feb. 2015	Nov. 2015	Nov. 2018	Nov. 2019
		Nov. 2020			

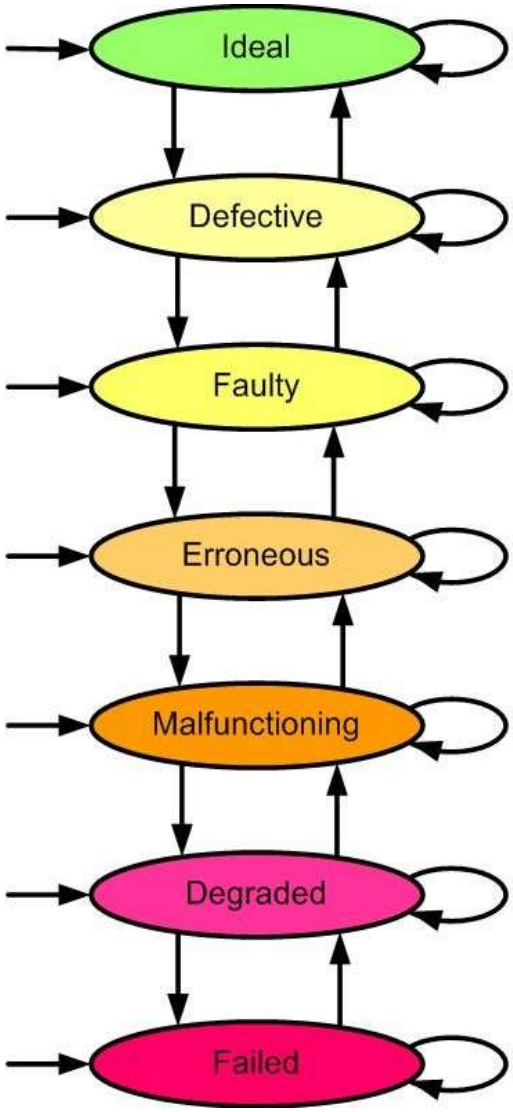
17 Malfunction Diagnosis

"It's not our fault. We attribute your poor portfolio performance to fund malfunction."

STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals / Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods / Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods / Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods / Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods / Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Robust Parallel Processing
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods / Examples	21. Degradation Allowance 22. Degradation Management 23. Resilient Algorithms 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods / Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

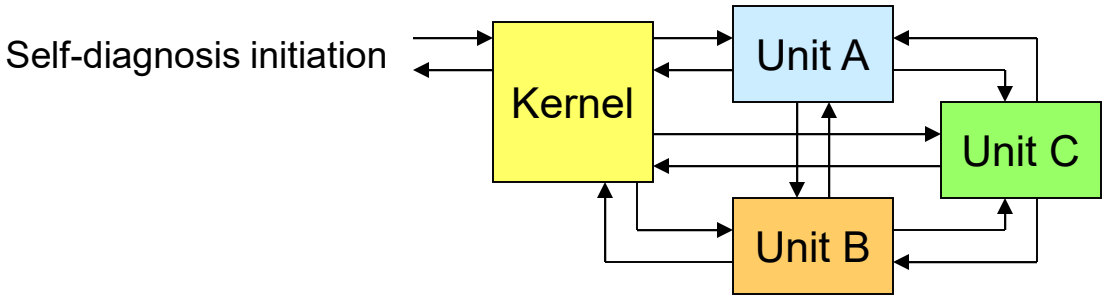
Appendix: Past, Present, and Future



17.1 Self-Diagnosis in Subsystems

Layered approach:

- A small part of a unit is tested, which then forms a trusted kernel
- The trusted kernel is used to test the next layer of subsystems
- Region of trust is gradually extended, until it covers the entire unit



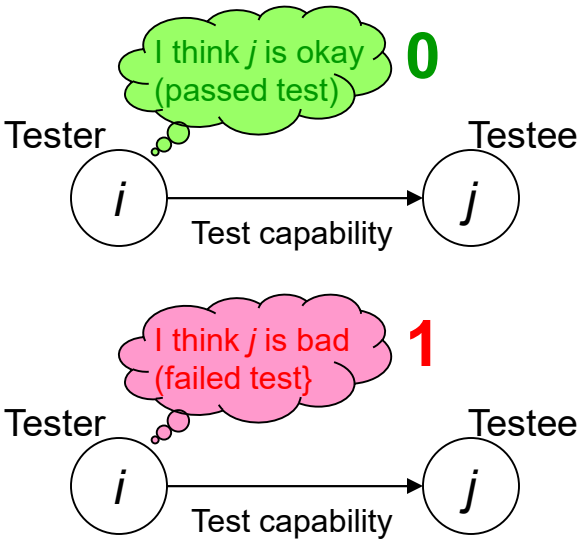
One approach to go/no-go testing based on self-diagnosis

- Tester supplies a random seed to the built-in test routine
- The test routine steps through a long computation that exercises nearly all parts of the system, producing a final result
- The tester compares the final result to the expected result

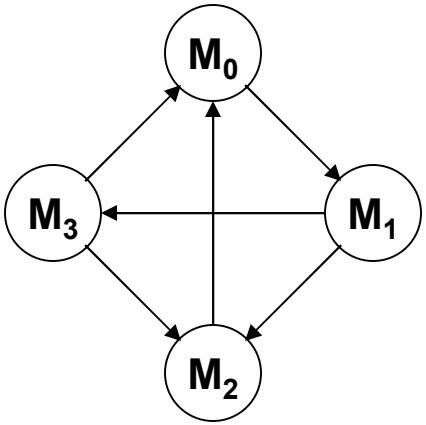
Ideally, if a properly designed self-test routine returns a 32-bit value, the value will match the expected result despite the presence of faults with probability $2^{-32} \approx 10^{-9.6} \rightarrow$ test coverage = $1 - 10^{-9.6}$

17.2 Malfunction Diagnosis Model

Diagnosis of one unit by another
- The tester sends a self-diagnosis request, expecting a response
- The unit under test eventually sends some results to the tester
- The tester interprets the results received and issues a verdict



Testing capabilities among units is represented by a directed graph



The verdict of unit i about unit j is denoted by $D_{ij} \in \{0, 1\}$
All the diagnosis verdicts constitute the $n \times n$ diagnosis matrix D
The diagnosis matrix D is usually quite sparse

More on Terminology and Assumptions

Malfunction diagnosis in our terminology corresponds to "system-level fault diagnosis" in the literature

The qualification "system-level" implies that the diagnosable units are subsystems with significant computational capabilities (as opposed to gates or other low-level components)

We do not use the entries on the main diagonal of the diagnosis matrix *D* (a unit does not judge itself) and we *usually* do not let two units test one another

$$\begin{bmatrix} -- & D_{01} & -- & -- \\ -- & -- & D_{12} & D_{13} \\ D_{20} & -- & -- & -- \\ D_{30} & -- & D_{32} & -- \end{bmatrix}$$

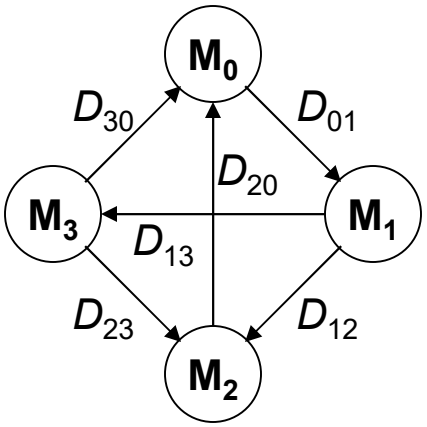
A good unit always issues a correct verdict about another unit (i.e., tests have perfect coverage), but the verdict of a bad unit is arbitrary and cannot be trusted

This is known as the PMC model (Preparata, Metze, & Chien)

We consider the PMC model only, but other models also exist (e.g., in comparison-based models, verdicts are derived from comparing the outputs of unit pairs)

17.3 One-Step Diagnosability

Consider this system, with the test outcomes shown



Diagnosis syndromes

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

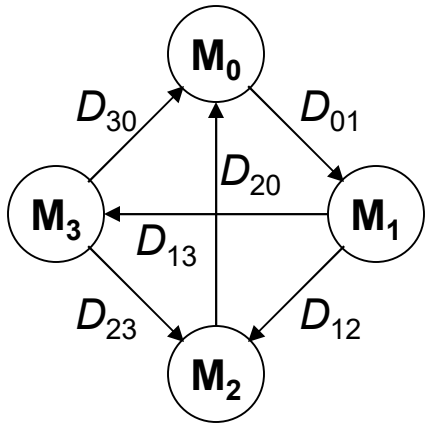
Syndrome dictionary:

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

We say that the system above is 1-step 1-diagnosable (we can correctly diagnose up to 1 malfunctioning unit in a single round of tests)

Requirements for One-Step *t*-Diagnosability

An *n*-unit system is 1-step *t*-diagnosable if the diagnosis syndromes for conditions involving up to *t* malfunctions are all distinct



Necessary conditions:

1. $n \geq 2t + 1$; i.e., a majority of units must be good
2. Each unit must be tested by at least *t* other units

Sufficient condition:

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

So, each unit being tested by *t* other units is both necessary and sufficient

The system above, has each unit tested by 1 or 2 units; it is 1-step 1-diagnosable

It cannot be made 1-step 2-diagnosable via adding more test connections

Analogy: “Liars and Truth-Tellers” Puzzles

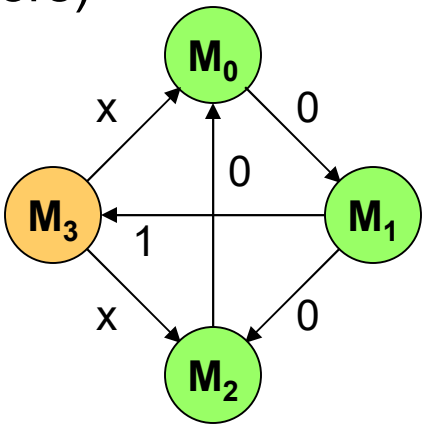
You visit an island whose inhabitants are from two tribes
Members of one tribe (“liars”) consistently lie
Members of the other tribe (“truth-tellers”) always tell the truth

You encounter a person on the island
What single yes/no question would you ask him to determine his tribe?

More generally, how can you derive correct conclusions from info provided by members of these tribes, without knowing their tribes?

How would the problem change if the two tribes were “truth-tellers” and “randoms” (whose members give you random answers)

In the context of malfunction diagnosis:
Truth-tellers are akin to good modules
Randoms correspond to bad modules
You do not know whether a module is good or bad
Module “opinions” about other modules must be used to derive correct diagnoses



1-Step Diagnosability: Analysis & Synthesis

Analysis problems:

1. Given a directed graph defining the test links, find the largest value of t for which the system is 1-step t-diagnosable (easy if no two units test one another; fairly difficult, otherwise)
2. Given a directed graph and its associated test outcomes, identify all the malfunctioning units, assuming there are no more than t

There is a vast amount of published work dealing with Problem 1

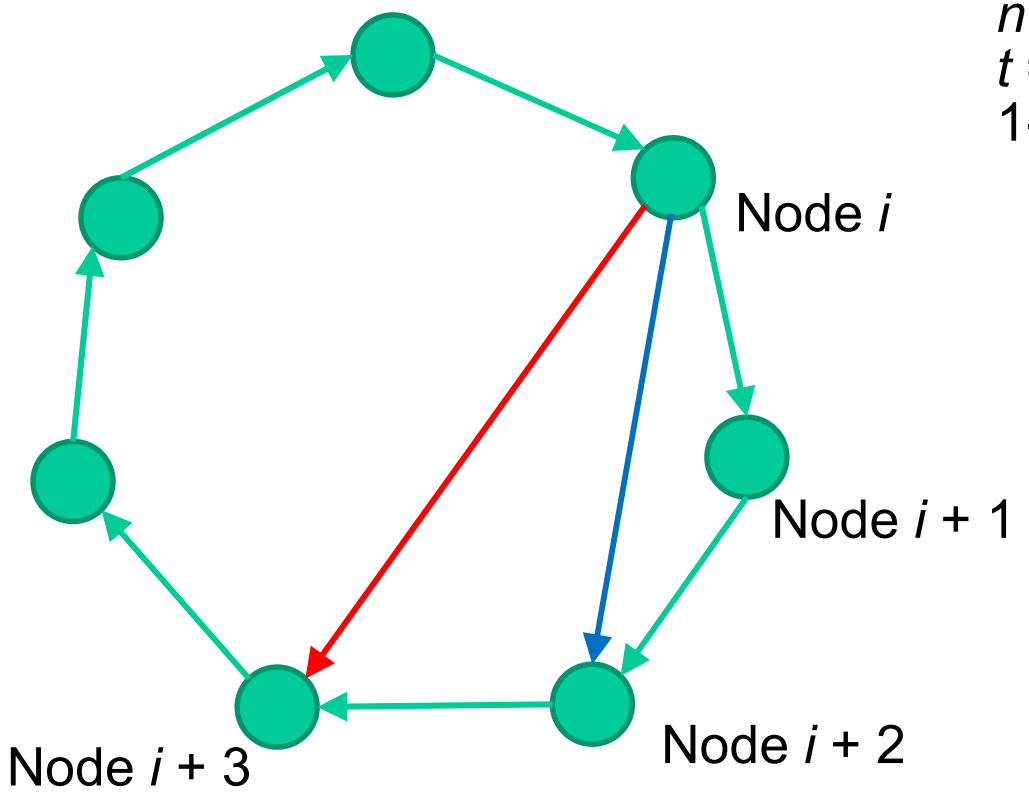
Problem 2 arises when we want to repair or reconfigure a system using test outcomes (solved via table lookup or analytical methods)

Synthesis problem:

Specify the test links (connection assignment) that makes an n-unit system 1-step t-diagnosable; use as few test links as possible

A degree-t directed chordal ring, in which node i tests the t nodes $i + 1, i + 2, \ldots, i + t$ (all mod n) has the required property

An *n*-node, 1-Step *t*-Diagnosable System

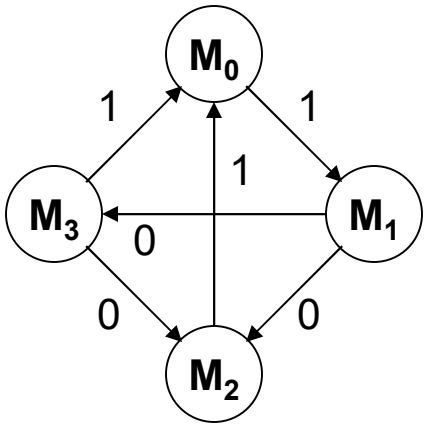


n = 7
t = 3
1-step 3-diagnosable

A degree-*t* directed chordal ring, in which node *i* tests the *t* nodes *i* + 1, *i* + 2, . . . , *i* + *t* (all mod *n*) has the required property

An O(n^3)-Step Diagnosis Algorithm

Input: The diagnosis matrix
Output: Every unit labeled G or B
while some unit remains unlabeled repeat
    choose an unlabeled unit and label it G or B
    use labeled units to label other units
    if the new label leads to a contradiction
    then backtrack
    endif
endwhile



$$\begin{bmatrix} -- & D_{01} & -- & -- \\ -- & -- & D_{12} & D_{13} \\ D_{20} & -- & -- & -- \\ D_{30} & -- & D_{32} & -- \end{bmatrix}$$

More efficient algorithms exist

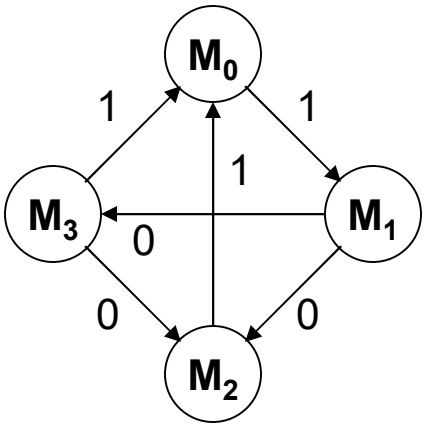
1-step 1-diagnosable system

M_0 is G (arbitrary choice)
M_1 is B
M_2 is B (contradiction, 2 Bs)

M_0 is B (change label)
M_1 is G (arbitrary choice)
M_2 is G
M_3 is G

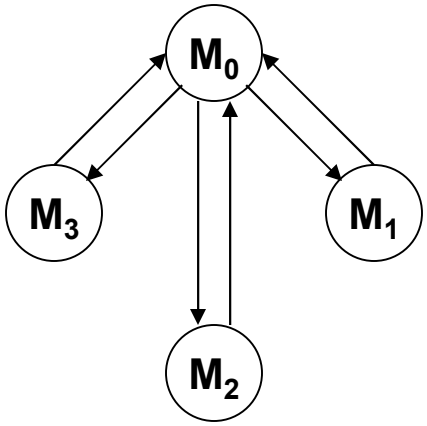
An O($n^{2.5}$)-Step Diagnosis Algorithm

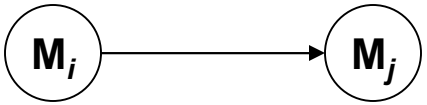
From the original testing graph, derive an L-graph
The L-graph has the same nodes
There is a link from node *i* to node *j* in the L-graph
iff node *i* can be assumed to be malfunctioning
when node *j* is known to be good



Testing graph
and test results

Corresponding L-graph





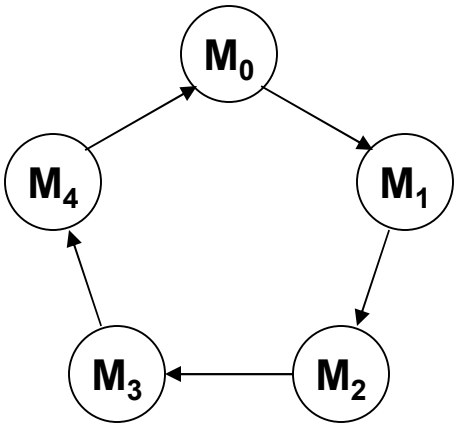
$\mathbf{M}_j$ good $\rightarrow$ $\mathbf{M}_i$ malfunctioning

Definition – *Vertex cover* of a graph:
A subset of vertices that contains at least
one of the two endpoints of each edge

Theorem: The unique minimal vertex cover of the L-graph is the set of *t* or fewer malfunctioning units

17.4 Sequential Diagnosability

An *n*-unit system is sequentially *t*-diagnosable if the diagnosis syndromes when there are *t* or fewer malfunctions are such that they always identify, unambiguously, at least one malfunctioning unit



This is useful because some systems that are not 1-step *t*-diagnosable are sequentially *t*-diagnosable, and they can be restored by removing the identified malfunctioning unit(s) and repeating the process

This system is sequentially 2-diagnosable

In one step, it is only 1-diagnosable

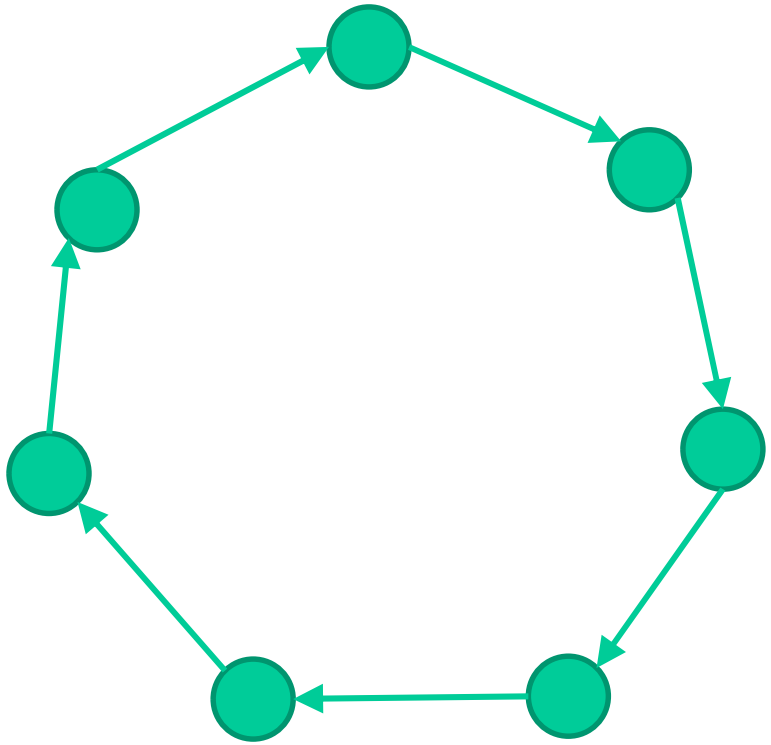
Necessary condition:

$n \geq 2t + 1$; i.e., a majority of units must be good

Sequential diagnosability of directed rings:

An *n*-node directed ring is sequentially *t*-diagnosable for any *t* that satisfies $\lceil (t^2 - 1)/4 \rceil + t + 2 \leq n$

Sequential Diagnosability of Directed Rings



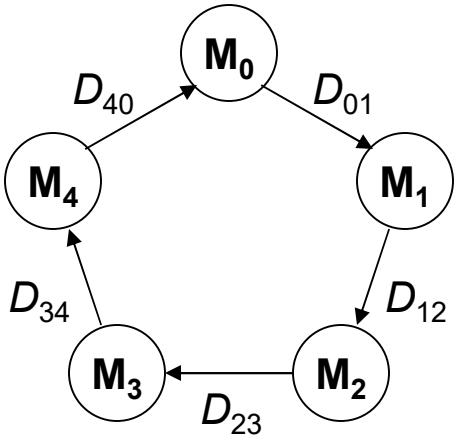
$n = 7$
1-step 1-diagnosable
Seq. 3-diagnosable

$\lceil (3^2 - 1)/4 \rceil + 3 + 2 \leq 7$

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

Sequential 2-Diagnosability Example

Consider this system, with the test outcomes shown



Malfunction syndromes (x means 0 or 1)

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

The system above is sequentially 2-diagnosable (we can correctly diagnose up to two malfunctioning units, but only one at a time)

Sequential Diagnosability: Analysis & Synthesis

Analysis problems:

1. Given a directed graph defining the test links, find the largest value of t for which the system is sequentially t-diagnosable
2. Given a directed graph and its associated test outcomes, identify at least one malfunctioning unit (preferably more), assuming there are no more than t

These problems have been extensively studied

Synthesis problem:

Specify the test links (connection assignment) that makes an n-unit system 1-step t-diagnosable; use as few test links as possible

An n-node ring, with $n \geq 2t + 1$, 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

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 $s \geq t$

The special case of 1-step t/t-diagnosability has been widely studied

Given the values of t and s, the problem of deciding whether a system is t/s-diagnosable is co-NP-complete

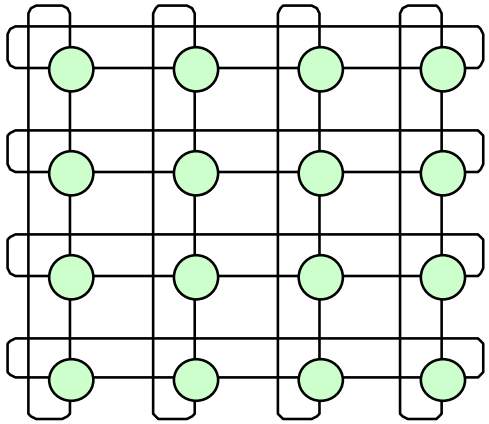
However, there exist efficient, polynomial-time, algorithms to find the largest integer t such that the system is t/t- or $t/(t + 1)$-diagnosable

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 $r < t$

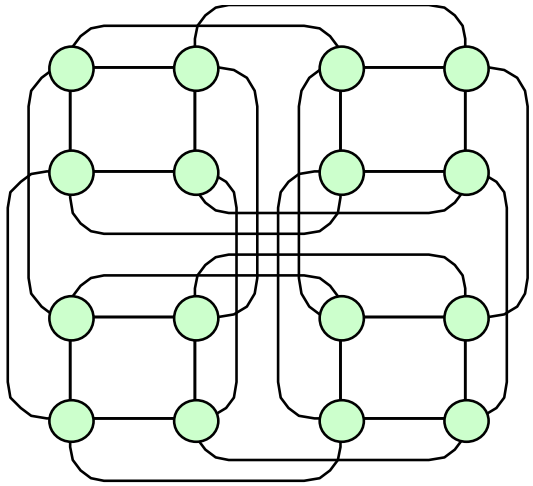
Safe diagnosability: Up to t' malfunctions are correctly diagnosed and up to u detected (no danger of incorrect diagnosis for up to u malfunctions; reminiscent of combo error-correcting/detecting codes)

17.6 Other Topics in Diagnosability

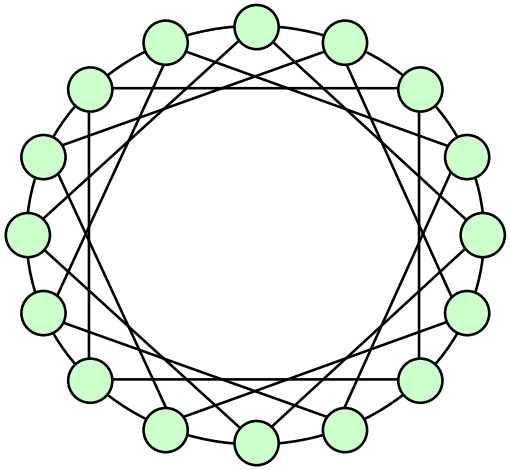
Diagnosability results have been published for a variety of regular interconnection networks



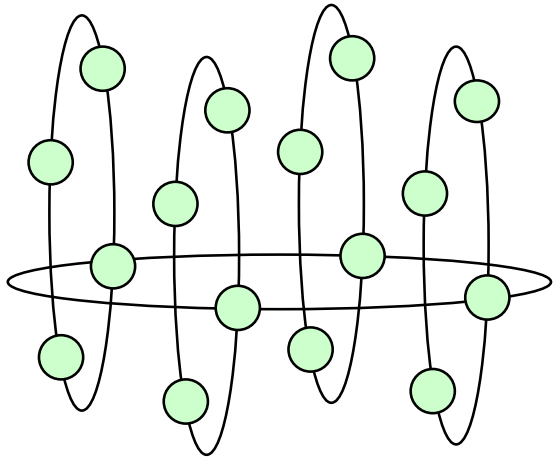
(a) 2D torus



(b) 4D hypercube



(c) Chordal ring



(d) Ring of rings

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:

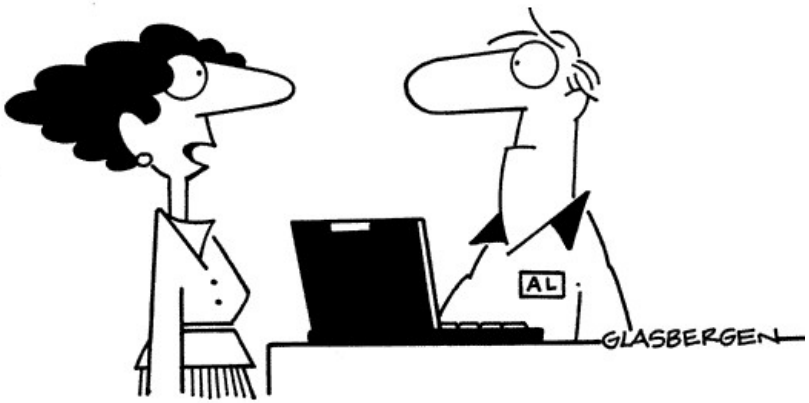
1. Recovering state info from removed modules or back-up storage
2. Reassigning tasks and reallocating data
3. Restarting the computation from last checkpoint or from scratch

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

18 Malfunction Tolerance

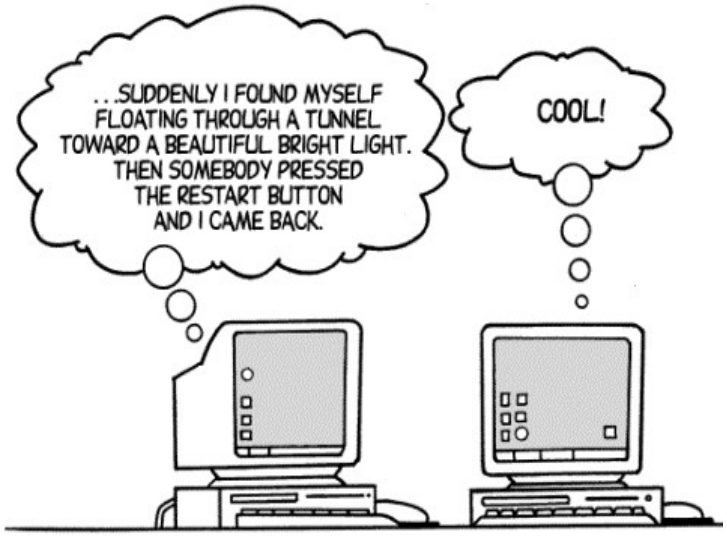
Computer Repairs

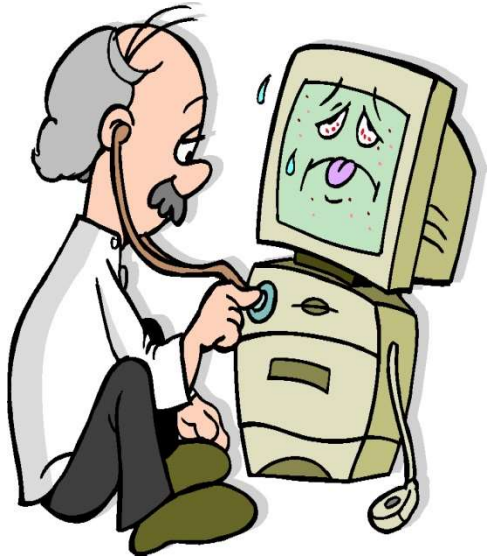


"When I'm away from my desk it goes into sleep mode...but the snoring annoys my coworkers!"

"Crashing is an expression of hostility against your network administrator. Though you appear to be uncooperative, it's actually a desperate cry for help."

www.generalcomics.com

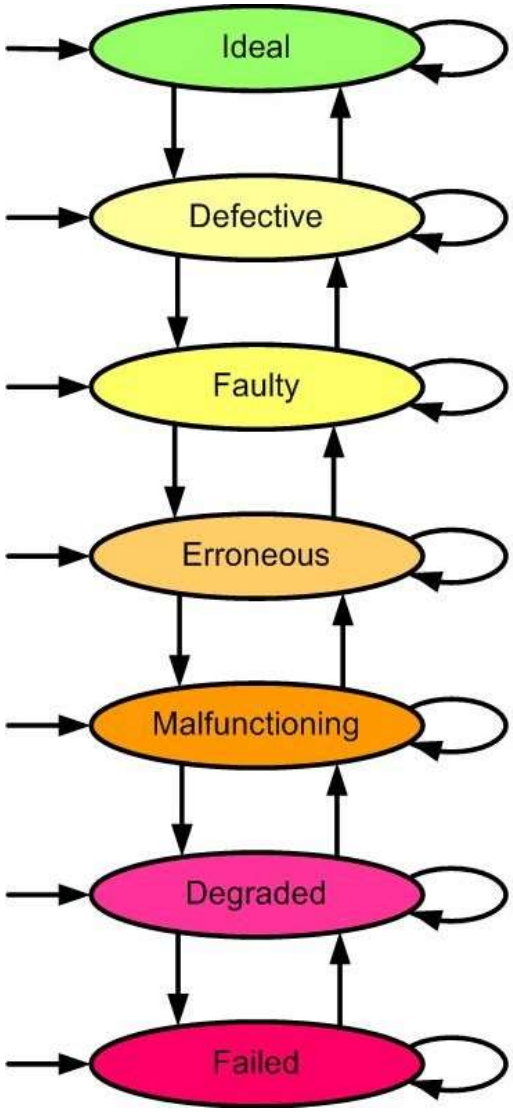




STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals / Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods / Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods / Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods / Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods / Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Robust Parallel Processing
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods / Examples	21. Degradation Allowance 22. Degradation Management 23. Resilient Algorithms 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods / Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

Appendix: Past, Present, and Future



18.1 System-Level Reconfiguration

A system consists of modular resources (processors, memory banks, disk storage, . . .) and interconnects

Redundant resources can mitigate the effect of module malfunctions

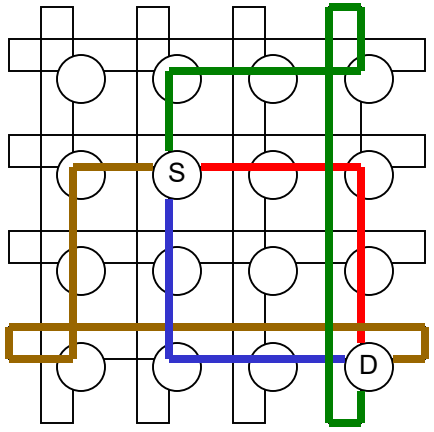
A main challenge in reconfiguration is dealing with interconnects

Assumption: Module/interconnect malfunctions are promptly diagnosed

Overcoming the effect of link malfunctions requires the availability of multiple paths from each source to every possible destination

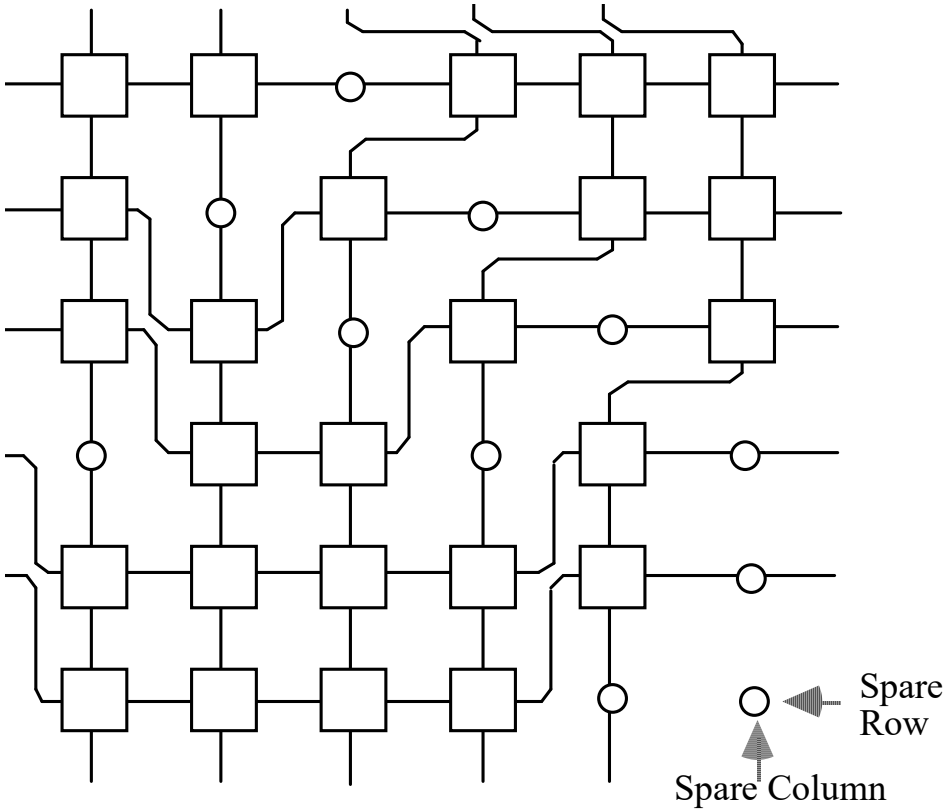
In graph-theoretic terms, we need "edge-disjoint" paths

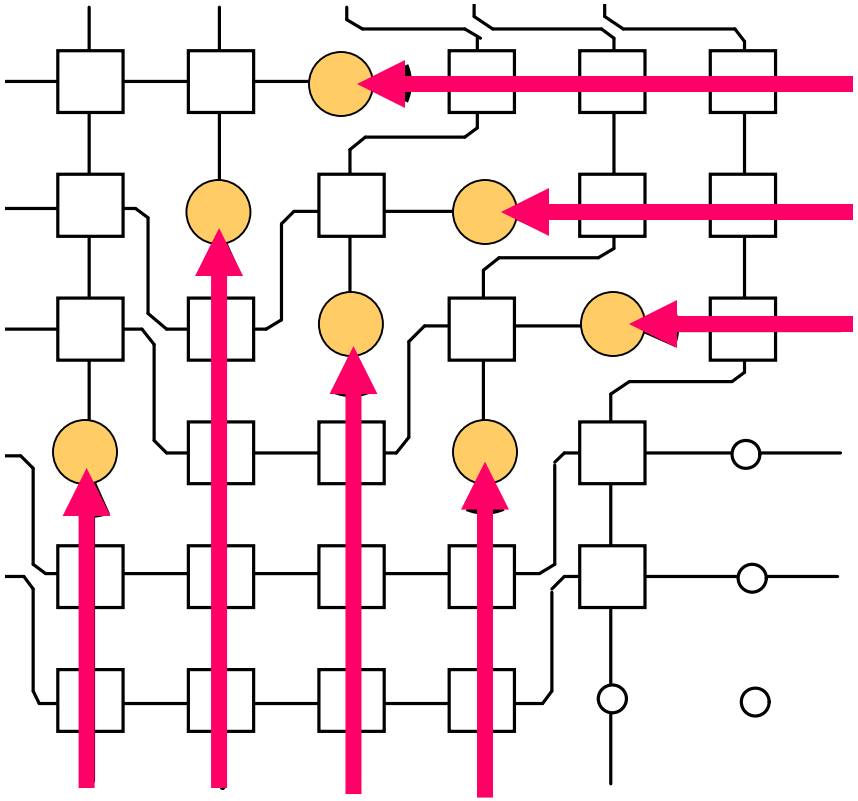
Existence of *k* edge-disjoint paths between two nodes provides the means for tolerating *k* – 1 link malfunctions



This particular interconnection scheme (torus) is 4-connected and tolerates 3 link/node losses without becoming disconnected

Reconfiguration Switching, Revisited

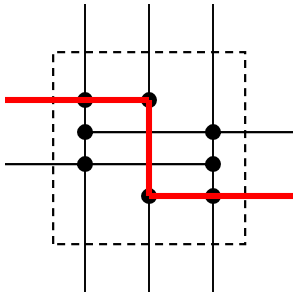




Question: How do we know which cells/nodes must be bypassed?

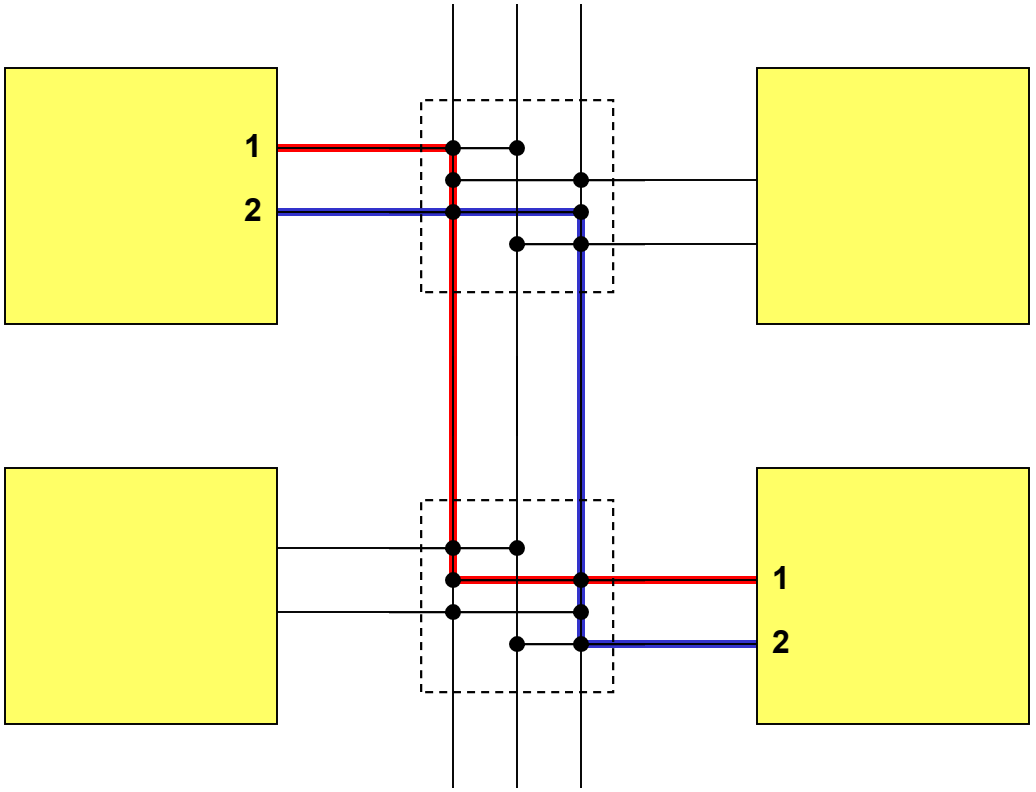
Must devise a scheme in which healthy nodes set the switches

Reconfiguration via Programmable Connections



Interconnection switch with 4 ports (horizontal lines) and 3 channels (vertical lines)

Each port can be connected to 2 of the 3 channels



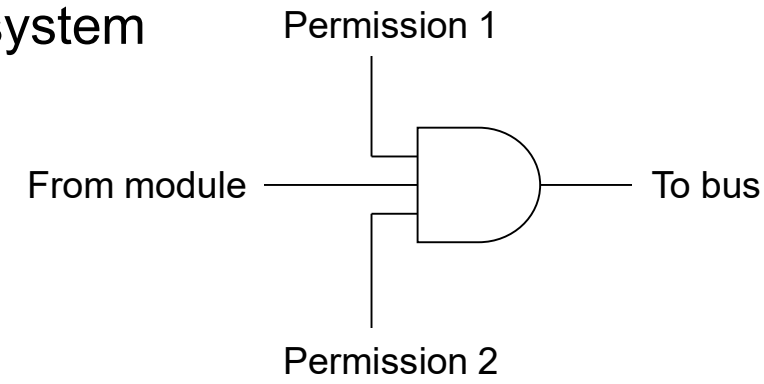
If each module port were connected to every channel, the maximum flexibility would result (leads to complex hardware & control, though)

The challenge lies in using more limited connections effectively

18.2 Isolating a Malfunctioning Unit

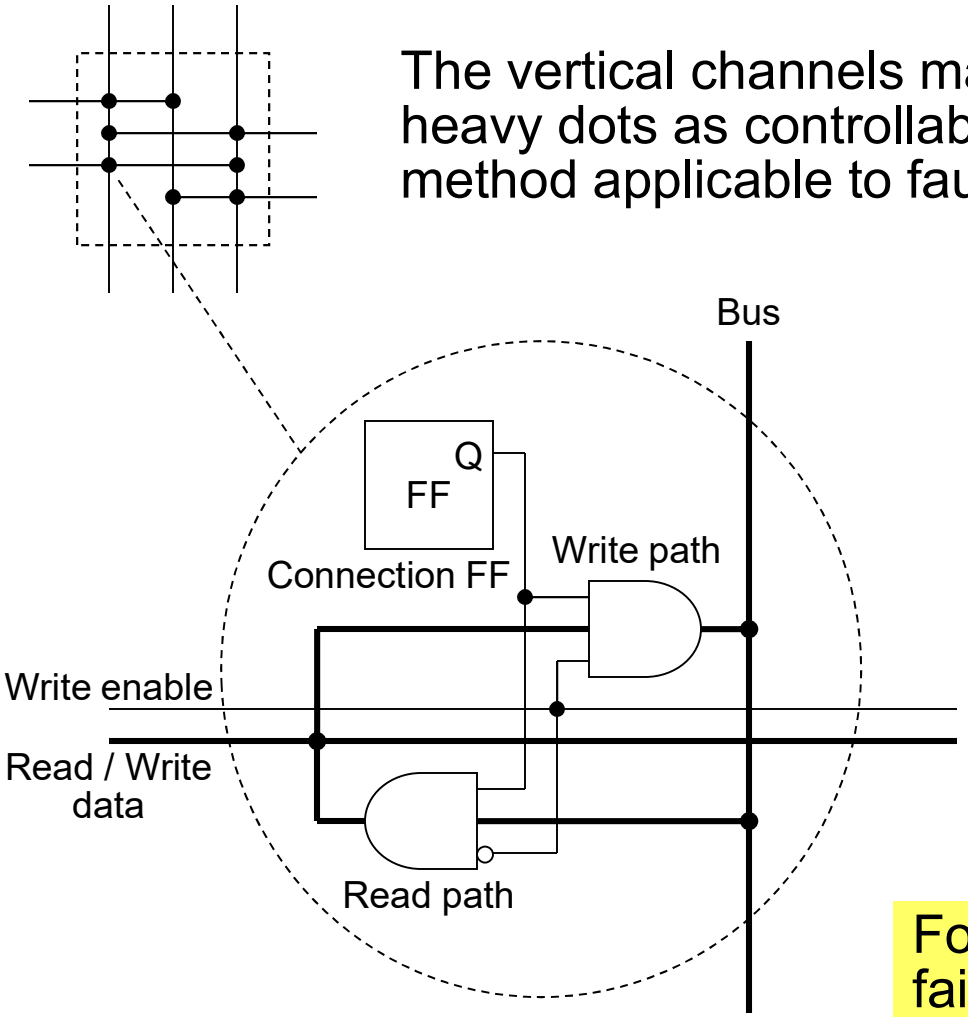
Isolation is needed to prevent malfunctioning units from interfering with the operation of the remaining good units

Notion of “bus guardian” from the SIFT system



Slide to be completed with other examples

Bus-Based Reconfiguration



The vertical channels may be viewed as buses and the heavy dots as controllable bus connections, making this method applicable to fault-tolerant multiprocessors

Failed units can be isolated from the buses

No single bus failure can isolate a module from the rest of the system

If we have extra buses, then faults in the bus connection logic can be tolerated by avoiding the particular bus

For reliability analysis, lump the failure rate of reconfiguration logic with that of its associated bus

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

Log-based recovery is performed via undo/redo:
- Undoing the effects of incomplete transactions
- Redoing transactions whose effects are not reflected in stable storage

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

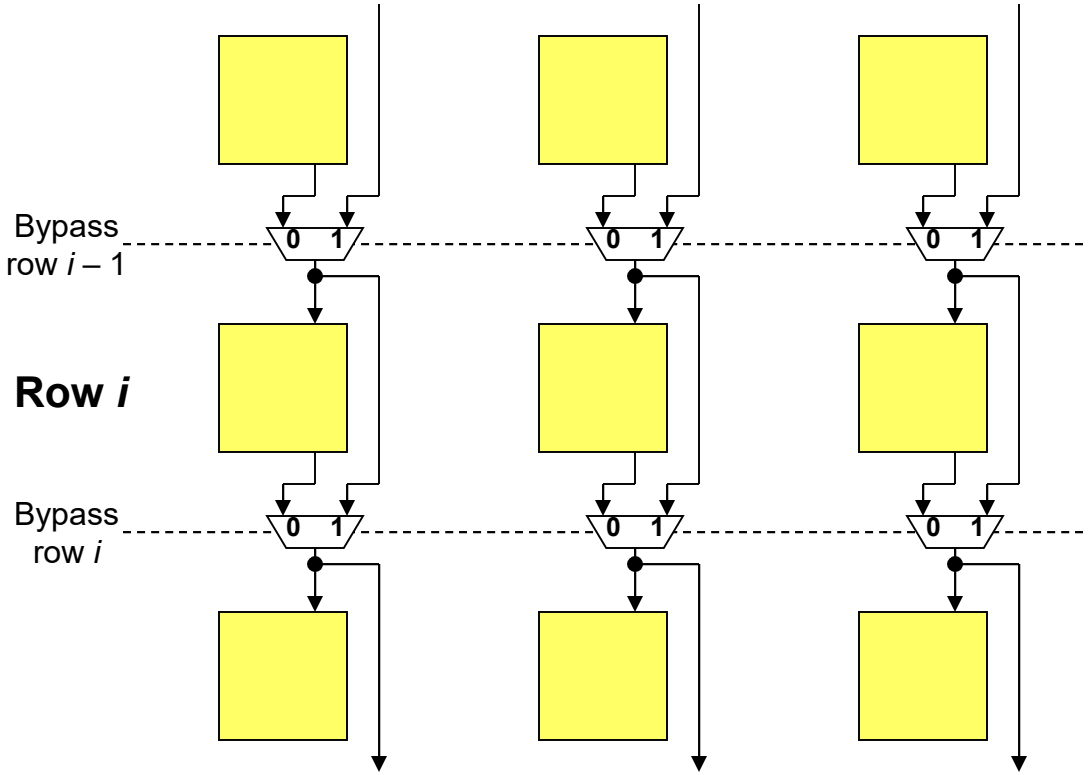
Research is being done at Microsoft and elsewhere to allow querying a database on its state at any desired time instant in the past

18.4 Regular Arrays of Modules

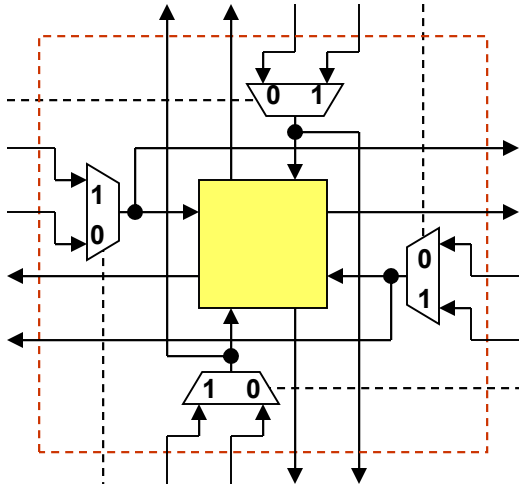
Regularity refers to the interconnection pattern, not physical layout (the latter may be the case for on-chip systems)

Many of the methods of malfunction tolerance in regular arrays are similar to those used for circumventing defects to improve yield

Row/Column Bypassing in 2D Arrays

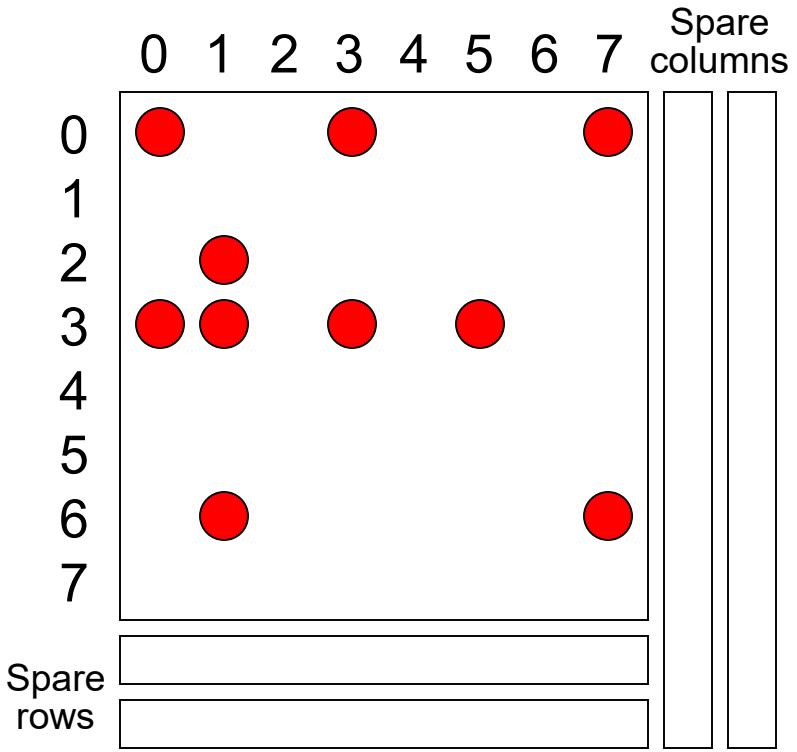


Similar mechanisms needed for northward links in columns and for the eastward and westward links in rows



Question: What types of mechanisms do we need at the edges of this array to allow the row and column edge connections to be directed to the appropriate (nonbypassed) rows and columns?

Choosing the Rows/Columns to Bypass

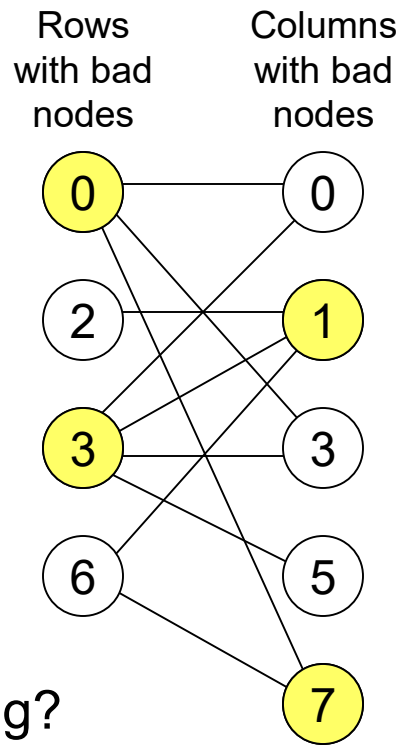


In the adjacent diagram, can we choose up to 2 rows and 2 columns so that they contain all the bad nodes?

Convert to graph problem (Kuo-Fuchs):

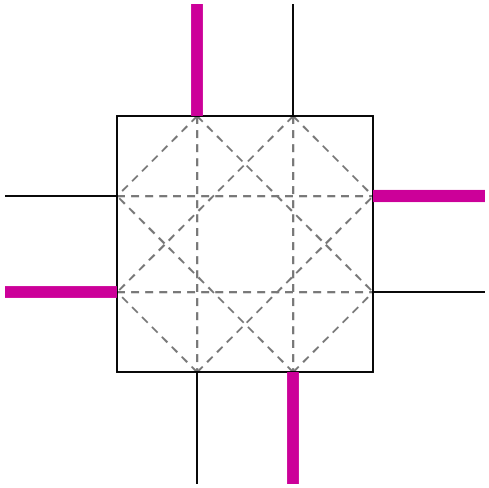
Form bipartite graph, with nodes corresponding to bad rows and columns

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?

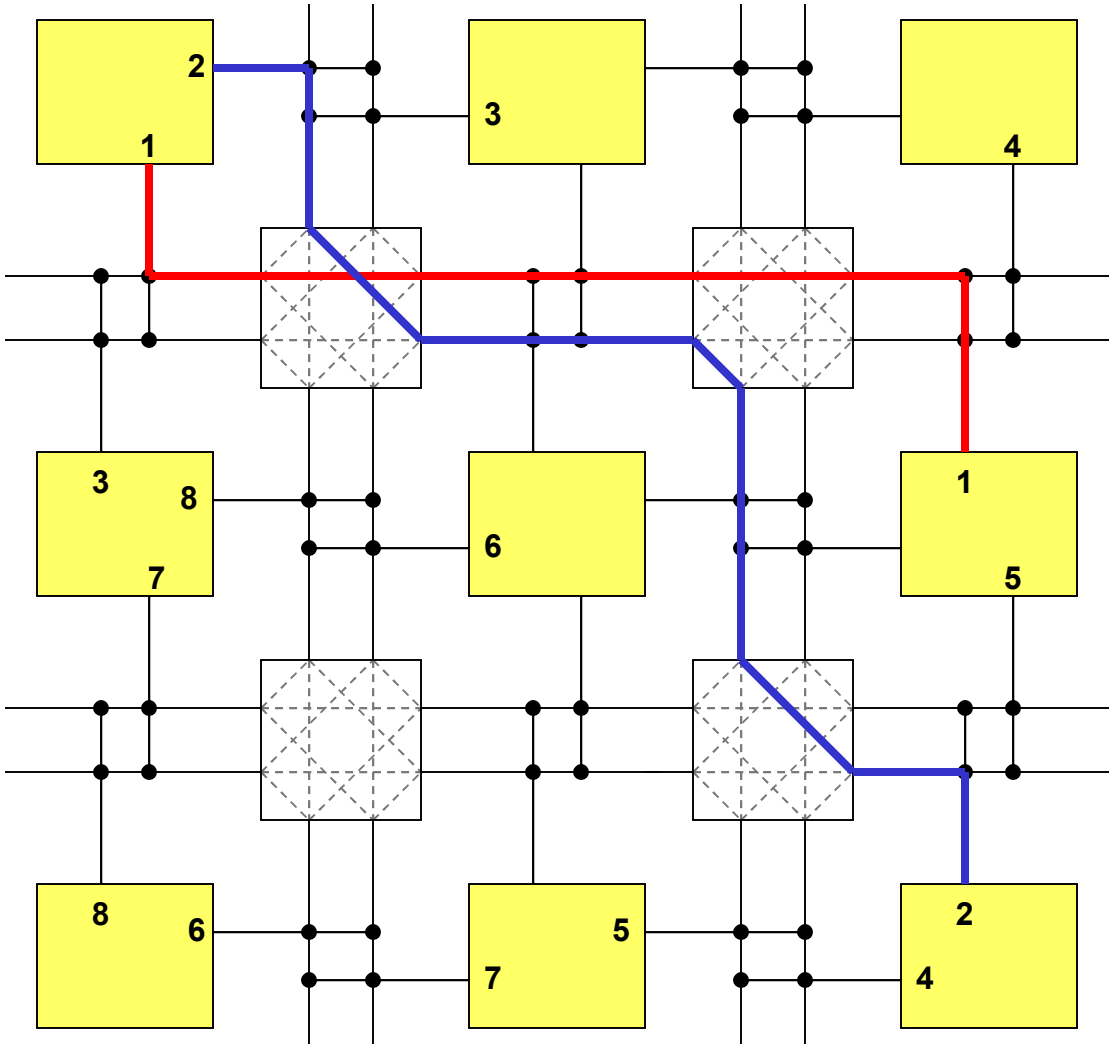
Switch Modules in FPGAs



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

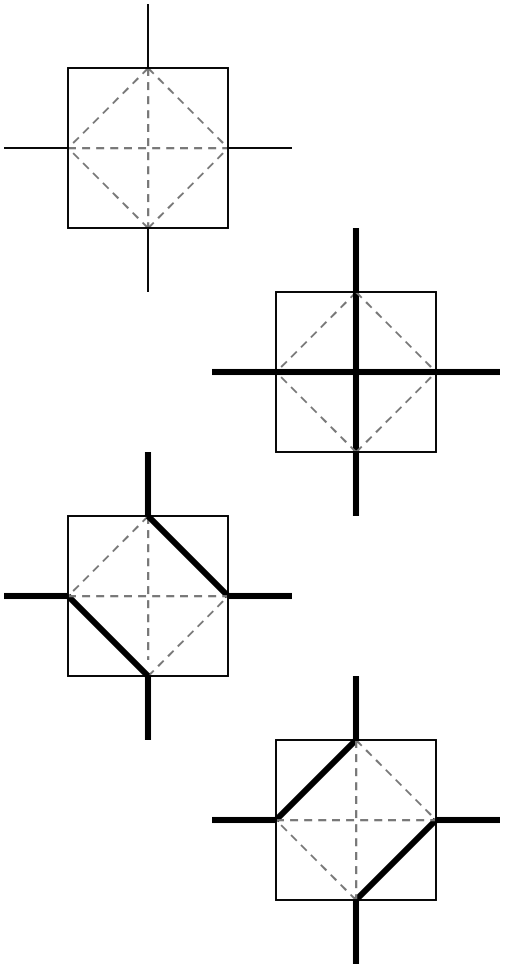
0 – No connection
1 – Straight through
2 – Right turn
3 – Left turn

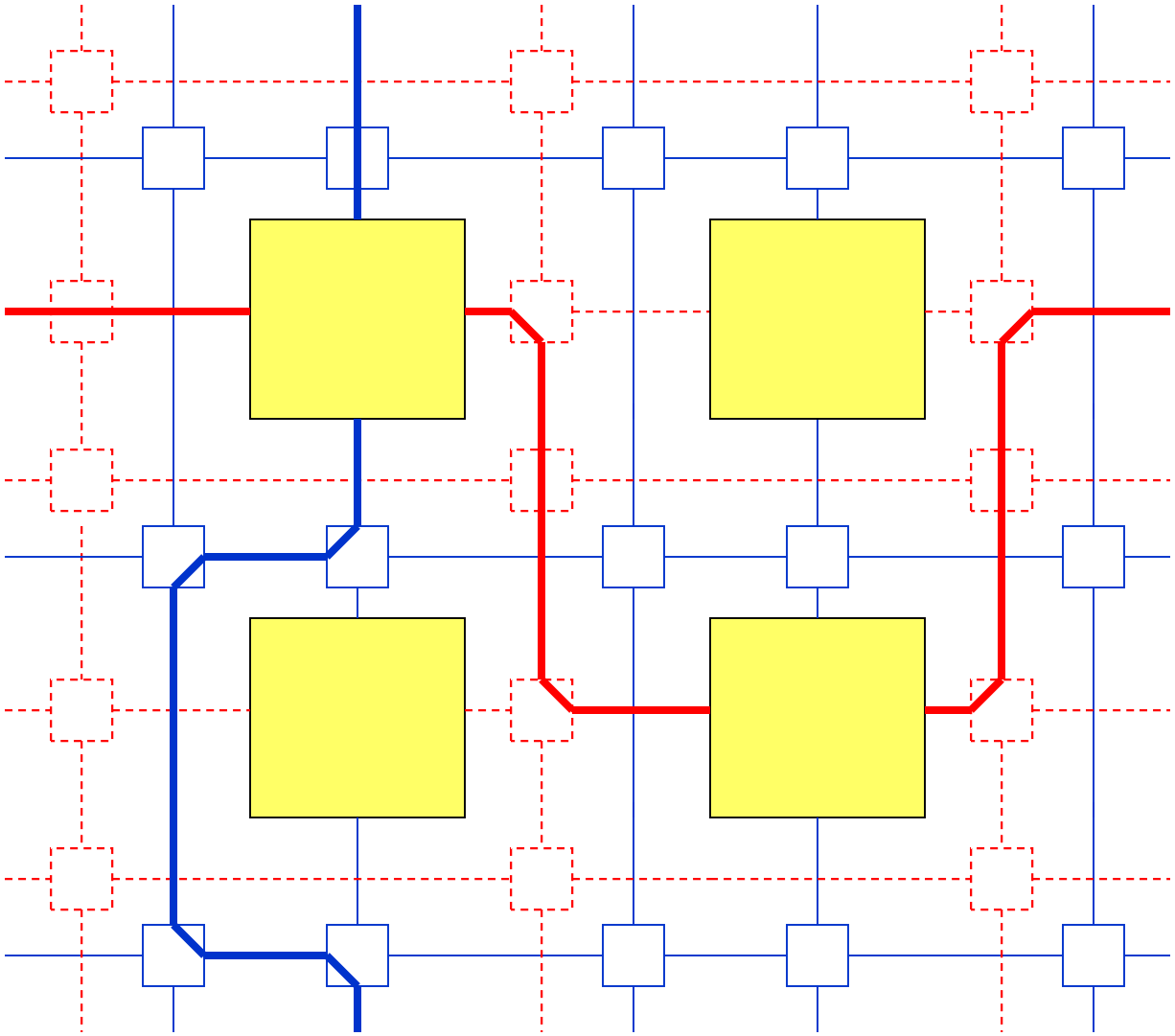
8 control bits (why?)



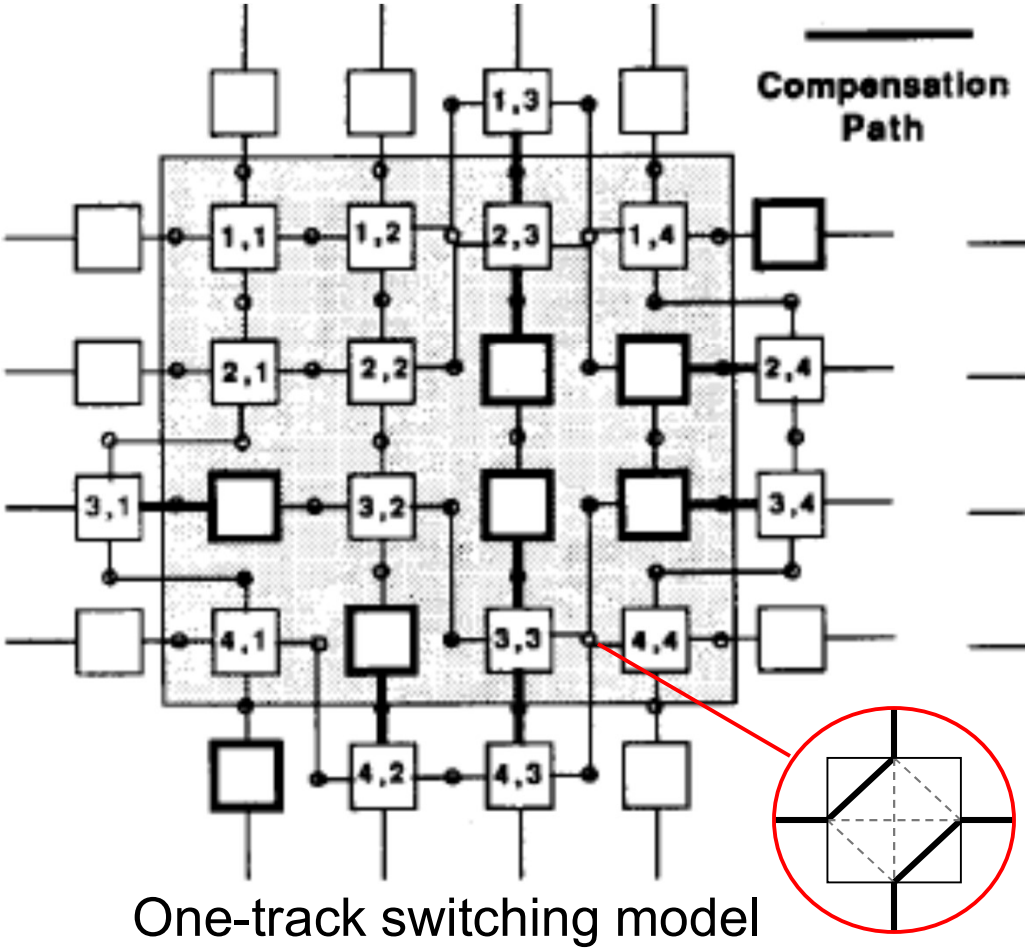
An Array Reconfiguration Scheme

Three-state switch

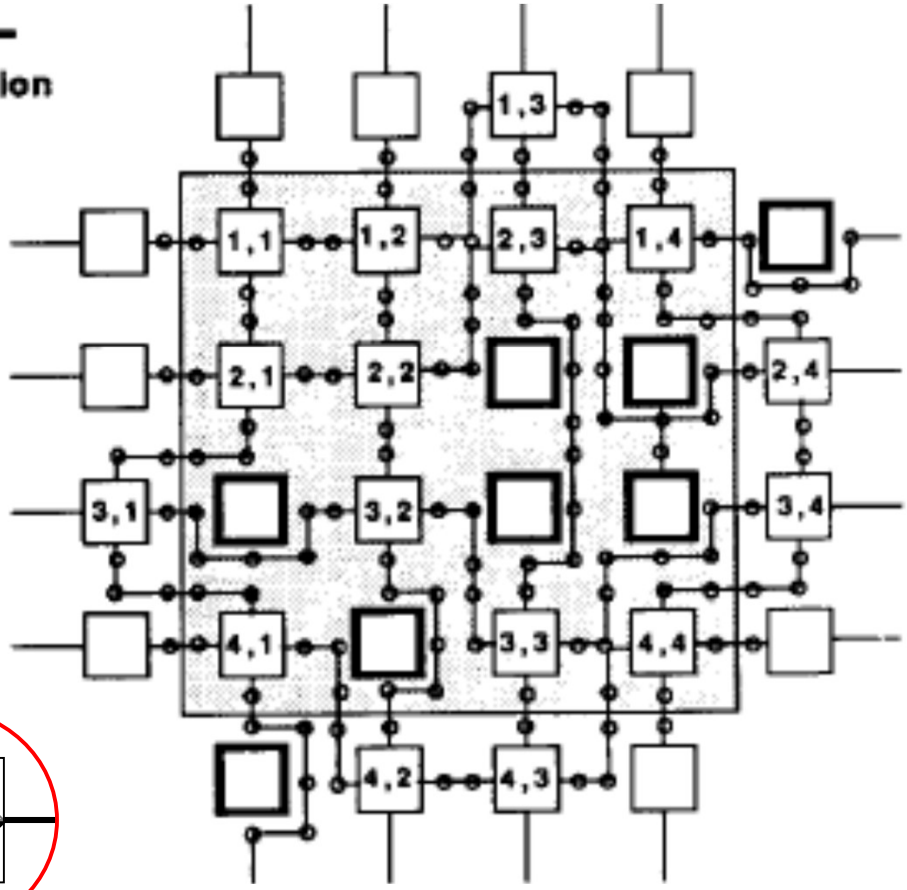




One-Track and Two-Track Switching Schemes



One-track switching model



Two-track switching model

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

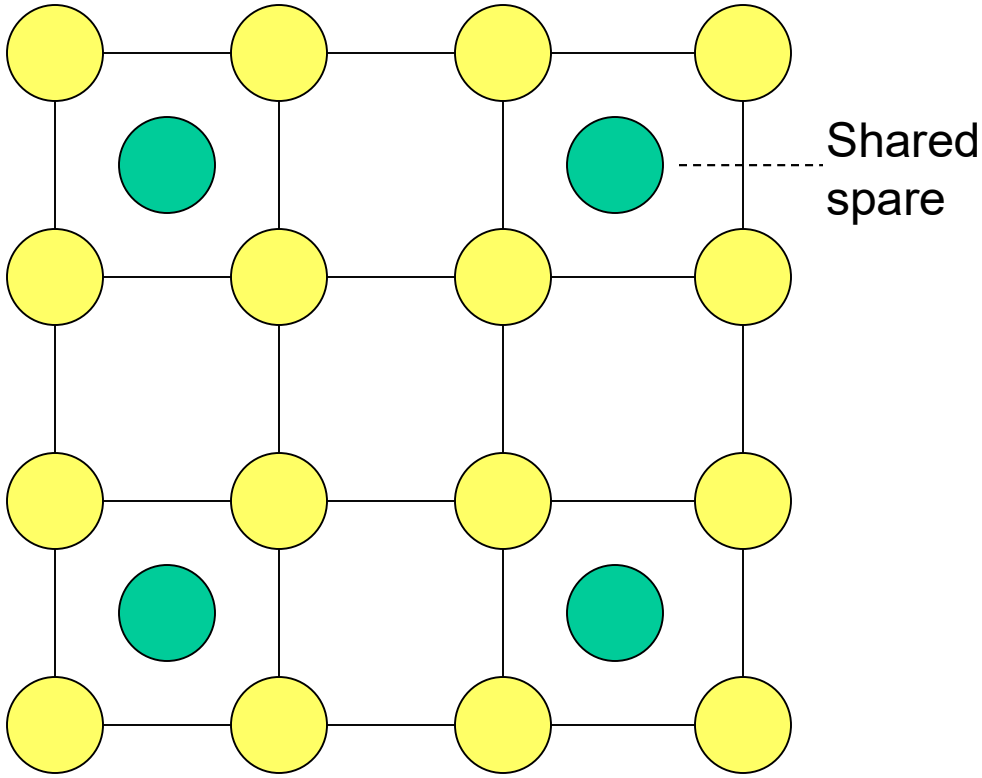
18.5 Low-Redundancy Sparing

Sharing of spares among clusters of modules reduces the hardware overhead, but has two drawbacks:

a. More complex switching

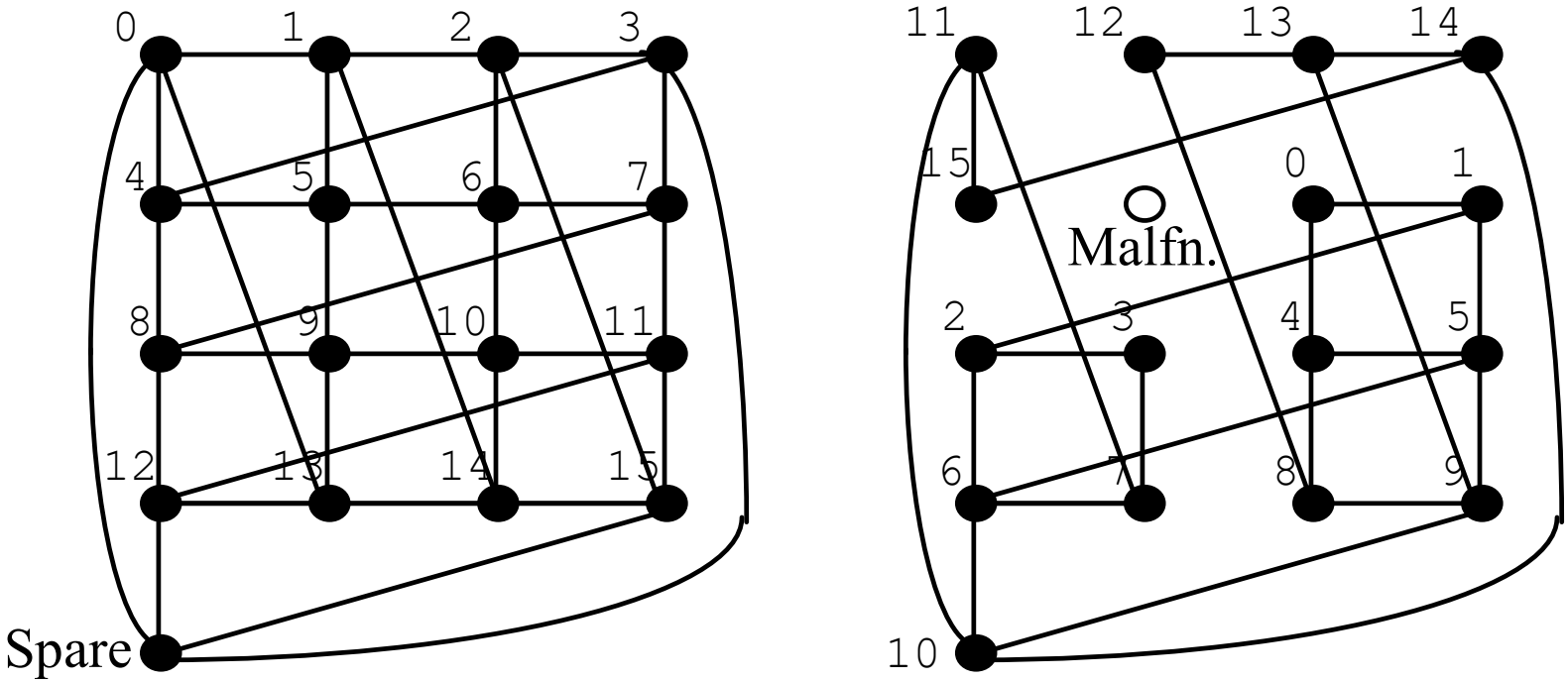
b. Nonuniformity, as spares will need more ports than the primary modules

In this example, a spare module needs 8 ports, given that the 4 modules it can replace collectively have 8 neighbors



Mesh with a Single Spare

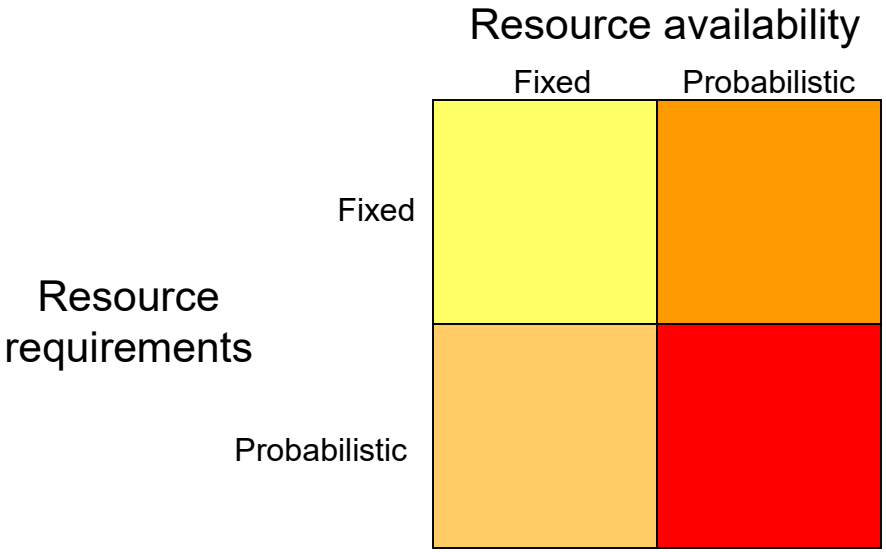
The following example scheme uses only one spare processor for a 2D mesh (no increase in node degree), yet it allows system reconfiguration to circumvent any malfunctioning processor, replacing it with the spare via relabeling of the nodes



Reconfigurable 4 × 4 mesh with one spare

18.6 Malfunction-Tolerant Scheduling

Scheduling problems are hard even when resource requirements and availability are both fixed and known a priori

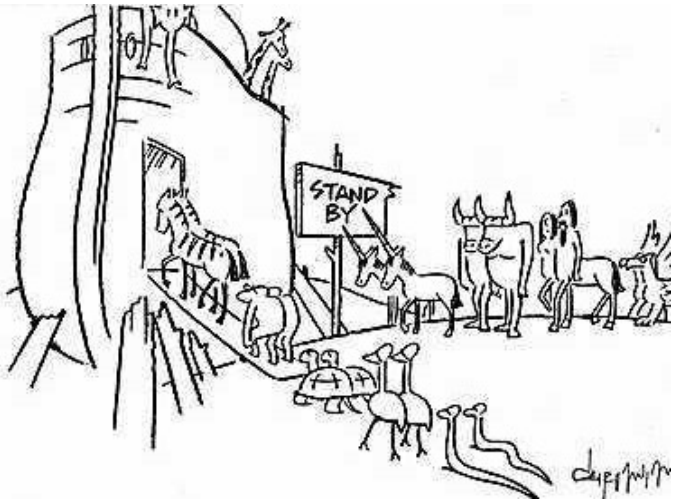


When resource availability is fixed, the quality of a schedule is judged by:
(1) Completion times (2) Meeting of deadlines

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

19 Standby Redundancy

TO DEAL WITH OVER-CROWDING IN OUR CLASSES, WE'VE PUT THE STUDENTS ON "STAND-BY"



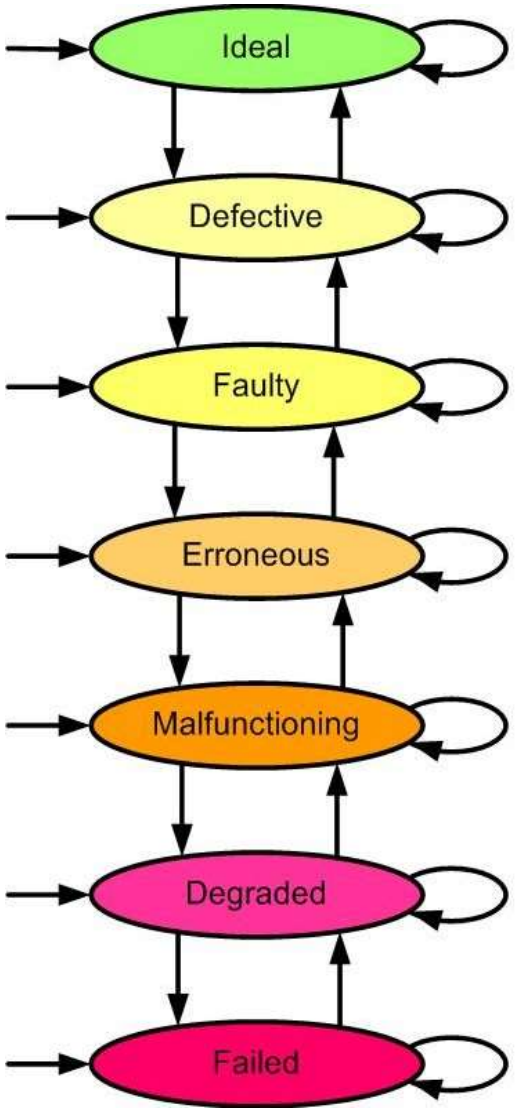
"We back up our data on sticky notes because sticky notes never crash."

"DON'T WORRY-HE WAS GOING TO BE MADE REDUNDANT ANYWAY."

STRUCTURE AT A GLANCE

Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals / Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods / Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods / Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods / Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods / Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Robust Parallel Processing
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods / Examples	21. Degradation Allowance 22. Degradation Management 23. Resilient Algorithms 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods / Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

Appendix: Past, Present, and Future



19.1 Malfunction Detection

No amount of spare resources is useful if the malfunctioning of the active module is not detected quickly

Detection options

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

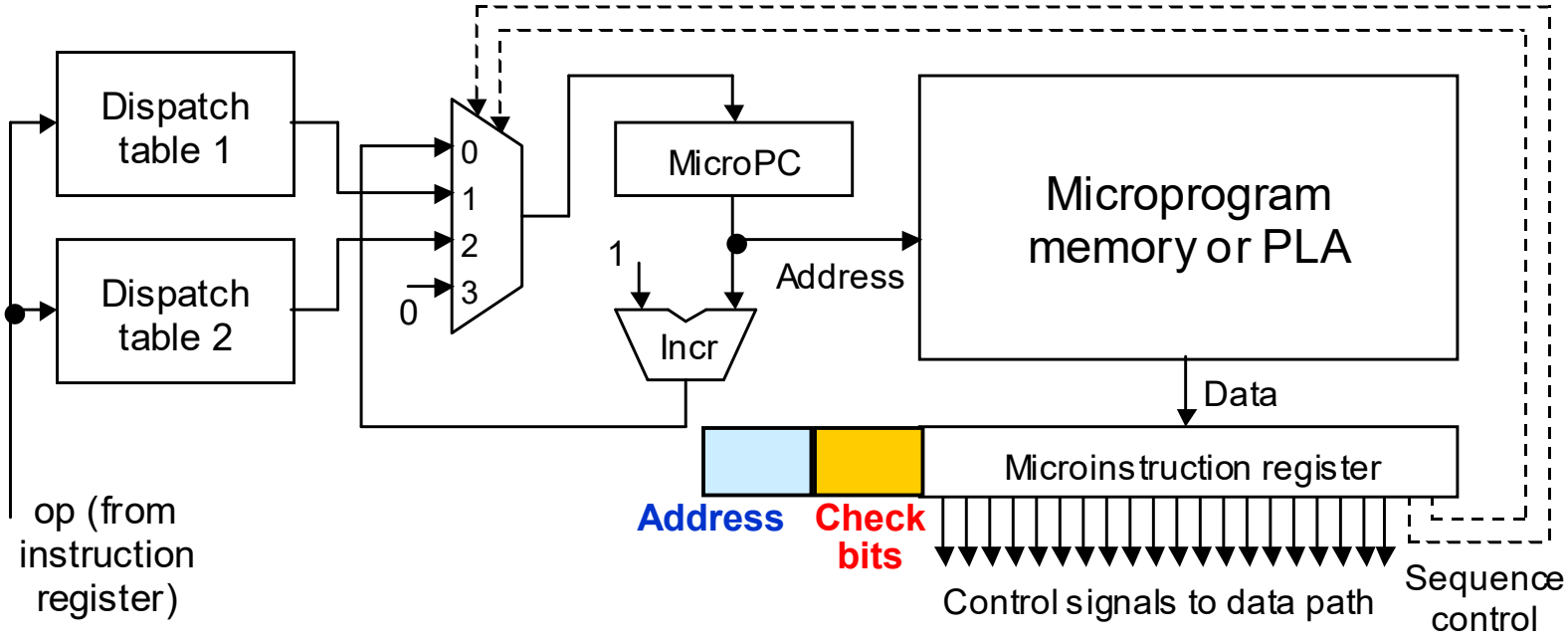
Coding: Particularly suitable for memory and storage modules

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

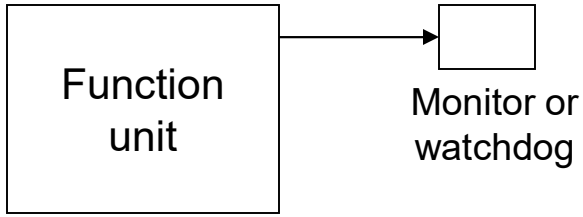
In a microprogrammed control unit, store the microinstruction address and compare against MicroPC contents to detect sequencing errors



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



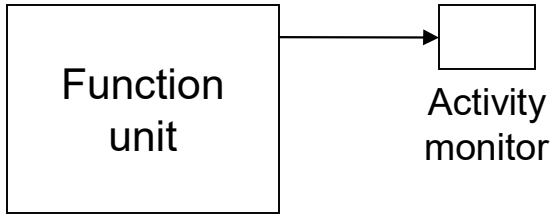
Watchdog timer counts down, beginning from a preset number
It expects to be preset periodically by the unit that it monitors
If the count reaches 0, the watchdog timer raises an exception flag

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)

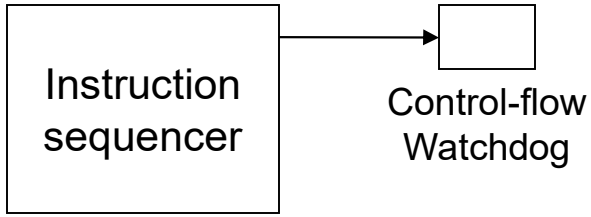
The type of monitoring is highly application-dependent

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

19.2 Cold and Hot Spare Units

Cold spare: Inactive, perhaps even powered down

Hot spare: Active, ready to take over in short order

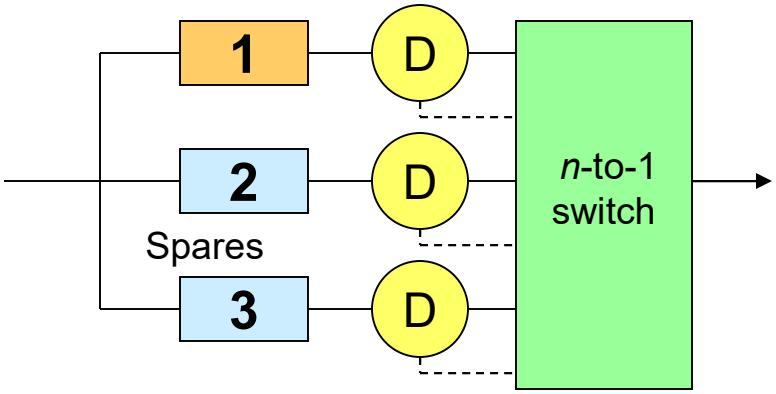
Warm spare: Intermediate between hot and cold (perhaps powered up, but not up to date with respect to the state of the active module)

19.3 Conditioning of Spares

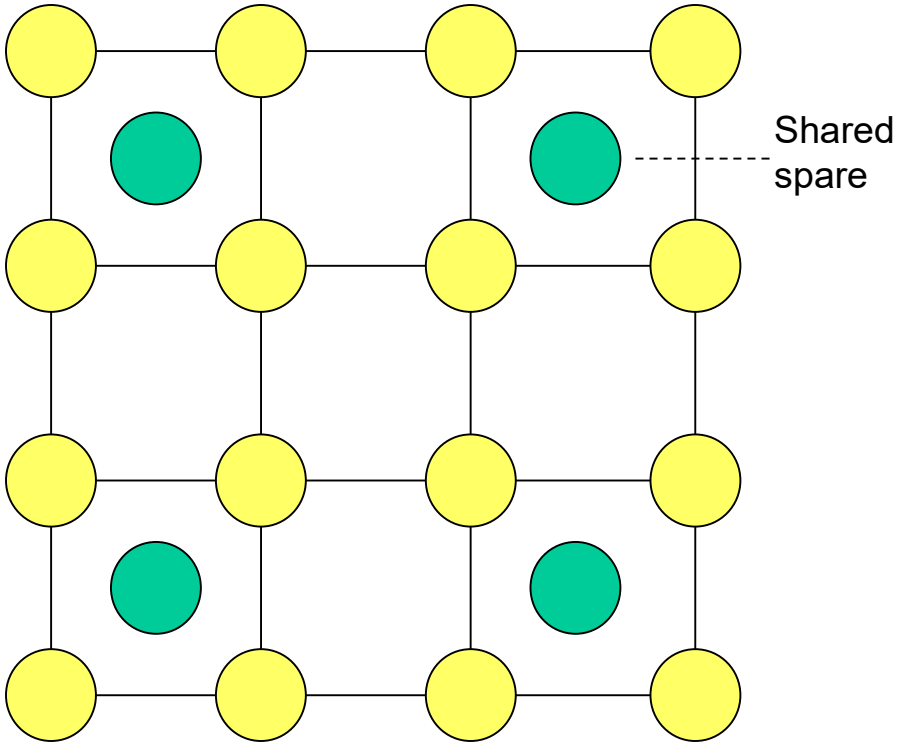
Conditioning refers to preparing a spare module to take the place of an active module

19.4 Switching over to Spares

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



Single-unit replacement



Replacement in ensembles of units

19.5 Self-Repairing Systems

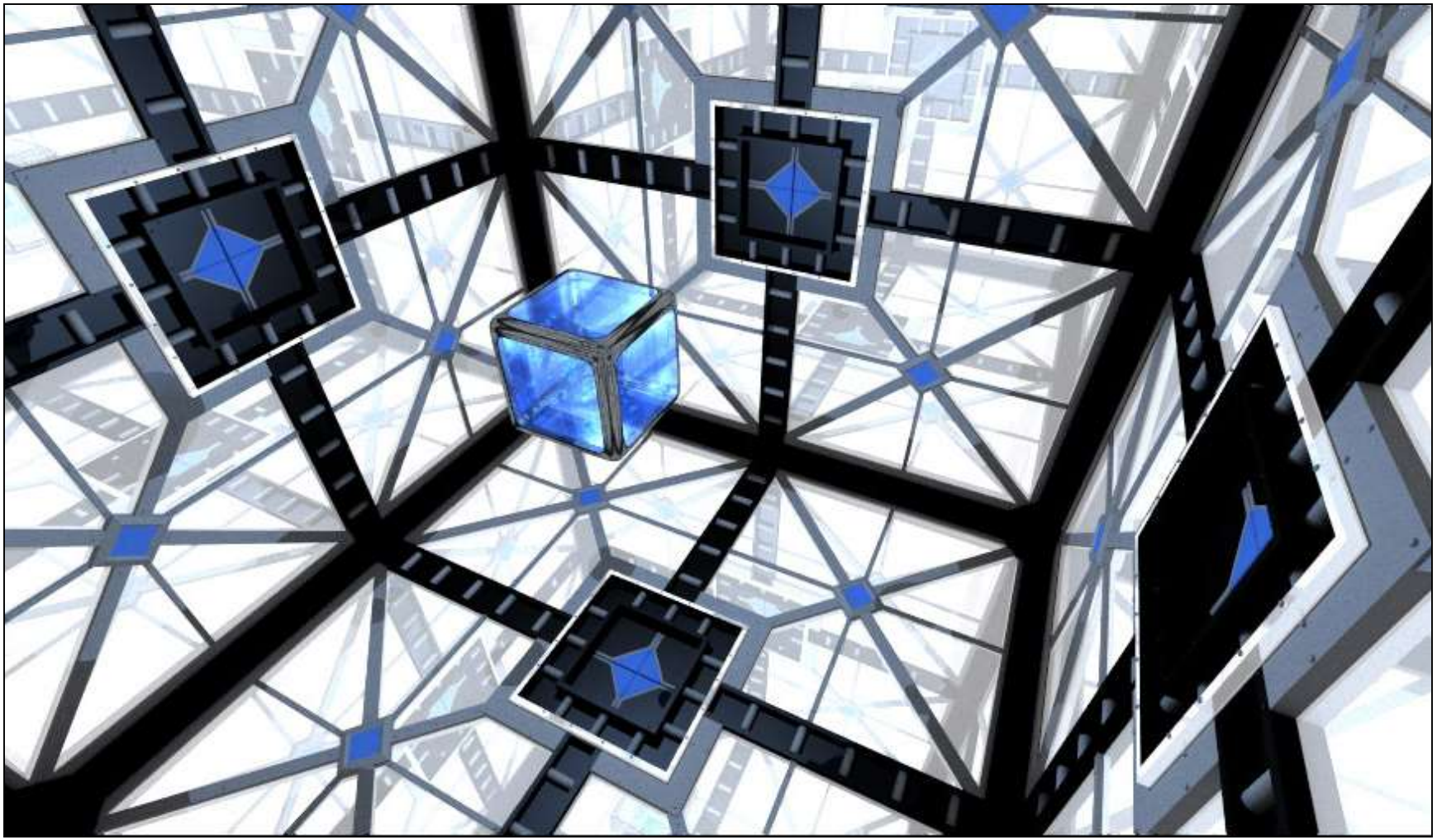
Self-repair is the ability of a system to go from one working configuration to another (after a detected malfunction), without human intervention

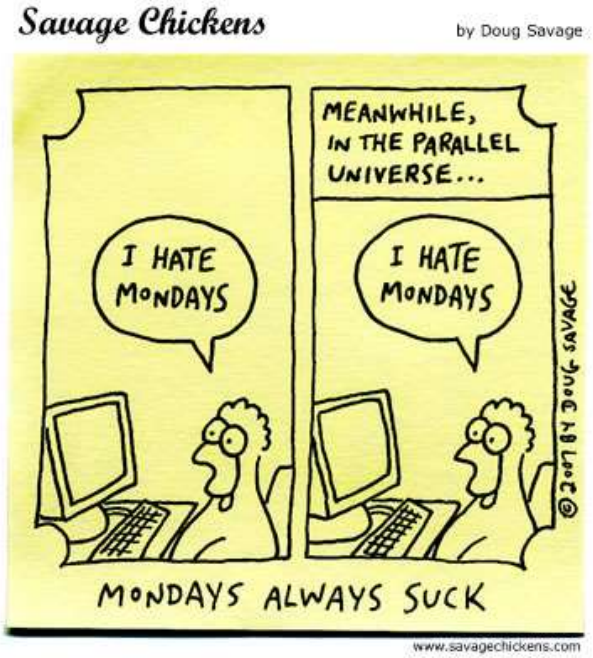
Autonomic systems, autonomic computing: Self-management in the face of changes in resources and application requirements

19.6 Modeling of Self-Repair

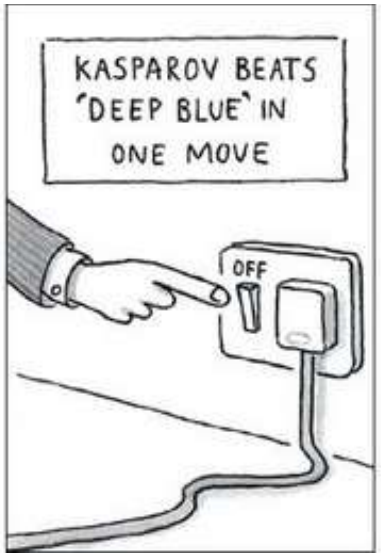
Both combinational and state-space models to be discussed

20 Robust Parallel Processing





Performance tuning technique number 39: Reading in parallel

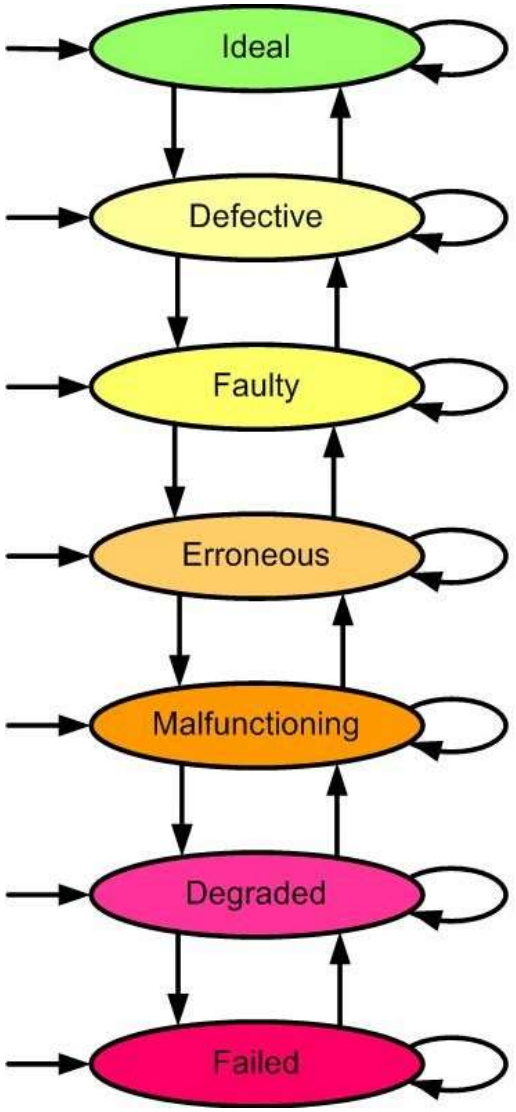


"Okay. Now, try to parallel park while talking on your cell phone and changing songs on your iPod."

STRUCTURE AT A GLANCE

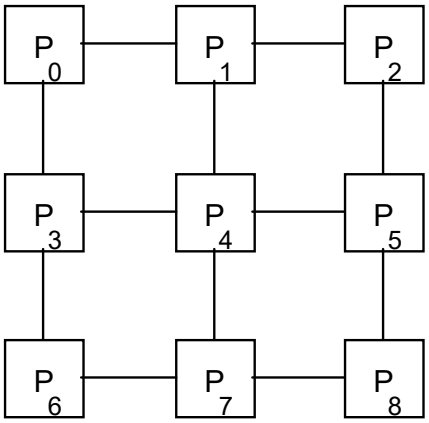
Part I — Introduction: Dependable Systems (The Ideal-System View)	Goals / Models	1. Background and Motivation 2. Dependability Attributes 3. Combinational Modeling 4. State-Space Modeling
Part II — Defects: Physical Imperfections (The Device-Level View)	Methods / Examples	5. Defect Avoidance 6. Defect Circumvention 7. Shielding and Hardening 8. Yield Enhancement
Part III — Faults: Logical Deviations (The Circuit-Level View)	Methods / Examples	9. Fault Testing 10. Fault Masking 11. Design for Testability 12. Replication and Voting
Part IV — Errors: Informational Distortions (The State-Level View)	Methods / Examples	13. Error Detection 14. Error Correction 15. Self-Checking Modules 16. Redundant Disk Arrays
Part V — Malfunctions: Architectural Anomalies (The Structure-Level View)	Methods / Examples	17. Malfunction Diagnosis 18. Malfunction Tolerance 19. Standby Redundancy 20. Robust Parallel Processing
Part VI — Degradations: Behavioral Lapses (The Service-Level View)	Methods / Examples	21. Degradation Allowance 22. Degradation Management 23. Resilient Algorithms 24. Software Redundancy
Part VII — Failures: Computational Breaches (The Result-Level View)	Methods / Examples	25. Failure Confinement 26. Failure Recovery 27. Agreement and Adjudication 28. Fail-Safe System Design

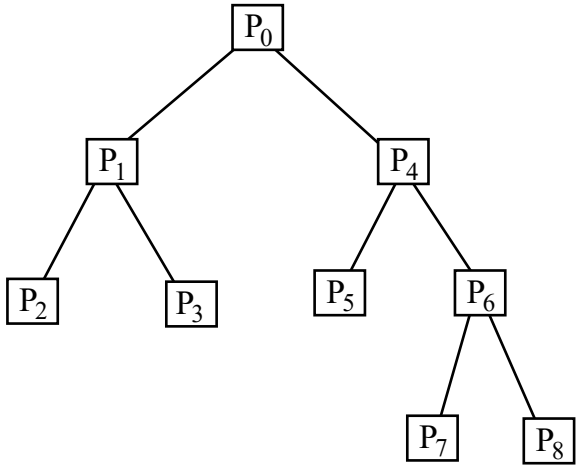
Appendix: Past, Present, and Future

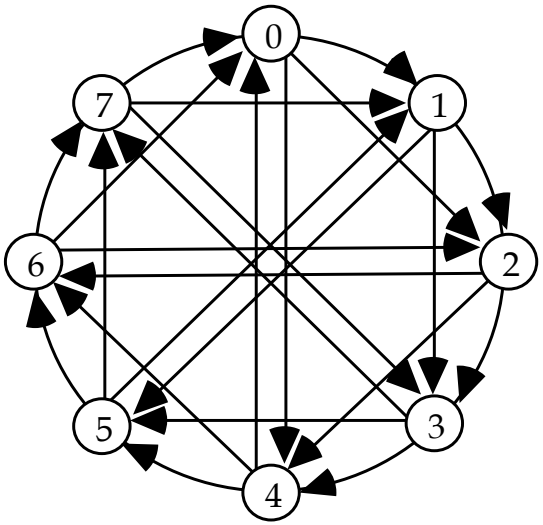


20.1 A Graph-Theoretic Framework

In robust parallel processing, we don't make a distinction between ordinary resources and spare resources







Parallel processors have redundancy built in, because they possess:

— Multiple processing resources

— Multiple alternate paths for message transmission between nodes

* Many of the ideas and figures in this chapter are from the author's textbook on parallel processing

Interprocessor Communication Architectures

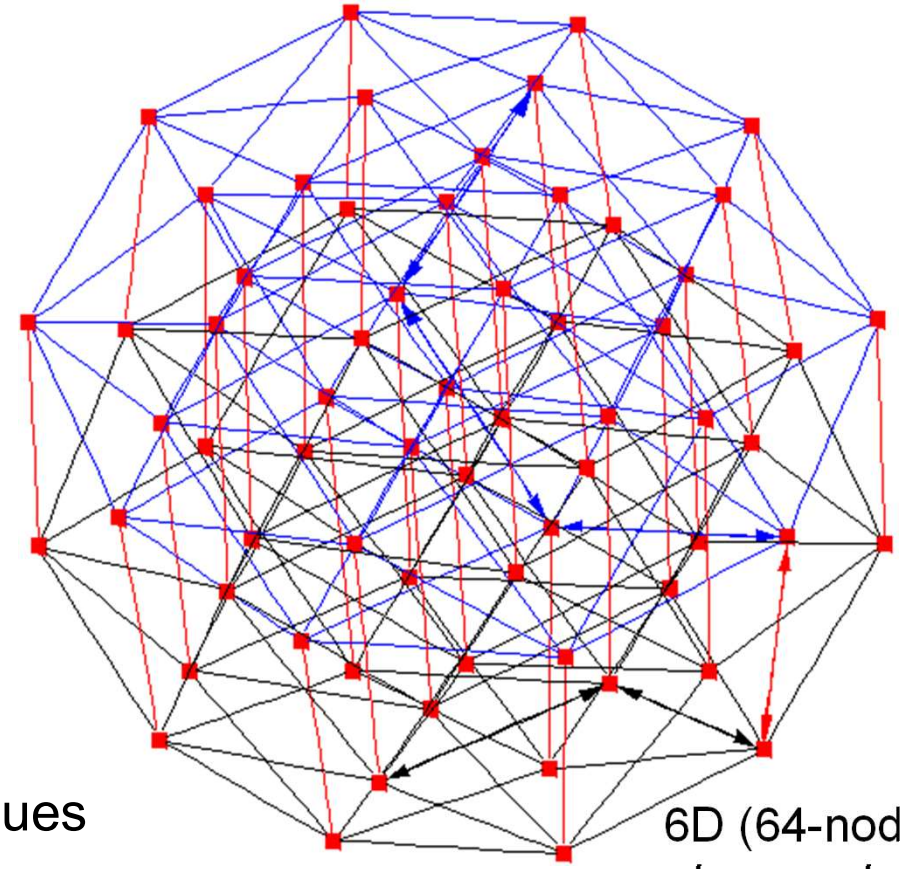
Consider systems built from (homo/heterogeneous) processing nodes
Such a parallel processing system can be modeled by a (di)graph

Communication architecture is Characterized by:

— Type of routing (packet switching vs. wormhole or cut-through)

— Protocols supported (e.g., whether nodes have buffer storage for messages)

These details don't matter at the level of graph representation, which models only connectivity issues

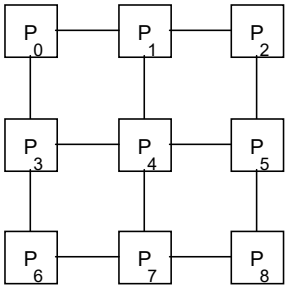


6D (64-node) hypercube

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, Δ: 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

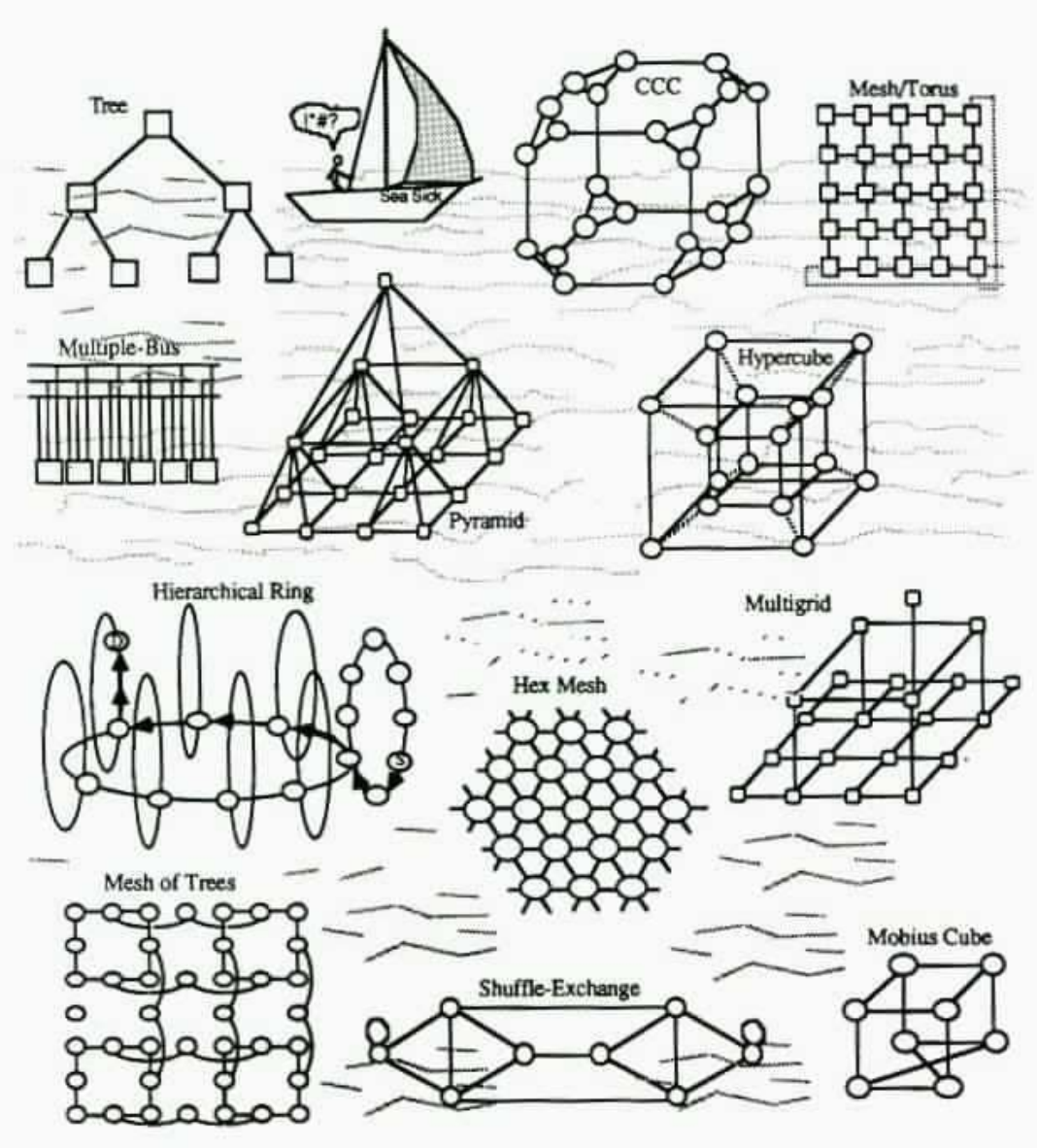
Edge symmetry: edges are interchangeable via relabeling

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

The Sea of Direct Interconnection Networks

A wide variety of direct interconnection networks have been proposed for, or used in, parallel computers

They differ in topological, performance, robustness, and realizability attributes.



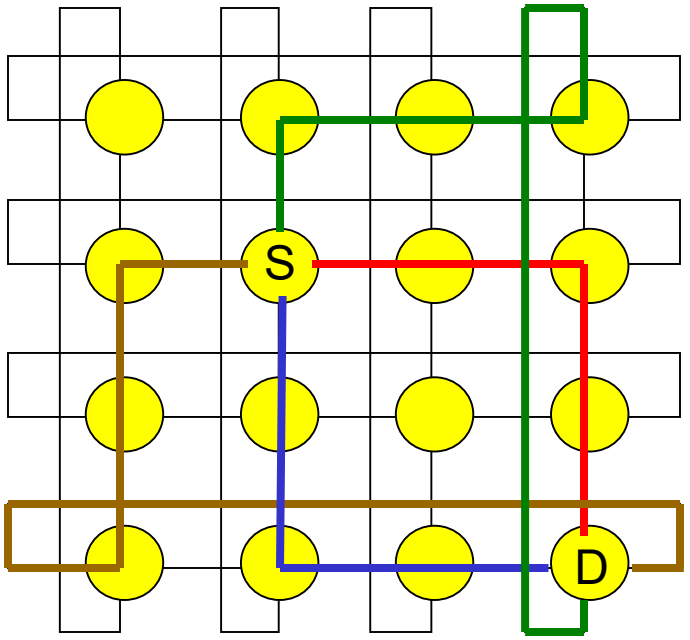
Robustness Attributes of Networks

Connectivity κ: 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

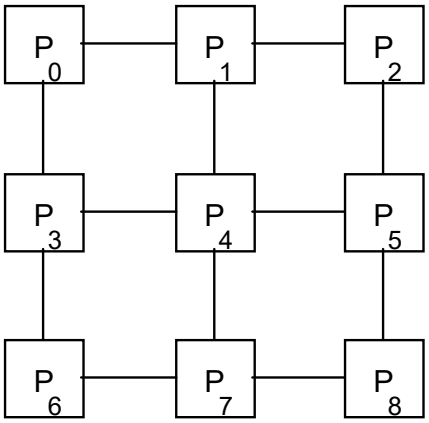


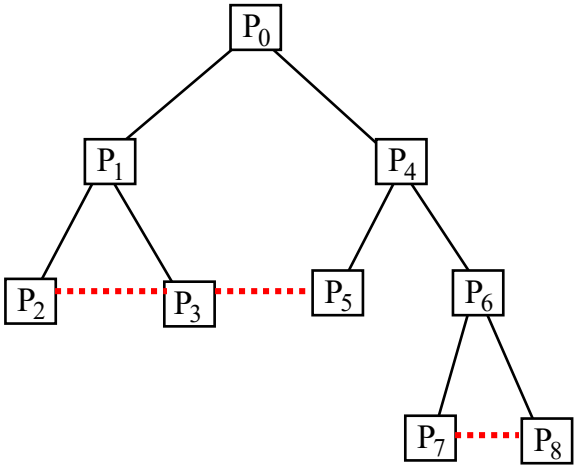
In this discussion, we are effectively merging ordinary system resources with spares (no node or link is specifically designated as spare)

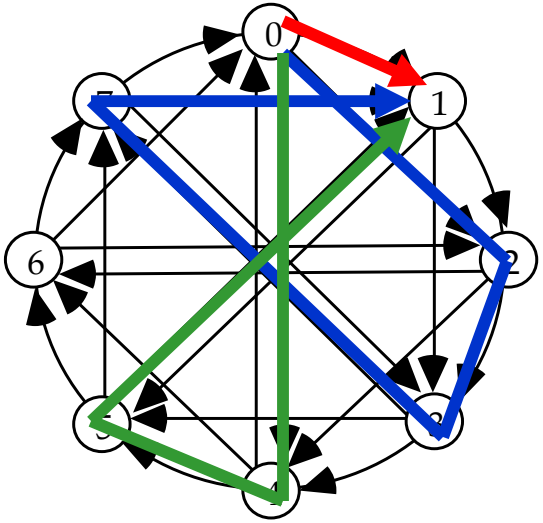
All units are simultaneously active and contribute to system performance, which, under no malfunction, is greater than the needed amount

20.2 Connectivity and Parallel Paths

Connectivity $\kappa \leq$ minimum node degree d_{min}
If equality holds, the network is optimally/maximally malfunction-tolerant







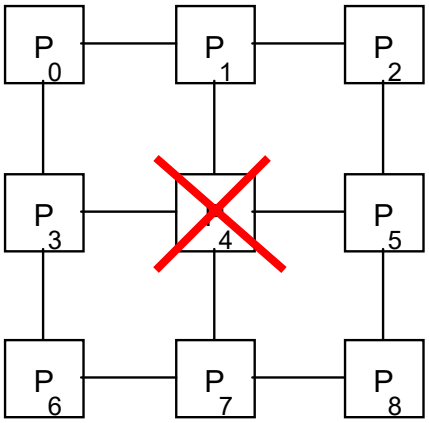
Symmetric networks tend to be maximally malfunction-tolerant

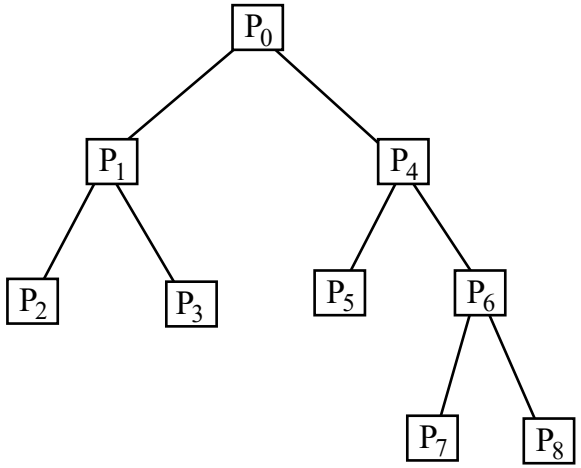
Finding the connectivity of a network not always an easy task

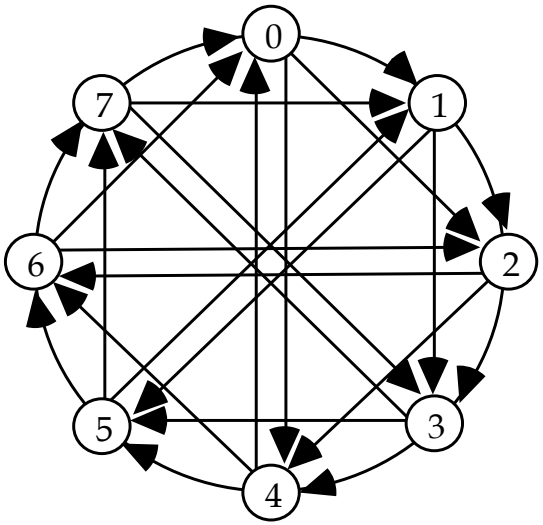
Many papers in the literature on connectivity of various networks

20.3 Dilated Internode Distances

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





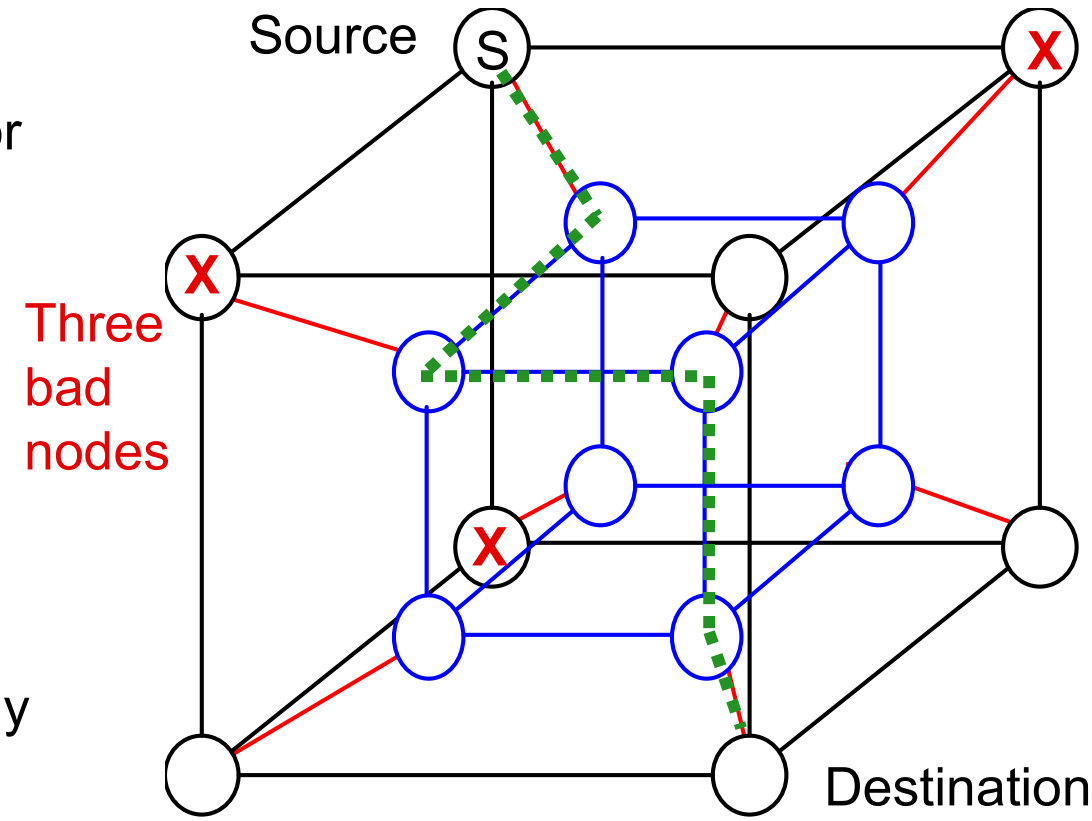


Malfunction diameter: Worst case diameter for $\kappa - 1$ malfunctions

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

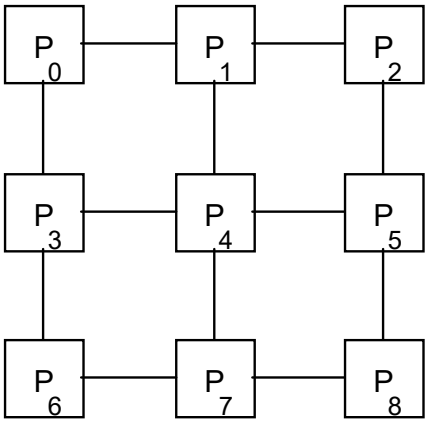


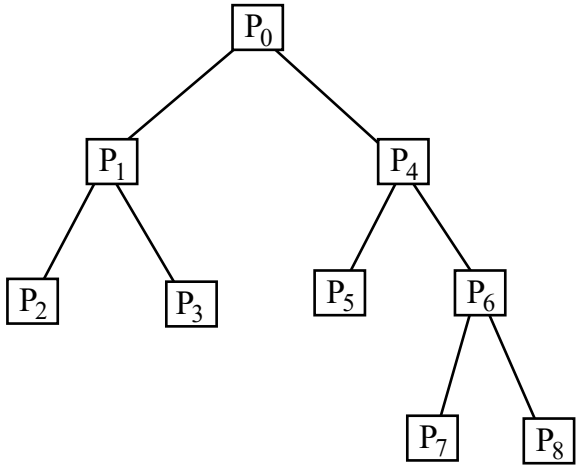
The node that is furthest from S is not its diametrically opposite node in the malfunction-free hypercube

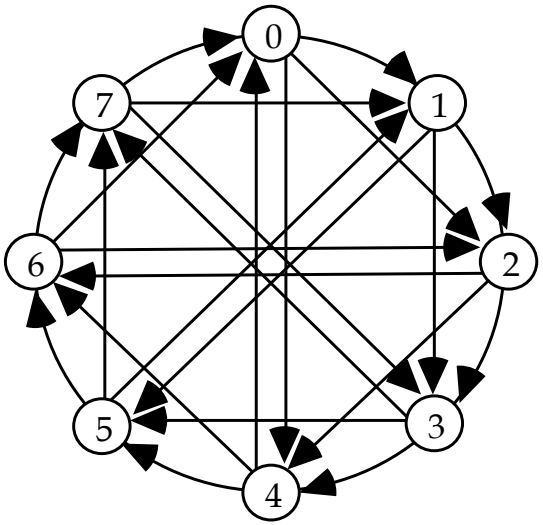
Malfunction diameter of the q-cube is $q + 1$

20.4 Malfunction-Tolerant Routing

1. Malfunctioning units known globally (easier case; precompute path)
2. Only local malfunction info available (distributed routing decisions)







Distributed routing decisions are usually preferable, but they may lead to:

Suboptimal paths: Messages not going through shortest paths possible

Deadlocks: Messages interfering with and circularly waiting for each other

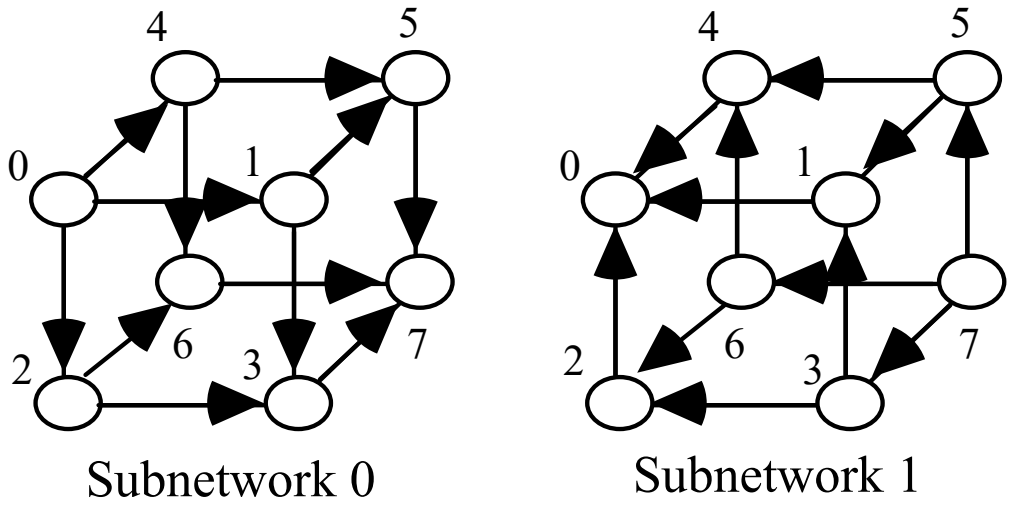
Livelocks: Wandering messages that never reach their destinations

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

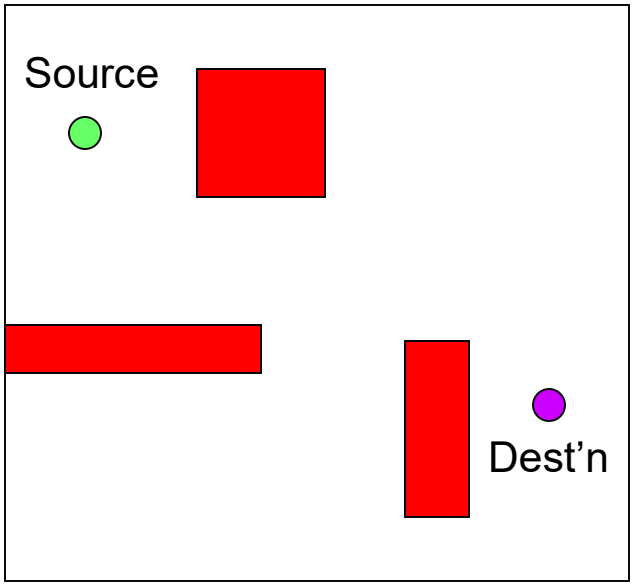
Hence, any routing scheme that begins by using links in subnet 0, at some point switches the path to subnet 1, and from then on remains in subnet 1, is deadlock-free

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

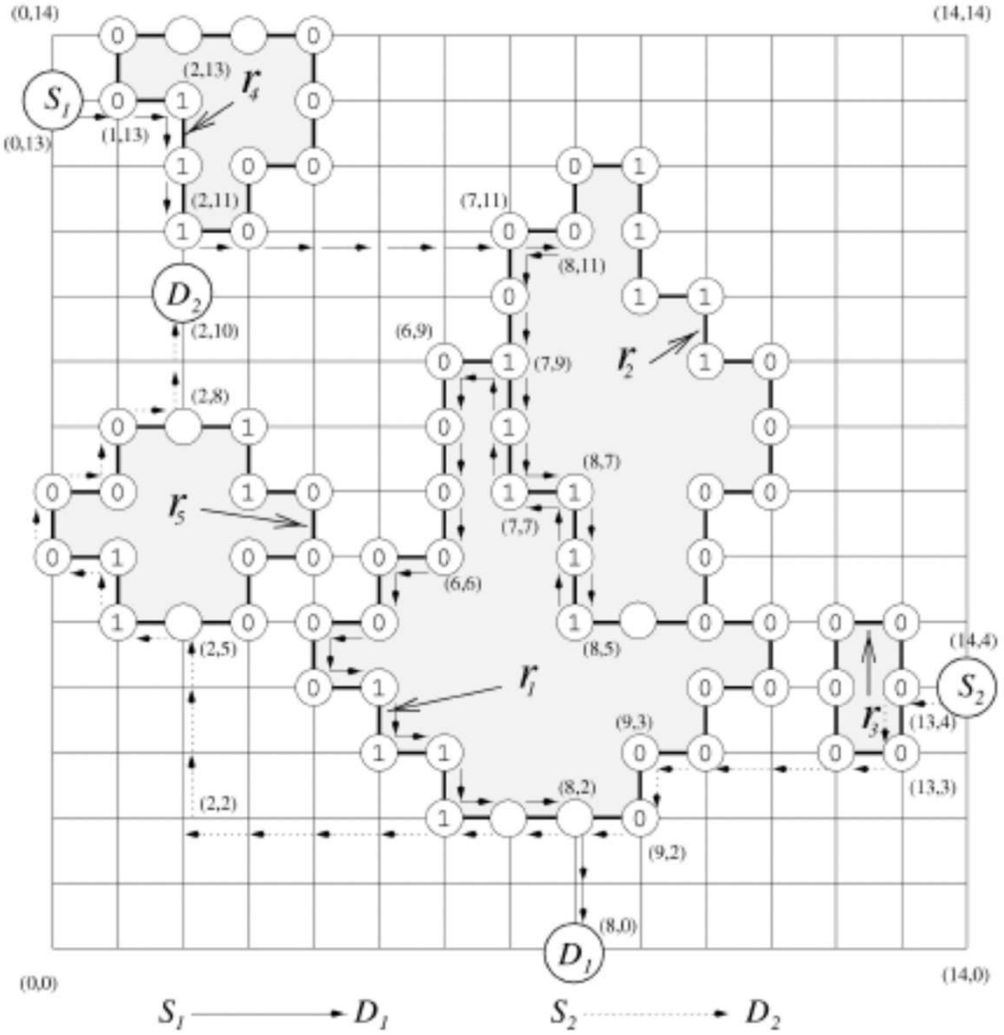
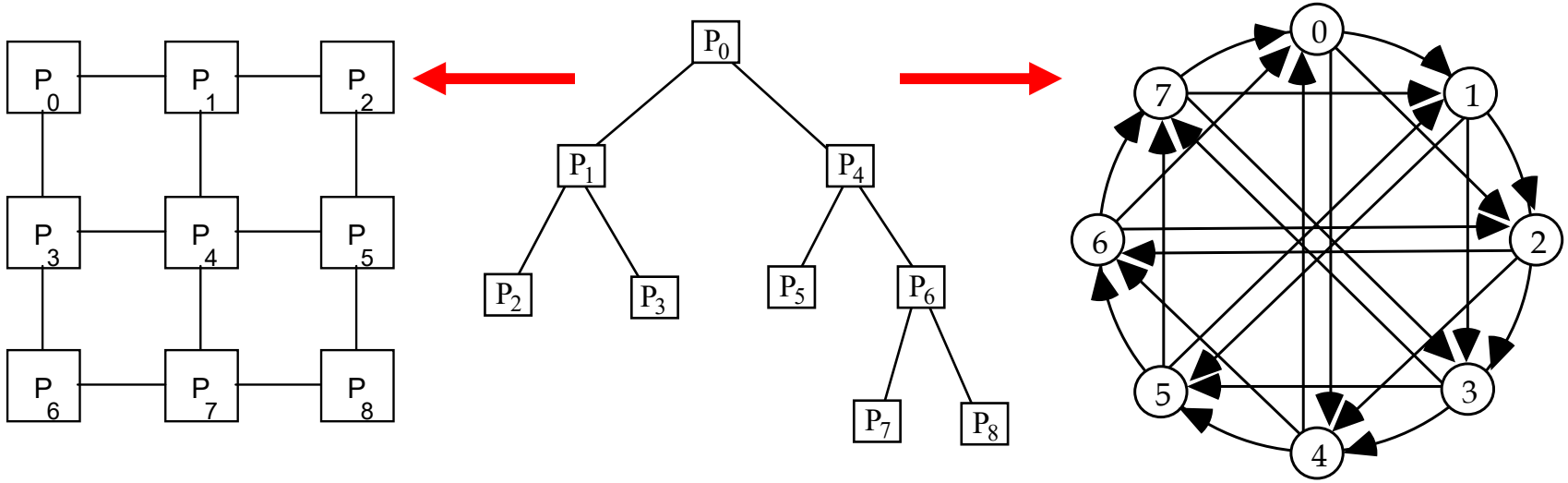


Figure from [Chen01]

20.5 Embeddings and Emulations

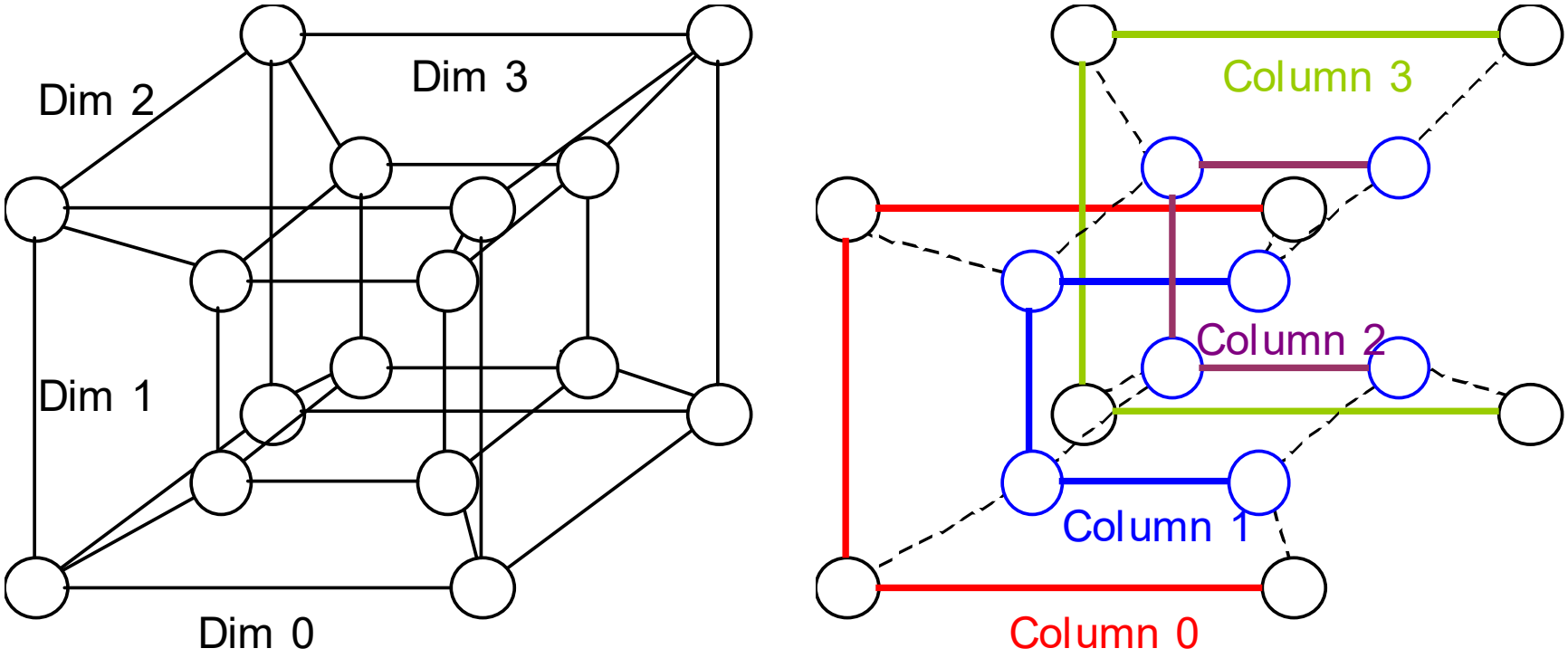
Embedding is a mapping of one network onto another
Emulation is one network behaving as another



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

A good embedding can be used to achieve an efficient emulation

Mesh/Torus Embedding in a Hypercube



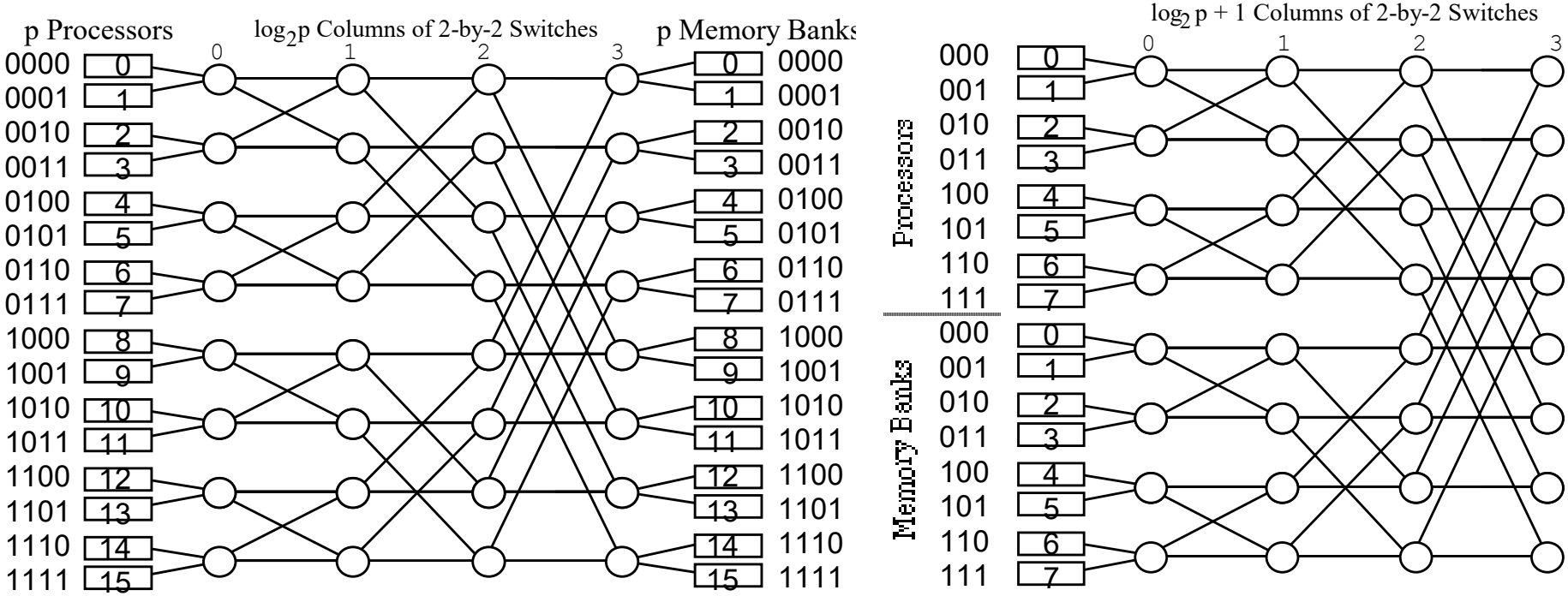
[Fig. 13.5] The 4 × 4 mesh/torus is a subgraph of the 4-cube

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

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

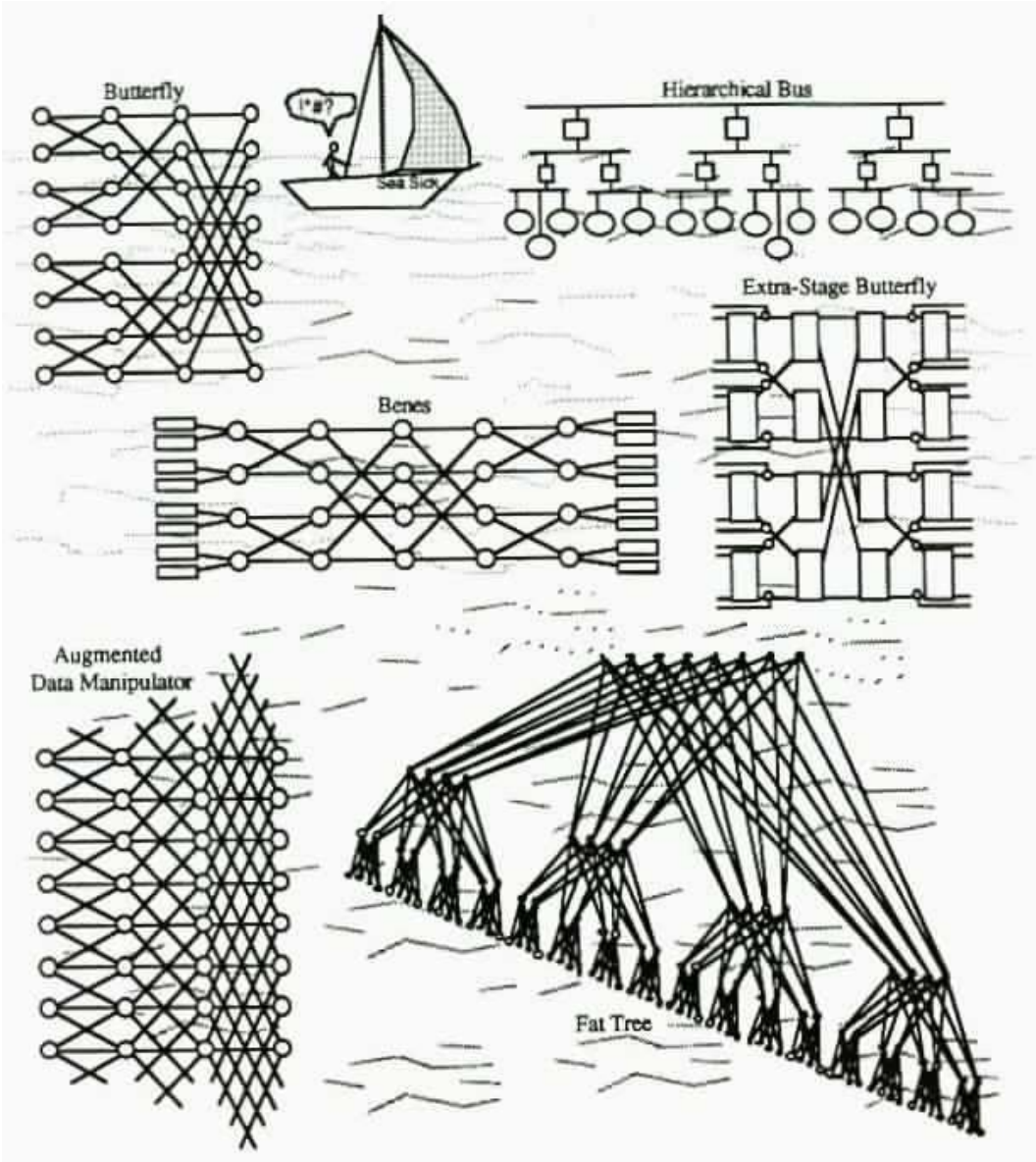


Examples of butterfly network and Benes network (back-to-back butterflies) shown above

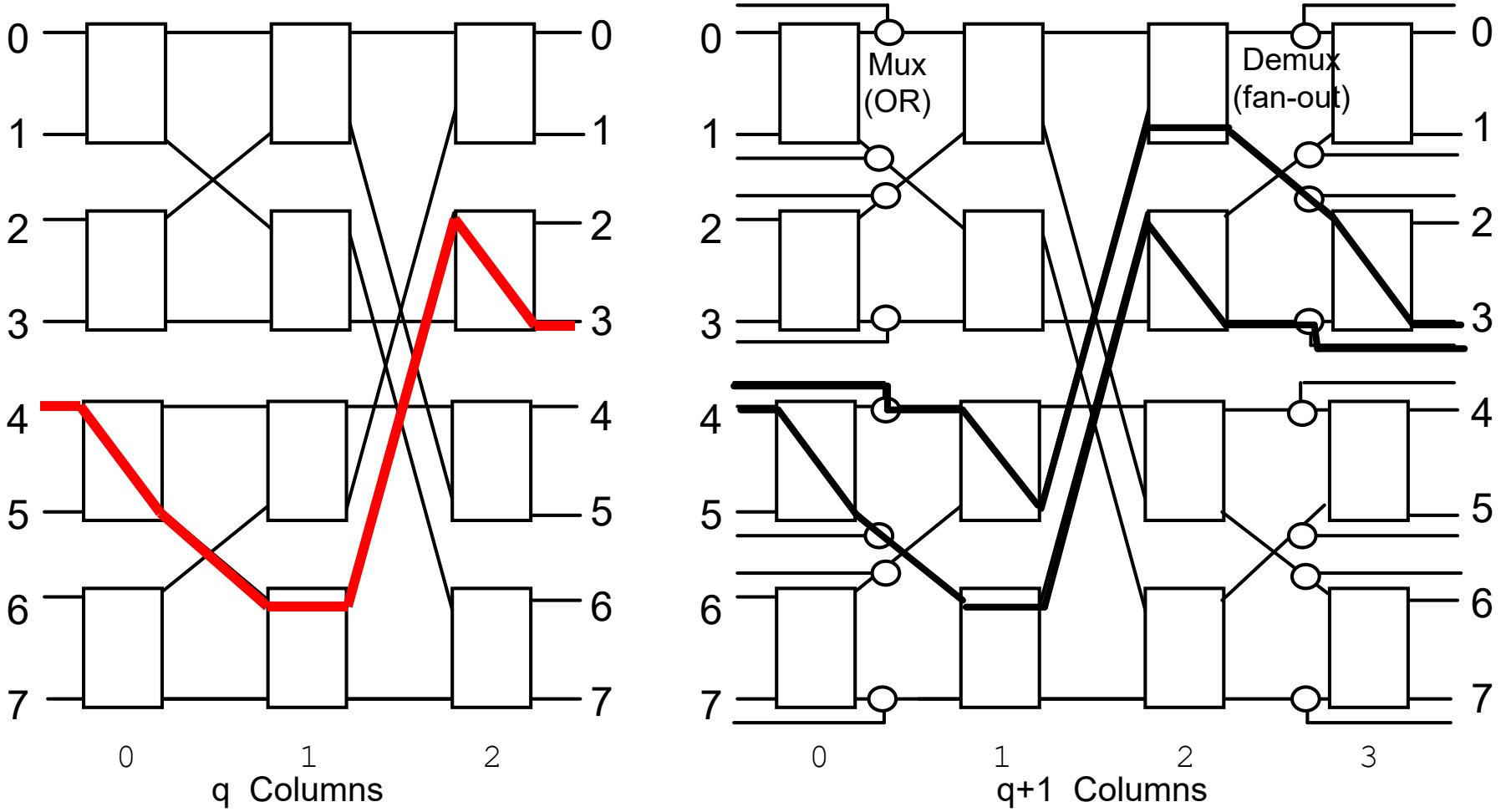
The Sea of Indirect Interconnection Networks

Numerous indirect or multistage interconnection networks (MINs) have been proposed for, or used in, parallel computers

They differ in topological, performance, robustness, and realizability attributes



Bypassing of Malfunctioning Switches



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