Algorithm Based Fault Tolerance in Massively Parallel Systems

20 October 1988

Mark Lerner

CUCS-360-88
Columbia University in the City of New York
Computer Science Department
New York, NY 10027

Table of Contents

1. Introduction 1
1.1. Algorithm-Based Techniques 2
1.2. Terminology 3
2. Overview of Fault Tolerance 5
2.1. Detection Phase 5
2.2. Correction Phase 7
2.2.1. Reconfiguration Subphase 7
2.2.2. Recovery Subphase 7
2.3. Hardware Methods 8
2.4. Software Methods 9
2.4.1. Language-Based Techniques 9
2.4.2. Operating System Techniques 10
3. Algorithm-Based Approach: General Principles 11
3.1. Redundancy 12
3.1.1. Invariants 14
3.1.2. Complete Redundancy 17
3.2. Malleability 17
4. Algorithm-Based Approach: Examples 18
4.1. Numerical Algorithms 18
4.1.1. Algebraic Invariants — Matrix Methods 18
4.1.2. Algorithm Invariants — Bypass Recovery and Extra Memory 19
4.1.3. Reconfiguration of Parallel Systems 20
4.1.4. General Numeric Fault Tolerance 22
4.1.5. Adding Redundancy — Checksums 24
4.2. Search Algorithms 25
4.2.1. Binary Search with Faulty Comparisons 25
4.2.2. Heuristic Search with Fail-Stop Processors 26
4.2.3. Ordered Search with Dictionary Machines 27
4.2.4. Crash Resistance — Ordered Storage Structures 28
5. Conclusion 30

List of Figures

Figure 1-1: Parameterization of Parallel Systems 4
Figure 1-2: Parameterization of Massively Parallel Processors 4
Figure 3-1: Clustering of Algorithms 12
Figure 3-2: The Vastness of Fault Tolerance 13
Figure 5-1: Examples of Invariants in use for Fault Tolerance 31

1. Introduction

A complex computer system consists of billions of transistors, miles of wires, and many interactions with an unpredictable environment. Correct results must be produced despite faults that dynamically occur in some of these components. Many techniques have been developed for fault tolerant computation. General purpose methods are independent of the application, yet incur an overhead cost which may be unacceptable for massively parallel systems. Algorithm-specific methods, which can operate at lower cost, are a developing alternative [1, 72]. This paper first reviews the general-purpose approach and then focuses on the algorithm-specific method, with an eye toward massively parallel processors. Algorithm-based fault tolerance has the attraction of low overhead; furthermore it addresses both the detection and also the correction problems. The principle is to build low-cost checking and correcting mechanisms based exclusively on the redundancies inherent in the system.

Before beginning the overview section, we anticipate a criticism of algorithm-based methods: that existing methods (such as checkpoints and triplication) are a solution to the problem. On the contrary, theoretical studies based on information theory have established that $\log(n)$ redundancy is adequate for general computation. Though the details are beyond the scope of this paper, we note the key features of these studies. To achieve reliable results on a synchronous network A of N processors which each enter a random state with probability p, one constructs a new network A_K that reliably simulates for T steps. This requires $O(\log(NT))$ time redundancy [Bierman]. ($O(\log^2(NT))$ space redundancy for real-time simulation). Asynchronous computation is more difficult, yet only needs a constant delay (e^{Bf}) despite compounded delays in buffers.

These results are significantly better than the $O(n)$ required for n-modular redundancy. Moreover, it has been observed [91] that "errors introduced by a gate must be offset by computation performed by that gate" since the encoding and decoding components are subject to noise. Since TMR doesn't do this, we have incorrect voter output with probability $E+(1+2E)+(3Z^2-2Z^3)$ where E is p(gate failure), Z is p(incorrect input to voter).

Thus assured that significant improvements can be achieved in the construction of fault-tolerant systems, section 1 presents the main ideas of algorithm-based fault tolerance. A broad overview of fault tolerance, is presented in section 2. Section 3 then describes general principles of the algorithm-specific approach, which have been derived from results in the field. Extensive technical examples are described in section 4, for both symbolic and numeric processing. The reader who is well versed in fault tolerance may go directly to sections 3 and 4.

The main phases of fault tolerance are *detection* and *correction*. Detection is the ability to discover and diagnose faults. Problems are then corrected through methods of reconfiguration and recovery. Several such methods are general-purpose. A common technique (called *voting*) runs many copies of the system and compares the results. A second method (called *checkpointing*) periodically saves images of the system's programs and data; if an error is detected by diagnostic routines, the computation can resume from the most recent checkpoint. If the fault was due to a permanent hardware fault, the problem must be corrected by manual repair or automatic reconfiguration. The voting technique masks faults so they do not produce errors, yet requires at least three times as much hardware. Checkpointing requires fewer resources, but may interrupt processing to save and recover data.

The general purpose methods must, in view of advances in computer architecture, be reassessed to make use of the new breed of parallel computers. Massively parallel processors (MPP) have recently been developed, composed of many thousands of interconnected processors, each with a small private memory and perhaps a shared memory as well. Traditional methods, such as triplication of a 100,000 processor machine, are unnecessary and unduly expensive. Likewise checkpointing has serious limitations, given the latency of mass storage devices.

Systems with many processors currently avoid faults by reliance on extremely reliable components. The mean time to failure (MTTF) would otherwise be as low as 1.5 hours for a 64K processor machine [67]. The use of highly reliable components with very conservative design techniques, though effective, is expensive and remains prone to occasional failures. For example, the cost of a 64K "Connection Machine" is $3,200,000 plus $264,000/year for maintenance [96]; this partially and indirectly reflects the cost of reliability. Nevertheless, the reliable components must be purchased for every copy of the machine. This money could be better spent on more powerful hardware and less costly fault tolerance mechanisms. This could be accomplished if fault tolerance were performed by software, since software can be reproduced at nominal cost.

A software approach exploits a prominent advantage inherent to parallelism, namely that failure of one component need not jeopardize the entire system. Less hardware investment is necessary, and the savings can improve the market acceptance of parallel computers. It may also facilitate the construction of more powerful hardware. There are without doubt serious technical questions about how to tap into this source of reliability, and hence applicability of the method is presently limited to specialized applications. One potential application area is artificial intelligence (AI), specifically search, associative retrieval, pattern matching and update of storage structures. This is evidenced by the development of many architectures expressly for AI [77, 120]. Other areas of applicability include adverse environments (air, desert, sea), life-critical (medical) and high-reliability (communications, fund transfers, international databases). Since maximizing performance is the *desiderata* of parallel processing, it is natural to develop the software-based approach.

1.1. Algorithm-Based Techniques

The algorithm-based approach will be discussed in detail in sections 3 and 4. A brief preview is presented here so the reader may contrast it with the other methods to be discussed. The goal of the approach is to build the detection, location and correction routines from the expected algorithm behavior. This exploits the relationships between partial results, as well as the anticipated input/output relationships. We note that communication networks are designed with a similar approach, because the higher levels of a communication protocol are insulated from errors at lower levels, and furthermore the low-level protocols provide reliable service even when confronted by hardware faults.

An algorithm may provide invariants about a system precisely because it is a concise statement of *what* is to be computed and *how* the computation should be performed. These are known to hold at certain points during algorithm execution regardless of the specific data in the system. The knowledge of "what" is computed allows use of domain knowledge to check the reasonableness of the results. Knowledge about "how" to compute the result can allow development of a fault tolerant version. These algebraic and

logical relationships can be exploited to improve reliability or reduce cost through improved error detection and fast correction. For example, in a binary search algorithm it is always the case that element $E_i < E_j$, *for* $i<j$. This property can be tested concurrently with program execution at low cost, and any violation signals an error.

An algorithm may be augmented to use redundant storage or perform extra computation if there is insufficient inherent redundancy to support the approach. Error-correcting codes are the most common form of augmentation, and take the form of additional hardware (memory, processors, communication channels) or extra data (storage caches, redundant executions). Several examples of augmentations are given in section 4.

A second view of algorithm-based fault tolerance depends on primitive operations that work even in a faulty environment. Programs can be defined with these primitives. The approach has been used to support *communication*, *agreement* and *mutual exclusion* to shared resources. Reliable communication (broadcast) to all processors [7, 35, 45]) and agreement by many processors (a form of *extrema-finding*) [29, 39, 46, 90] provide service even in the presence of faults. A reliable broadcast is important for communicating information between processors. Agreement algorithms are essential for tasks such as software voting and election of a master processor (so-called "leader election"). Fault tolerant mutual exclusion algorithms, necessary to share resources, have been developed [95].

1.2. Terminology

At this point we present a minimal amount of terminology, including the common metrics necessary for quantitative evaluation of a system. These terms are used throughout the literature.

A *fault* indicates an internal anomalous behavior [2, 12]. An *error*, on the other hand, is an external manifestation of the fault, through erroneous results "not consistent with the intention of the user" [58]. Thus an error occurs when the output differs from what it would be in the fault-free situation. A fault might not be directly detectable (or locatable) but its manifestations are. Clearly, fault tolerance seeks to produce no errors even when faults occur.

Comparative performance analysis has developed several ways to measure fault tolerance. *Fault coverage*, a static measure, gives the percentage of possible faults that will be detected. The dynamic behavior is described by other measures, such as reliability, performability and availability [40]. *Reliability* is the probability that the machine will not fail before time t. *Performability* is the probability that the machine will operate above some specified performance level. It is not directly related to the performance level, and the performability may either increase or decrease as the performance level changes. *Availability* is the expected value of computational capacity at time t, for example, the number of processors not removed from the processing array, or the expected service delay. The goal of 100% availability may be achievable if processing proceeds concurrently with correction, whereas a significant delay will result if periodic interruption is needed for checkpoint or recovery [107].

The architecture may be described by a *system parameterization* and *fault model*. These concepts provide the basis to design fault tolerance given system, by capturing the essential features of the architecture. The key parameters describe the processors, memory and communication. These emphasize

the differences between multicomputers and massively parallel machines. Examples of the parameterization and machine description are in figure 1-1. The massively parallel machines, equipped with vast numbers of processors, are fundamentally different from multiprocessors.

Conservative design methods, combined with general-purpose fault tolerance, are widely accepted as satisfactory solutions for the general class of multiprocessors shown in machines of figure 1-1. However, the methods do not work for newer machines, such as those of 1-2, when the tasks are interactive, or when very high availability is required. Furthermore, the latency of storage devices is increasing relative to the speed of the CPUs, which places checkpoint methods at a further disadvantage.

Processors	Memory	Communication
(Number, Instruction set, Speed/Processor)	(Amt. volatile (local / shared) Amt. non-volatile (local / shared))	(Topology, Synchronization, Capacity, Setup cost)
Shared RAM multicomputer: (10, general, 5 MIP)	(512/16Meg, 0/Infinite)	(Shared ram, locking, cpu-speed, none)
Workstation (100, general, 10 MIP)	(16 Meg/0, Infinite/0)	(Network, shared, 1Meg/sec, 0.05 sec.)
Connection Machine (64,000, small, 1 MIP)	(1K/0, 0/Infinite)	(binary n cube, asynch. router., 1Meg/sec, 0)

Figure 1-1: Parameterization of Parallel Systems

Processors	Memory	Communication
Huge Connection Machine (1,000,000, small, 10 MIP)	(1K/16 Megabyte, 10Meg/Infinite)	(binary n cube, asynch. router, 10Meg/sec, 0)
Huge Systolic Array (10,000,000, specialized, 100MIP)	(64K/0, 0/0)	(grid, adjacent PEs, 100Meg/sec, 0)
Extremely Large ALU (1, 100,000 bit arithmetic, 0.5MIP)	(1Gigabyte/0, 0/0)	(none, none, 0, 0)

Figure 1-2: Parameterization of Massively Parallel Processors

2. Overview of Fault Tolerance

As stated earlier, the two phases of fault tolerance are detection and correction. The *error detection* phase recognizes that an error has occurred, and may diagnose the problem by identification of the component and the type of fault. Next, in *correction*, a reconfiguration subphase removes the faulty components, and organizes the remaining hardware into a functional — and hopefully effective — system. An ensuing *recovery* subphase restores whatever data may have been lost or damaged and whatever executions had been corrupted.

2.1. Detection Phase

The behavior of circuits or programs can be monitored to determine if a fault occurs. The monitoring method depends on the class of fault. Deterministic (or *permanent*) faults are due to broken hardware. Probabilistic faults, which cannot always be detected, are of two types: *transient* and *Byzantine*. In the transient case, a broken processor gives consistent information to all neighbors, and it is unlikely that faults will be undetected. The Byzantine case is more difficult, since a faulty processor can give different or deceptive information to its neighbors. Faulty components may even conspire to subvert the system. Diagnostic testing of probabilistic failures is complex because the condition might clear up immediately after producing polluted data. One general solution is to encode data at a high level and recognize a fault when a non-codeword is encountered. A second solution employs voting among $2N+1$ redundant units, where N faults may occur. The errors of a component should be offset by the computation of the component.

All detection techniques share the common idea of utilizing redundancy, though there is significant variation in the form of redundancy. Five techniques are given here, which may also be used for correction. The first two, *component design* and *voting*, operate at the signal-level to produce circuits that withstand "adverse physical phenomena." Next, the method of *error detecting codes* can be used at all levels, including logic, data communication and computation. *Algorithms* and *assertions*, achieve fault tolerance in the software design process.

- **Fault-secure/self-testing components** (Logic-level fault tolerance). Fault-secure components employ specialized logic to monitor their own functioning. The approach builds fault detection and location into the components, which in turn indicate if they are functioning through an "I'm alive" indicator. The problems of fault detection and location are solved, it would seem, because the processors themselves indicate if they are functioning correctly. Moreover, the spread of contaminated information can be limited through suitable designs.

  High cost and imperfect design curtail the success of the fault-secure processor. The design and construction of such devices is expensive. Self-testing processors have failed on notable occasions [47]; a faulty processor can erroneously assert its "I'm alive" signal. In response, researchers have described special forms of fault-secure processors that are secure from worst-case error syndromes, but at a cost of 500% hardware redundancy. This is too expensive for most computer users, because the cost of preventing an error may exceed the cost of the error!

- **Voting** (General-purpose fault tolerance). This method executes redundant copies of the logic, program, or communication. The copies are compared at suitable intervals by a voting circuit or program [18, 38]. Reliability improves exponentially with the number of copies, under the assumption that errors are independent and the voting circuit does not fail. However, these assumptions are not universally valid. Coincident failure of several

processors can result from a systematic design fault, or from a transient physical condition exceeding the tolerances of many adjacent processors.

- **Error detecting codes** (Communication, computation and storage-level fault tolerance). Error codes maintain the integrity of data, which could otherwise be contaminated by errors during storage, communication, or computation. One commonly used code is the simple parity bits found in most computer memories. More powerful methods include Huffman encoding and linear feedback shift registers. See [128] for a good textbook treatment of the subject.

  An error correcting code (ECC) conceptually consists of two parts, a data portion and a check portion. The check is computed and written to memory when a data item is stored, and upon retrieval the check is recomputed. An error is detected when the stored value differs from the computed value. Correction of errors is possible with codes that contain additional information [128]. Codewords are generally defined over an algebraic field for reasons of computational efficiency.

  Computation, as well as storage, can be made resilient through these codes. For example, ECC's can be closed under multiplication or other operations [89]. On the other hand, symbolic algorithms present new coding problems, and the fault tolerance of these computations has not yet received widespread attention.

  Fault tolerance in the domain of symbolic retrieval by associative memories has been achieved by special hash functions [28]. The functions map similar input values to adjacent memory locations, where *similarity* is defined according to insertions and deletions (*i.e.* "minimal edit distances"). If a memory cell is inaccessible, then an adjacent memory will probably store a value which approximates the correct one. This value can be used when exact retrieval is not necessary, for example, if the input is noisy. However, the development of fault-tolerant representations and manipulations for arbitrary symbol structures is an open problem. For example, it is relatively easy to define syntactic similarity, but semantic similarity may be elusive.

- **Algorithm Properties** (Algorithm-based fault tolerance). Many algorithms have specialized properties, as will be detailed in sections 3 and 4. For example, an *a priori* bound on the norm is known for many matrix algebra routines. This can be checked at low cost. Likewise, the relationships between nodes in a binary tree are rigorously defined by the algorithm. Low-cost checking and correction methods can be constructed from these properties. Unfortunately, these tests can be expensive. In such situations, a lower cost property should be derived, if possible.

- **Assertions** (Program-level fault tolerance). The algorithm definition can be augmented through declarations or error handling routines, anticipating likely areas of faults. Declarations are added by programmers who are familiar with the application, possible faults, and specific fault tolerance requirements. The programmer can provide error handlers [130], as well as acceptance tests to detect failures. A formal view of the approach associates a first order logic predicate with each statement, and raises an exception if the predicate is false. For example, the early work on *executable assertions* [74] has evolved into a language-based approach that uses input guards in the Communicating Sequential Processes (CSP) approach, and also a general object-oriented style of fault tolerance [20, 37]. The approach currently is being extended for fault detection [72, 104, 105].

2.2. Correction Phase

If a problem occurs and is diagnosed, the fault tolerant system must then repair itself and deliver a correct result. This is done by *reconfiguration* and *recovery*, although a different approach is used in fault-masking systems that provide correct results without removing faults.

2.2.1. Reconfiguration Subphase

Reconfiguration removes the faulty components and modifies other parts of the system to provide the required communication and computation resources. Reconfiguration can be physical or logical. In physical reconfiguration, the system is fabricated with additional processors and switches; the interconnection between PEs can be circuit switched to bypass faulty components. Logical reconfiguration, on the other hand, reroutes message traffic and alters work assignments. Carter [27] has written an excellent survey of hardware fault tolerance.

The hardware parameterization (figure 1-1) should remain the same despite reconfiguration. The software could thereby operate with minimal modifications, without degradation of system performance. These goals can be achieved when there is sufficient spare capacity or components. When spare capacity is insufficient, it may nevertheless be possible to provide only the essential functions and degrade gracefully; an alternative is to run the task more slowly. In the example of a square processor array without spares, a half-size array can always be formed to run the task more slowly. Alternatively, logical reconfiguration of a parallel system can change the resource allocation to ignore defective processors.

There is a tradeoff between the computational power subsequent to reconfiguration (*performability*), and the time required to reconfigure (*availability*). Many final configurations can result from a reconfiguration, though it may be difficult to pick one that performs well. One centralized solution precomputes many reconfigurations based upon expected patterns of communication and computation. In the contrasting distributed approach, each processor validates its neighbors and reconfigures locally to produce a globally correct system. The reconfiguration scheme should not succumb to the dilemma of minimizing the performance degradation and yet completing the reconfiguration quickly.

2.2.2. Recovery Subphase

Restoration of important data must compensate for the errors introduced by faults. The state of important structures is usually restored to the "before-fault" instance. A weaker restoration produces a new intermediate state, from which the algorithm generates the correct answer. For example, when adding up the elements of a list, the ordering of the elements does not have to be retained, because it does not affect the final result. Other applications are immune to some errors, in which case the recovery criteria can be relaxed. A numeric example is integration by adaptive quadrature [94], in particular when an error in one interval can be ignored because it will have only a vanishingly small effect on the error in the total result. Such errors are sufficiently small to be indistinguishable from other unavoidable errors, due to the fact that most numeric algorithms are approximations. A third aspect of recovery is the need to adjust data and work allocations for the purpose of rebalancing the computational load.

The before-instance is usually recovered by use of techniques known as *backward* or *forward* recovery. The *backward* approach saves periodic checkpoints of the process image. For recovery, the system "rolls back" to a logically consistent checkpoint state. Costs can be significant because system operation must

be suspended during the checkpointing and because of the substantial I/O needed to copy a large program. Checkpoints are therefore taken infrequently and substantial work may be lost when the system reverts to an old checkpoint. The situation has improved somewhat through the development of faster logging-based techniques, as well as the selective checkpointing of only the essential state information [92, 101, 108, 112].

The checkpointing method has developed into *message logging* for parallel systems, which combines infrequent checkpoints with a repository of every transmitted message. Since messages are the only cause of state changes, logging the messages allows reconstruction of the state. The up-to-date data is recovered by replaying the logged messages since the most recent checkpoint [108, 110], and thus no computations are lost. A potential problem of both message logging and checkpointing, is cascading errors (the so-called "domino" effect). Replaying one message can generate a chain of additional messages, each generating yet more messages [24]. Cascade-free volatile logging methods provide potential performance advantages but have not yet been rigorously analysed [55, 109].

In contrast to the backward approach, *forward recovery* proceeds by analysis of program state. As described by Mili, forward recovery "uses the *natural* redundancy that exists between program variables rather than duplicating the program space. In terms of execution time, no overhead is incurred unless it is needed" [76]. The approach is appropriate if the system behavior is well understood [26] and errors can be isolated and corrected. This forward method limits the state change, since the full state is never rolled back. The approach requires accurate damage assessment to allow selective repair of the damaged portions.

The greatest success of the forward approach has been in numerical domains such as signal-processing and matrix-vector computations [1, 12, 123]. Success has also been reported for problems where the solution technique generates redundant solutions [80], as well as for transaction processing [10]. There is, however, some controversy about when to use forward recovery. Some researchers claim that performance decreases because interprocess synchronization is needed to correct the fault [53]. Others, however, find performance improvements because minor inconsistencies are allowed to develop, with correction at the end of the computation [10].

2.3. Hardware Methods

Hardware fault tolerance provides machines that work despite anomalous behavior in some components. The primitive hardware design should ensure the system is not disabled by a hardware fault. This provides the support essential for software methods.

The essential hardware characteristics should be preserved to allow efficient continued operation. The *number* and *capacity* of the processors and their interconnections are important because these determine the spare processing power available for backup and reconfiguration. The *interconnection topology* defines the communication paths between components. A redundant topology connects working components even if some parts of the system fail. Fault-tolerant tree architectures [48, 68, 131] provide hardware redundancy, whereas theoretical study shows the hypercube is well-suited for fault tolerance due to its redundant communication paths [49]. An empirical study using the Intel hypercube is presented in [17]. Other fault tolerant graph structures [67] include various kinds of arrays. A special issue of IEEE

Computer (June 1987) is dedicated to fault tolerant interconnection networks. Two popular topologies include the *binary tree* and the *n-dimensional hypercube.*

When the interconnection patterns are preserved by the reconfiguration, the software will need only minimal modification. A notable example is reconfigurable processor arrays. These consist of a grid of interconnected components and have the significant feature of allowing continued operation of all the functioning components subsequent to failure of a single component. Design techniques for these arrays include the Diogenes approach [30]. In this approach, a switch external to each PE provides the means to bypass the defective components. Although the non-faulty components can be used, the algorithm may have to adjust to retain efficiency. The major problem with the initial Diogenes approach is the delay due to long wire lengths [124], since unmatched delays between components may decrease performance. This difficulty is solved by algorithm modifications. One modification technique is *retiming*, which adds delays to some communication links to prevent internal bottlenecks. The internal system timing does not change, despite the delay of bypassing defective PEs [125].

Reconfiguration should retain the critical timing and synchronity aspects of high performance parallel processing systems. A synchronous system is bound closely to the system clock [73], with each processor executing the same instruction stream. These systems provide excellent performance, provided the problem "fits" the system. Asynchronous systems, on the other hand, allow greater independence in both the instruction sequences and the communication patterns. Reconfiguration is easier with an asynchronous system because timing is less important; the programs can be changed without affecting correctness, although load balancing becomes an issue. This leads to the question of the best way to reconfigure an asynchronous computer computer [123]. Synchronized systems include the ICL DAP, the NASA MPP and the Goodyear STARAN. Asynchronous systems include the Intel HYPERCUBE and the BBN Butterfly, as well as NYU UltraComputer, IBM RP3 and Columbia DADO machines.

2.4. Software Methods

Software techniques extend the hardware capabilities to build fault tolerant systems. Software can be reproduced cheaply. Thus, it would be economical to depend on it for fault tolerance.

2.4.1. Language-Based Techniques

The first type of fault tolerance is language-based [105]. Programming tools are used to construct programs that work even if a fault occurs, as in specialized languages that can be annotated with logical assertions. These assertions can be processed by automatic theorem-provers to determine if the program will run correctly in the event of hardware errors. Examples of this approach assume a volatile main memory and a nonvolatile disk [33, 34]. Faults result from "the influence of adverse physical phenomena," which may affect stable storage. These errors are modelled by a special fault-operation called DECAY. The work includes proof rules with predicates for "all-perfect" and "gracefully-degraded" operation. Functional requirements specify the behavior, such as the ability to read/write the disk, storage atomicity, and periodic repair (to return the system to all-perfect status). Unfortunately, it has not been extended to parallel systems. Other formal approaches are given in [19, 76, 102, 118].

The second method defines fault tolerance within a programming paradigm, such as dataflow programming or object-oriented programming. Specifically, the dataflow programming model is side-

effect free and describes state by tokens. The absence of side effects simplifies recovery [84, 100] simply by encapsulating smaller substates that interact in well defined and predictable ways. State is described by the "active unconsumed tokens" and thus reliability is achieved by keeping duplicates of these tokens. Multiple copies are maintained in different physical processors and deleted when the primary is consumed. Furthermore, a technique called "token chasing" can determine if a program will work correctly even if some tokens fail to fire [8, 9]. These ideas have recently been extended with a software engineering approach for forward error recovery using semantically equivalent abstract data types with maximally disjoint fault spaces [81].

2.4.2. Operating System Techniques

A second form of software provides fault tolerance through a reliable virtual machine [3]. An easily used language provides access to the machine. Unfortunately, few performance studies rigorously address the applicability of OS techniques to massively parallel systems [113], and none have studied interactive processing on those systems. This may be due to the fact that massive parallelism is extremely new. There has been research, however, on specialized real-time systems for critical tasks [59, 101, 107].

A major form of OS fault tolerance is based on *atomicity*, where computations are executed as *atomic actions*. An atomic action has an effect on other actions only if all steps complete successfully. If the action fails, no changes are made, and a backup process can be invoked. Atomicity can be specialized for an application area, as in *serialization theory*, which applies to transaction processing. Examples of OS fault tolerance include the CLOUDS operating system, which is based on object-oriented invocation, running on general-purpose processors with an Ethernet interconnect and dual-ported disks. The fundamental operations are written as recoverable object classes, with the implementation mechanism hidden. For example, the CLASS RECOVERABLE INTEGER depends on logs written as a side effect of the base class. Operating systems also use other approaches, such as recovery blocks [51] and N-version programming [5].

One means of achieving atomicity is through checkpointing and rollback, which can be done invisibly by most fault tolerant operating systems. A contrasting approach is taken in the GUTTENBERG [126] system, where recoverable communicating actions guarantee recovery and consistency. This work emphasizes several general principles, including a "communication dependency relationship" that is distinct from the data dependencies of EDEN, CLOUDS, or Nested Transactions. The GUTTENBERG system claims to be an improvement over checkpoint methods [108] because "we contend that checkpointing all processes involved in a reliable distributed computation in the system is an unrealistic burden on the kernel ... [and] since behavior may be time-dependent the behavior of the second execution may not be the same as the first." However, checkpoint methods have recently been improved in the form of volatile logging [111] operating on piecewise deterministic programs. The determinism assumption addresses the issue of time-dependency.

It will be important to see how performance of these OS methods develops. We note that these methods cannot be utilized within individual PEs in existing MPPs simply because they have no onboard operating system. In the case of the Connection Machine, for example, not even a runtime kernel is available. Moreover, there is no assurance that the I/O capacity of an MPP would be able to support the checkpoint/restart algorithms. Furthermore, the diminished availability due to checkpointing or logging

might be unacceptable.

3. Algorithm-Based Approach: General Principles

Having earlier indicated the preference for the algorithmic approach — because of potential efficiency in parallel systems, because a small amount of hardware redundancy is adequate, and because the program may not require modification by the user — we will now present details of the approach. This will proceed in two parts. First is an exposition of the general principles and premises of the approach. Then, in section 4, we demonstrate the utility of the approach through extensive examples.

No theory of applied fault tolerance has been presented in the literature [13], although an important theoretical basis was established by von Neumann [127] and is being extended by Gacs and Pippenger [43, 44, 91]. We therefore group algorithms according to discernible features of the fault tolerance methods. This taxonomy is a step toward a theory and methodology of applied fault-tolerance. The features include problem structure, malleability of the solution technique, and inherent redundancies of the domain. Explicit mention of these features should help us to understand existing fault tolerance schemes and to develop new ones. We consider here the two domains of numeric algebra and symbolic search.

Figure 3-1 shows that algorithms cluster into two main groups according to characteristics such as *behavior* and *structure*. This clustering helps to select fault-tolerance methods for new algorithms, because the algorithms within each cluster tend to share similar fault-tolerant mechanisms. Within a well-structured mathematical domain there may be information that allows a problem-solving technique to check its results. For example, multiplication is defined on well-ordered sets, with clearly defined distance measures. By monitoring the change in distance during execution of the algorithm, we can detect unexpected changes and attribute them to faults. An alternative, less-structured, example is the non-algebraic domain of symbolic search. Often there is meager information about the relationships between elements of the domain; worse yet, there may not be a partial evaluator to describe how close a partial solution is to a solution node. This makes it difficult to use algorithm-based fault tolerance, because we cannot tell if a node is on the path to a correct solution. Indeed, search is widely known as a "weak" method precisely because there is little information about the search space. It has nevertheless been possible to exploit some formal definitions within search problems to exploit the algorithm approach. For example, estimates on the distribution of answers in the search space have been used in fault tolerant search [54, 103].

The second characteristic is how *well-behaved* the algorithm is, *i.e.*, whether a small perturbation in the computation produces little or no error in the output. Many numeric algorithms are well-behaved. Small variations in the data or internal values do not result in a large error of the output. Although poorly behaved numeric algorithms are known, they are rarely of interest because well-behaved alternatives can be used. On the other hand, many search algorithms are poorly behaved, since a wrong decision can result in completely missing an answer node. Nevertheless, some well-behaved search algorithms have been devised, including Rivest's "adversary answering" scheme (to be discussed below). This gives the correct answer even under worst-case error assumptions. Fault tolerant algorithms have been developed for search of ordered data structures as well as for heuristic search through randomized backtracking.

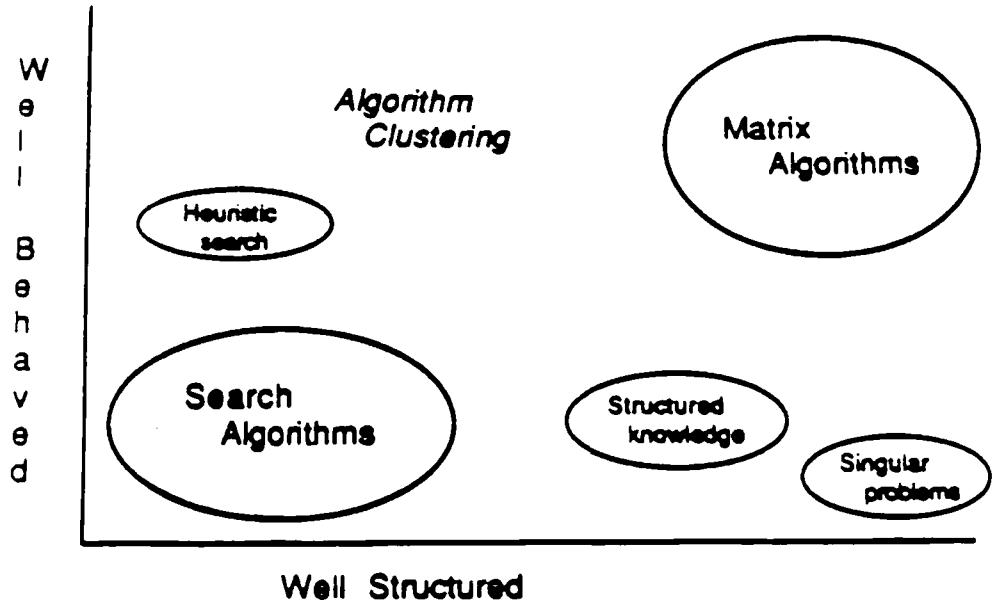


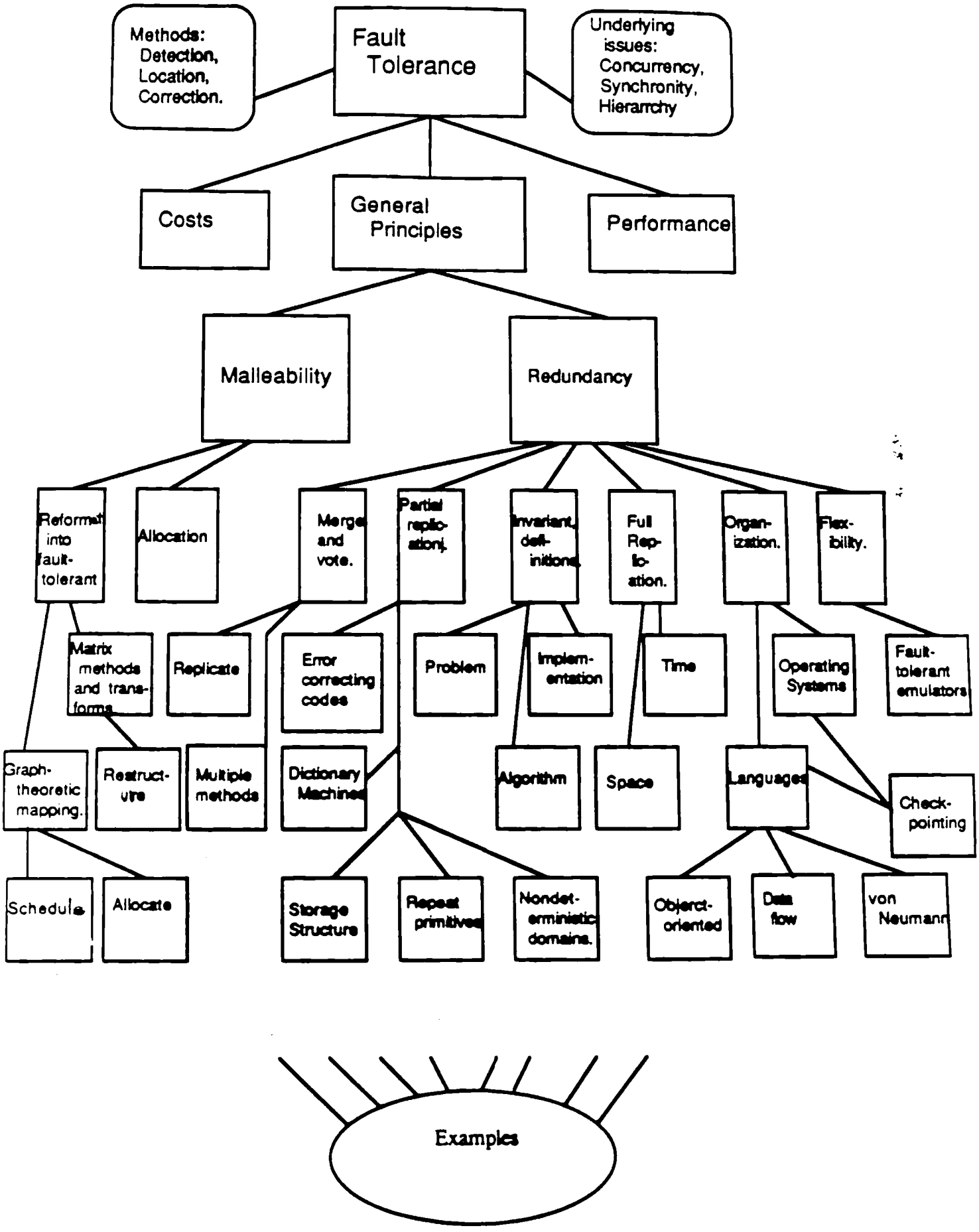
Figure 3-1: Clustering of Algorithms

The algorithms can also be taxonomized in a second way, according to the perspectives of *malleability* and *redundancy* shown by the hierarchical chart of figure 3-2. Malleability describes the ease with which the information and hardware can be restructured, and also the ability to continue functioning after a partial loss. This contrasts with brittle machines, where the loss of a component cripples the entire machine, or brittle computations where one factor of a product can completely change the result. Highly malleable hardware systems include restructurable arrays and distributed systems with flexible interconnection networks. Less malleable systems, such as the Alliant FX/8, can be reconfigured offline to provide high availability with low hardware redundancy; the Alliant's self-scheduling system software can automatically adjust to the number of processors. Brittle systems require that all components are fully functional, although frequently some malleability is provided by the ability to deconfigure defective memory. Malleability has been discussed under many different terms in the literature; including conformability, allocation and reallocation.

3.1. Redundancy

Figure 3-2 shows that redundancy is a key aspect of fault tolerance. Four flavors of redundancy are presented: these are *partial*, *invariant*, *full*, and *organizational*. The first two introduce less than 100% replication, whereas the second two require many redundant copies. An example of partial replication is checksums in numeric problems. Replication in space and time are examples of full redundancy. *We note, critically, that no practical instrument has yet been designed to measure how "redundant" a program is.* Comparison to a theoretical lower-bound may help, but it does not show where the redundancies are. It is therefore essential to observe the different kinds of redundancy and to develop metrics for measuring their effectiveness, when possible.

Figure 3-2: The Vastness of Fault Tolerance



3.1.1. Invariants

An invariant is a constant property of a domain, algorithm, implementation, or model. It provides partial fault coverage. Many systems test these invariants and execute corrective routines when an invariant has to be restored. This approach brings us closer to the goal of fault tolerance that operates at zero cost in the fault-free case.

The use of invariants suggests the following methodology for fault-tolerant programming. The key idea is to *(1)* identify invariant problem features, in particular the inherent redundancies. These features can then *(2)* be classified according to the faults that result in a violation of these invariants. The system is *(3)* constructed to check for violations of the invariants, and *(4)* use corrective actions if a violation occurs. It may be necessary to *(5)* augment the system by development of new invariants and redundancy techniques. This has been demonstrated with iterative algorithms [4, 14, 97], and makes use of coding techniques. The techniques are not widely used because the identification of redundancies is done by the system designer. Programming tools have only recently started to receive attention (for example dataflow analysis and VLSI design tools [8, 62]).

Invariants are the basis for the general fault tolerance approach defined by Dijkstra. He introduced *self-stabilizing* programs which are guaranteed to converge to a legitimate state in a finite time. Work in this area defines a system in formal terms, for example through formal grammars of strings that match a characteristic regular expression. Context-sensitive rewrite rules describe the possible state transitions. Such systems are self-stabilizing if "starting from any state, the system will reach the [distinguished] home state." [25] The system must also satisfy a program specification, and guarantee progress toward the result. One domain where the approach is useful is in "mutual exclusion," which provides exclusive access to a shared resource passing a token between different requesters. Transient failures may induce errors by injecting duplicate tokens into the system. A fault-tolerant distributed algorithm [25], discussed below, shows an example of self-stabilization for the mutual exclusion problem.

The self-stabilization and other properties should be proved by the rewrite rules of the system. The considerable advantage of formal correctness is offset by the major drawback, namely the difficulty in developing the system description and proof. The formal description can be very difficult to write, and the proofs must be generated manually. Moreover, an algorithm cannot currently be derived automatically from state specifications, since this requires solution of the automatic programming problem. Nevertheless, the approach is of considerable value when the algorithm is sufficiently simple.

We now present a perspective on the many kinds of invariants that can be used in system construction. A *domain invariant* is a property of the problem, whereas an *algorithm invariant* is a property of the computational method. Both describe a high-level characteristic that can be checked to provide assurance that the computation is proceeding correctly. Extending this idea, an *implementation invariant* is a property of the particular way the algorithm is programmed. A *model invariant* is built into the programming methodology, such as functional or object-oriented programming.

Instances of *domain invariants* are described by theorems. One such theorem, due to Parseval, states that "the sum of the squared input energies is a constant times the summed squared output energy" [4]. This gives a relationship between the input and the output values, and can be tested during program

execution. Any violation of the invariant indicates an error. Further analysis may be necessary to decide if the error is attributable to a hardware fault as opposed to computational errors that might result from an ill-conditioned problem. Identification of a domain-invariant requires detailed knowledge of the problem domain, and this formal knowledge is not known for many problems.

In contrast, the *algorithm invariant* depends on the formalization of the solution, not upon the problem itself. An example of an algorithm invariant is the balance condition of a tree. Specifically, the difference in height between the two subtrees is well defined, as is the ordering between adjacent tree nodes. These properties can be checked periodically to ensure the system is behaving correctly. Likewise, some matrix algorithms have *a priori* bounds on the values of the norms. If these bounds are violated it indicates that an error occurred.

Implementation invariants are properties of how the algorithm is written, and are not due to the problem itself. Checksums, in which a sequence of data values is summarized in a scalar, are basically signatures which validate the data. An error is indicated when the checksum does not match the data. Well-known techniques, described in [128], include Parity, Hamming, and finite field codes. Some codes allow recovery of correct data from the errorful data, for example by augmenting a matrix with additional rows and columns. The addition of error coding augments the data to increase redundancy, thereby providing an invariant.

A *model invariant* considers either the primitive operations or the programming model. One study with faulty primitives assumes a model of computation with a comparison operator that may lie with probability p [98]. A correct algorithm is designed despite the errorful operations. This is based on a study of invariant conditions, such as the number of data items examined during execution of a loop.

The invariant can be a characteristic of a programming model. This can determine the appropriate recovery techniques. In functional systems side effects do not occur and checkpoints do not have to maintain the full environment of each function. This is one reason for the interest in functional languages [84], as well as graph reduction languages [32] where the behavior of a function is determined entirely by the parameters to the function. Such function-based methods allow reassignment of processing resources, because information is not preserved between invocations. A languages can also incorporate assertions, which are first-order predicates that are added to the source code definition of a program. Such augmentations are similar to Floyd-Hoare proof rules that augment programs with correctness proofs. These are subsequently verified during program execution. Special languages have been developed to express these assertions, complete with systematic techniques to verify the programs [19, 33, 34, 74, 75, 76]. In object-oriented programming (for example fault-tolerant C++) the recovery behaviors are defined on the basic data objects and are inherited by all members of the class.

In all cases it is important to note that testing an invariant does not guarantee total correctness of the execution. Satisfaction of one invariant does not guarantee that all other invariants are also satisfied. Furthermore, it may be too expensive to fully utilize an invariant, in which case partial fault checking may be the only reasonable alternative. For example, consider the problem of garbage collecting unused memory and maintaining a list of unused storage. The invariants include the fact that no "in-use" element should give the address of an unused block. It is infeasible to test every pointer access against a list of

every free location, yet a suitable design choice will allow partial fault coverage. In particular, a 2 *heap garbage collector* has the characteristic that all free storage migrates to a single area in memory, for subsequent allocation according to a stack discipline. Partial testing of the invariant can be implemented by checking the bounds on each access to be certain it does not reference the wrong area.

Invariants have been referred to as constraint predicates, or algorithm-oriented tests, that give acceptable limits for the errors in a processor. These tests are constructed from the natural constraints on the application. The metrics should evaluate the percentage of faults covered by the constraint. Such predicates can be easily built to never erroneously identify a correct result as faulty. However, it is more difficult to evaluate the ability of a predicate to prevent wrong results from being presented as valid. For example, the following predicate on the computation of the *surface area* of an object will only prevent grossly wrong errors. More detailed information is necessary to improve fault coverage.

```
Predicate Ar(area: real): Boolean;
  if (area < 0) error();  else return correct;
```

The garbage-collection example given above is a second instance of a correctness predicate.

A major difficulty with the approach has been the lack of guidance in selection of constraint predicates, particularly the lack of metrics to measure their effectiveness. One solution is to initially formulate the specification based upon metrics, which are then applied to a distributed problem solver. This yields a constraint predicate that embodies the desirable fault detection (and correction) function of the resulting algorithm. A key issue is what *metrics* can be used to guide the development of constraint predicates for general problem classes. The metrics are important because they guide the selection of features extracted from the application, and this leads to executable assertions which compose the constraint predicate.

A basis for constraint generation is that each intermediate result should satisfy the three criteria of *progress*, *feasibility*, and *consistency*. The first criteria corresponds to progress toward a goal or a final solution. For example, an iterative algorithm may demonstrate progress through the property of a *convergence envelope*, such as:

$E^k = |U^k - U|$ Where U is correct, and k is the step.

$\|E^{k+1}\| < \|E^k\|$ For suitable norm or measure.

Either local or global information can be the basis of the convergence tests. Global tests are more common, but bear the additional cost of at least $\log(n)$ for communication alone. Local tests can therefore be completed more quickly because they do not require communication. However, the local test may have less information at its disposal.

In addition to convergence, the predicate should be feasible. For example, boundary conditions are natural problem constraints frequently found in physics or engineering problems. Branch and bound algorithms dynamically narrow the bounds on feasible solutions, during a search process. They thereby have the feasible constraint of excluding certain values as erroneous. The third criteria, consistency conditions, is a development method for constraints. For example, locally computed values in the conjugate gradient algorithm must be consistent with the values received from other processors [6].

3.1.2. Complete Redundancy

Faults can be masked out by use of duplicate components, since it is unlikely that a majority of the units will fail simultaneously. The hardware, program and data are replicated, with voting to compare the results. The majority answer is selected in an N-Modular-Redundant (NMR) approach. Time can be traded for hardware through repetition of the computation at different times, in so-called *time redundancy*. In both cases a voting circuit selects the answer, generally by a majority rule. The method is expensive in hardware cost, but is inexpensive in software cost. The reliability improvement is exponential under the assumption of independent executions.

Voting can be done *continuously* or *periodically*. In the first case, a comparison is made for every computation (or group of logic gates). This is extremely popular for logic-level or processor-level comparisons. However, continuous comparison is sensitive to design faults, particularly if the same design flaw is present in each replicant. A second problem with voting is the need to keep the components close to each other (to minimize propagation delay); this physical adjacency makes them potentially subject to the same physical sources of error. Third, it is suboptimal in the sense that it wastes components and may be subject to errors nevertheless.

Periodic comparisons, in contrast, can be performed by an operating system. The system state is transmitted by each replicant to a voting unit. A highly compressed summary of the information (known as a "signature") may be transmitted to avoid the expense of communicating the entire memory. This periodic approach suffers because the systems must be in a consistent state. Synchronization and quiesence of the subsystems should occur before comparison. This may detract from availability and performability. Complete redundancy techniques will not be discussed further in this paper because they are not algorithm specific.

3.2. Malleability

A system needs to reconfigure in order to sustain performance despite loss of components. Two issues arise here. *Hardware* malleability requires specialized design techniques, such as redundant busses and defect isolation. *Software* malleability is the capacity to modify the software (program, data and control structure) to continue execution on the modified hardware. Malleable systems are easily reconfigured. The primary software reconfiguration techniques symbolically formulate the computer system as a connected graph which describes the processes, processors, and interconnections. Manipulation of the graph into a new form, one that can tolerate loss of some vertices and edges, gives a mapping of the software onto a fault-tolerant hardware system [50, 70, 93]. Correctness is demonstrated by showing the new mapping preserves essential properties of the initial one.

The initial graph-based approaches had to make restrictive assumptions to ensure a solution could be obtained. For example, a process was reassigned only to a processor of the same type and capacity as the original assignment. More recent work has relaxed these assumptions through heuristic approaches (see page 23). Heuristic methods have had good success at reasonable cost [56, 86]; this is fortunate because the general graph-based methods are equivalent to the *NP*-complete subgraph isomorphism problem. To avoid this difficulty, new configurations can be precomputed by well-known heuristic methods [23, 31], including dynamic programming [60].

Matrix-based models can also describe the system. These models use matrices to represent the resource capacity and computational requirements. They use linear algebra to manipulate the representations through correctness-preserving equations, which restructure the system into a fault-resilient form (see page 21). The matrix and graph methods are equivalent due to the bijection between graphs and adjacency matrices, and thus the choice of representation is based largely on convenience. The matrix methods are appropriate in array-oriented domains, whereas the graph methods can be used in highly flexible domains such as networks.

4. Algorithm-Based Approach: Examples

Having reviewed the main principles of fault tolerance, we now turn our attention to examples of algorithm-based fault tolerance. Few reports of this approach are presently available in the literature. This is an artifact of the newness of the approach. It is a reasonable expectation that more work will be reported as MPPs continue to be popularized.

4.1. Numerical Algorithms

Three primary methods have been developed for fault tolerance of numeric algorithms. Low-cost verification of results is built from theorems about the expected ranges of results. The most useful theorems state an invariant condition that cross-checks intermediate results. Such verification conditions are not always known, and therefore we may develop new methods by fusing an algorithm with an error-correcting code. This creates a hybrid algorithm with improved reliability, but greater cost. Moreover, the data and fundamental operations are not always amenable to such coding techniques. Therefore, it may be necessary to employ a more powerful technique. One such approach builds a directed graph to describe the computation, faults, and corrective actions. This graph is manipulated to add fault-tolerance to a system, or to measure the fault tolerance already in the system. Unfortunately, manipulation of such a graph into an optimally fault tolerant one is generally intractable (NP-complete). We therefore construct fault-tolerant systems that have slightly more redundant components than is theoretically required.

4.1.1. Algebraic Invariants — Matrix Methods

A rich collection of algebraic invariants characterizes many numeric algorithms. The invariants provide relationships between the program states at different steps of execution. These form the building blocks for low-cost tests that detect, locate and correct faults. System-dependent invariants give the essential execution properties such as timing, topology and resource allocation. These invariants should be preserved through reconfiguration to sustain system performance. The invariant may be intrinsic to the computational method, or it may be added, as in checksums where redundancy was added. Invariants have been developed for fault detection and recovery of QR factorization, the conjugate gradient method, orthogonal transforms, and partial differential equations (PDE's).

One particularly strong invariant is *orthogonality*. Orthogonal matrices occur in many areas, including the solution of simultaneous linear equations. A matrix Q is said to be *orthogonal* [94] if $Q^TQ=I$; this is a domain invariant. For any orthogonal matrix Q and any vector x, $\|Qx\|_2=\|x\|_2$. These two equalities can be tested concurrently with program execution, and non-orthogonal behavior indicates a system error. An experimental evaluation of the approach for LU decomposition [4] uses orthogonality for detection under

various fault models. It found fault coverage as high as 90% - 100% when accurate arithmetic (32 bits) was used, though it was as low as 70% with smaller word sizes. This approach has been used to develop fault tolerant FFT and singular value decomposition on the hardware of VLSI arrays [16] and the Intel hypercube [17].

A second example is solution of a linear system of equations by QR factorization, which computes a sequence of matrices $\{A_k\}$. The algorithm has nice error properties: the eigenvalues and L_2 norm are constant. These properties can be checked concurrently with algorithm execution, as is done in the Gentleman and Kung QR implementation consisting of triangular systolic arrays with two additional rows of processors [4]. Computation and comparison of norms is achieved without affecting the systolic flow. The hardware checkers cost about 13% extra for a 16×16 array. Other examples include exploitation of the iterative nature of the algorithm to do fault tolerance of LaPlace and Jacobi methods [52]. This depends on boundary values, monoticity and local equilibrium constraints. The most recent rigorous analysis of algorithm-based fault tolerance is based on floating point error analysis of triangularization procedures, using a checksum scheme [69].

Unfortunately, it can be quite difficult to find or use an invariant. In particular, the obvious property may be too expensive to use. In this situation, the derivation of a new property is a good compromise. For example, the conjugate gradient algorithm has error correcting properties, though they are expensive to use. A low-cost variant checks algorithm-specific properties at the end of each iteration. This performs error-detection at a moderate time penalty of 20% by testing the orthogonality of the prior iteration [6]. Specifically, the conjugate gradient theorem gives a checking condition that requires time $O(k^2\frac{N}{n})$. This is reduced to $O(k\frac{N}{n})$ by only checking the orthogonality of the new residue to the previous vector, since this can be computed without a distributed matrix-vector product. Thus, algorithm information is of help in control of faults.

4.1.2. Algorithm Invariants — Bypass Recovery and Extra Memory

Information about the algorithm can diminish the requirements for saving state information, as well as the amount of lost work that we must recompute. The novel technique called *bypass recovery* [88] is one example that does not repeat the computations that are lost through faults. Instead, the approach uses algorithm properties to continue running an iterative process without restoring the lost information. The bypass recovery method has been demonstrated for the multi-grid method of solving partial differential equations (PDE). Such equations are solved on a near-neighbor architecture by a *discretization* and an *iteration* processes. Discretization places a piece of the equation onto each processor. Iterations consist of local problem-solving followed by communication of the results to neighbors. The loss in accuracy from missing a few mesh points is claimed to be small, since only one small portion is in error. Reconfiguration adapts the algorithm subsequent to the fault, by a *rediscretization* process. This changes the model around the faulty processor. Although the order of error in the neighborhood of the fault increases, the overall results do not change significantly. Theoretical and practical results show "the effect of a faulty processing unit is almost negligible." However, their analysis is not rigorous, and no simulation results are shown. A similar self-correcting principle has been described for *resistive neural networks* [57]. In essence, the extra work done in each processor ensures that adequate state remains to continue operation subsequent to loss of some processing elements. In the case of neural networks, the

threshold behavior of neurons masks out the faults.

4.1.3. Reconfiguration of Parallel Systems

Subsequent to removal of a faulty component, it is necessary to reconfigure the system. This reconfiguration depends on the class of machine, such as the Multiple Instruction Multiple Data (MIMD) machine. These are composed from many — perhaps tens of thousands — fully functional autonomous processors. Some MIMD machines can be easily reconfigured because the multiple processors can execute independent programs. Each processor executes a different instruction stream, stores data in its local memory or a shared memory, and communicates with other processors. The communication patterns are defined by an interconnection topology. The local-memory model has greater promise for reliability, compared to the shared memory machine, because an errant processor cannot easily destroy main memory.

An algorithm-based approach to fault tolerance is appropriate for these machines running specialized algorithms, since it is important to preserve the mapping between the algorithm and the architecture. A fault in a MIMD machine manifests itself as a node or link failure which the system must remove. Subsequent adjustments to the algorithm and the data allocations may be required to efficiently obtain correct results; the processors can be individually changed because each one can execute its own instruction stream. A centralized adjustment uses complete information about fault locations. On the other hand, distributed reconfiguration uses local and partial information. Such strategies do not have a single point of failure, and thus are not vulnerable to the failure of a central host. Moreover, the system can be easily expanded because it does not have to stay within the host's capacity. Distributed reallocation algorithms have been discussed by Uyar, Banerjee and others.

Data reallocation is the major issue in MIMD fault tolerance, and allocation decisions should be based on the topology. For example, the best known reconfiguration methods for generalized interconnection networks are "uniform data distribution" (UDD) and "reduced data column" (RDC) methods. The cost of these methods has been extensively analyzed by Uyar [121, 122, 123] through a model with parameterized communication costs. The processors must also save state by periodic checkpointing of intermediate results to adjacent processors.

The parameters of Uyar's model includes the number of faults, the computational capacity, byte v.s. block-oriented communication and I/O load. The work defines several metrics, including *performance degradation*, $PD=\frac{T_{faulty}-T_{normal}}{T_{normal}}\times 100$ (where T_{normal} is the time in a fault-free machine, and likewise T_{faulty} is the time in a faulty machine) and analyses PD under various models. This measurement is meaningful because it shows how much the system's performance degrades for a given task. Performance degradation is minimized by the UDD method.

Reconfiguration is more complicated for the class of Single Instruction Multiple Data (SIMD) machines. Many simple processors are interconnected, with every processor executing the identical instruction on local data. Tight data synchronization allows very high performance, provided the propagation delays between processors match the processing delays within the processors. However, it is necessary to adhere to stringent timing constraints in order to achieve maximum processor utilization; otherwise cycles are lost to communication delays. Therefore the algorithm and data must both be reconfigured. The

reconfiguration is done by algorithm transformation techniques that preserve crucial invariants — such as timings and algebraic properties. The primary goal is to design algorithms and architectures that must *conform* to each other, even when the architecture changes dynamically.

Two effective hardware reconfiguration strategies for processor arrays are the "reduced row" (RR) and "reduced column row" (RCR) methods. These can be implemented at reasonable cost, and maintain the basic topology though not the system size. The RR technique always removes the entire row that contains any defective processors. It requires hardware support in the form of one switch per processor to allow row deconfiguration. However, this cannot preserve the array shape, and each reconfiguration removes N processor (in an N^2 size array). The second method, RCR, alternates between removal of rows and columns. It retains the array shape, and removes fewer processors than the RR method. These improvements result in better performability, thereby offsetting the more complex switches needed to deconfigure columns as well as rows.

Software techniques have been developed to create new algorithms that compute the same result as the initial algorithm, yet conform to the architectural capabilities of the reconfigurable architecture. This is done by application of a suitable sequence of correctness-preserving transformations [40, 41, 79]. In particular, an algorithm can be manipulated into a form that will operate subsequent to RR or RCR hardware reconfiguration. Fault tolerance is thereby achieved by simple reconfigurable hardware in conjunction with correctness-preserving algorithm transformations. We describe here a particularly elegant transformation method that has been developed for SIMD arrays. This approach may eventually prove usable for non-systolic systems as well.

The approach uses linear algebra to describe the algorithms, hardware, program transformations and reconfiguration properties. The descriptions give the patterns of data generation and use. An *index set* describes the shape and activity of the processor array and algorithm. The index set of an array gives the shape of the interconnection primitives. An augmented index set describes an algorithm, complete with the time and location of each computation [41].

This heavy machinery is powerful enough to describe general algorithms, processor arrays and reconfiguration techniques. By use of a six-step transformation process, the algorithm is manipulated to fit the architecture and support the reconfiguration. The steps are:

1. Select possible transformations which are heuristically acceptable.
2. Generate all possible matrices that do not violate the dependency requirements.
3. Find all nonsingular transformations S which solve the matrix equation $SD=PK$. K indicates utilization of primitive interconnections in P, D is the algorithm dependencies, and P is the interconnection primitives.
4. Select the transformation that needs the least number of bands.
5. Map the indices to processors.
6. Schedule the bands, perhaps in lexigraphic order.

Formally, the *processor array* is defined as a tuple (J^{n-1},P) where $J^{n-1} \subset \mathbf{Z}^{n-1}$ is the index set of the array and $P \in \mathbf{Z}^{(n-1)r}$ is a matrix of interconnection primitives ($\mathbf{Z}$ is the set of all integers, and I is the set of nonnegative integers) [79]. General systolic structures can be modeled with this approach, for

example, the 8-neighbor bidirectional connections and triangular arrays. Likewise, an *array algorithm* (A) is defined over an algebraic structure S is a 5 tuple $A=(J^n,C,D,X,Y)$, where J^n is a finite index set, C is triples of (index set, variable, term) where the terms are operations of the structure S. The set of input variables is X. The set D describes dependencies, which may be *input* dependence, *output* dependence and *internal* dependence. Convenient matrix forms of D have been developed. An *execution* is a partial ordering, a *valid execution* only depends on previously computed results and an *execution rule* insists that all computations in C terminate.

This language is sufficiently powerful to describe reconfigurability. An algorithm has the RR property if the first space dimension of the dependency vector (D) is nonnegative. The dependency vector for an algorithm with RCR property is nonnegative in both space dimensions. Thus, it is easy to check if certain hardware implementations of the reconfiguration methods can be used. The approach to algorithm based fault tolerance modifies the algorithm so it will conform to a robust hardware architecture.

Massively Defective Arrays

A novel approach to reconfiguration works on a "massively defective processor array" that consists of simple cells which can (dis)connect from their neighbors. Fault tolerance of the hardware array combines distributed self-configuration algorithms with algorithm transformations. The purpose of self-configuration is to construct a specialized hardware structure from the defect-free computer cells. One implementation of the defective processor array, by Lee [62], makes use of regular replicated structures and switchable interconnects. Usable clusters are identified, organized into a spanning tree, and connected into the desired configuration. The cells can be configured into an irregular graph, as necessary for a general computation.

Lee's work makes realistic assumptions about fault patterns. The size, shape and occurrences of faults determine the patterns of processor clusters. These patterns are described by *percolation theory* [61] This allowed for accurate simulation of the configuration and execution of several filtering algorithms, such as FIR filters.

Algorithms to run on the machine are described as signal flow-graphs, which have very tight synchronization requirements. This is problematic, because the reconfigured hardware may be unable to satisfy the timing requirements. Lee's approach transforms the signal flow-graph into a data flow-graph, since the latter substitutes data dependencies for time dependencies. This is done with domain-specific methods, such as Z-transforms. The I/O relationships between variables form a digraph, and manipulation of this graph is equivalent to manipulation of equations. The time sensitivity of systolic algorithms is thereby avoided by producing a *restricted* dataflow algorithm through systematic transformations of the systolic algorithm. The problems of implementing a full-scale dataflow architecture are also avoided, because the result of these transformation techniques remain within the current hardware technologies.

4.1.4. General Numeric Fault Tolerance

Invariants and checksums are not a panacea. A more powerful and mechanism must be employed for algorithms that do not fit easily into the above framework. One such approach considers the algorithm characteristics, yet is quite general. This is based on graph-language descriptions of the architecture and the program [12, 13, 50, 70, 71, 93]. Allocation and configuration can be described with respect to the

graphs.

Operations upon graphs — isomorphism, path-finding, and flow analysis — have direct utility in augmenting a computation with error detection/correction, as well as mapping to parallel hardware and allocating runtime resources. For example, if the algorithm graph is isomorphic to a subgraph of the architecture, then the hardware can execute the algorithm [50]. A fault can be modeled as the loss of nodes. The system is capable of running the program after loss of k nodes, provided that an isomorphic subgraph always remains after the removal of any k nodes. A difficulty of the graph-theoretic approach is that many of the problems are *NP*-complete, and therefore approximations must be developed. Tractable solutions have been developed for some important special cases, notably *tree* and *lattice* structures.

The most general work builds a graph model of system components, including the computation, the errors, and the error-checks. A check is any combination of hardware and software applied to test the results. Bounds on the data and check nodes are developed subject to "regular" constraints on the cardinality of the elements, and error patterns are derived subject to communication constraints.

Recent work in graph models [12] creates a general tripartite graph from three detailed source of information: error patterns, fault subsets and checks. The initial tripartite graph is simplified into a smaller, computationally tractable, bipartite graph with n inputs and p outputs. The graph is manipulated under the assumption that changes reflect modifications to the entire system, and a *checking graph* is thereby constructed for the simplified problem. Once modified, the bipartite graph is expanded back to a graph of the original problem, under the assumption that the bipartite simplification preserve correctness and is near-optimal. The result is frequently is not optimal because of the bipartite assumption. The only exact bounds have been developed for the non-trivial special case of 2-error detection.

Approximate bounds for *fault detection* can be obtained by 0/1 linear integer programming. The model has $\binom{n}{g}$ subsets corresponding to selection of g checks from n inputs. In the 0/1 formulation a value of x_j is set according to output and neighbor constraints. The $a_{i,j}$ of the constraint matrix are 1 if and only if the cardinality of the intersection between the i^{th} input set with the j^{th} output set does not exceed the number of allowable neighbors. The problem, minimization of $\sum_j x_j$ subject to the constraint $\sum_j a_{i,j} * x_j \geq 1$, can be solved by linear programming to obtain the number of checks. Bounds for fault location are larger, and are not considered here.

The *overhead* of the resulting allocation is bounded by a crude fan-in argument. If p checks are evaluated, only a log-time fan-in tree is required as a lower-bound. An upper bound is more difficult to evaluate because the checks could be arbitrarily complex, though a restricted checking model gives the following bound, where f is the fan-in, g data elements, and p checks are evaluated:

$$time \leq \lceil \log_f(p) \rceil$$

$$\lfloor proc \rfloor \leq \frac{p(g-1)}{f-1} + p$$

Other researchers have also considered algorithm behavior in the recovery algorithms. Since these algorithms depend only on local information, they are not sensitive to central site failures [63, 78, 129]. For example, distributed recovery methods make local recovery decisions based on a graph that represents the "minimal system configuration." The graph is task-dependent, and faults are modeled as

removal of a node from the graph. The recovery process selects a spare node adjacent to the faulty node, and this processor will perform the work of the failed processors. If no spares are available, then the work will migrate to another processor. This processor propagates the error to activate a spare node. However, the algorithm depends on a distinguished "leader" processor. The recovery has to elect a new leader, if a fault disables the distinguished processor. The main problems with the approach are that (a) errors can occur where there are no spares, (b) the local reconfiguration is valid but nevertheless results in an invalid global configuration, or (c) there is interference between multiple simultaneous reconfiguration. Researchers avoid these problems by making the restrictive assumptions that only one error occurs at a time, and that a unique node is responsible for it. They also assume that no faults occur during recovery, and that recovery of multiple faults can be treated as a sequence of single faults [129]. The severity of these problems can be decreased by considering more detailed information about the problem.

Extensions of the method consider specialized structures that have a loop or a tree configuration. This makes explicit use of the algorithm characteristics to bound possible recovery actions, with the desirable consequence of relaxing the restrictions on reconfiguration. The extension for systems where the interconnection topology is a loop extends the initial structure. The basic topology is augmented by additional nodes within a diameter $\leq Z$ to construct a power graph C_{n+k}^{k+1} that is k-FT with respect to the initial graph C_n. For *tree systems*, one can develop necessary and sufficient conditions for fault tolerance to be *p-step k-FT* with respect to *R*. A *k-FT* tree is constructed from the basic T_b graph by assigning spare nodes to *k* spare levels and interconnecting these spares to the root. Thus, the specialized characteristics are key to improved fault tolerance.

4.1.5. Adding Redundancy — Checksums

Often the invariants are not strong enough to build a fault tolerant computation. In this case the algebraic invariants can be added to a system through data redundancy, particularly for numeric algorithms. For example, checksums operate with less than 100% additional hardware, yet provide close to 100% fault coverage. These data level redundancies, such as include parity or checksums, are maintained by memory accesses and arithmetic operations. Data aggregates (such as arrays) can also be made redundant by use of row or column checksums. The difficulty is to find an encoding method that can be computed quickly yet remains invariant over the operators of the domain. The problem has been successfully tackled for many machines, including processor arrays [83], hypercubes [17] and volatile memories.

Three kinds of checksums have been developed for numerical problems such as matrix-matrix, convolution, filters and linear system solvers [1, 83]. The distinction between the many different kinds of checksums is illustrated by the example of LU decomposition. Three ways to solve the problem are:

- *Row and column checksums*, which can detect but not correct errors in matrix vector multiplication or LU-decomposition errors.
- *Weighted checksums*, which provide a low-cost solution to the above limitations, but suffer from the introduction of roundoff errors.
- *Weighted average checksums* (WAC) which are preserved by the operations of matrix addition, multiplication, LU-decomposition, transposition and scalar product.

The reliability of these methods has been extensively analyzed under the assumption of a constant number of processors [14]. Comparisons were made for both theoretical and empirical results. The most reliable method was the weighted average checksum (WAC). In contrast, the triplicated approach was *less reliable* than the non-redundant algorithm, since it must use three smaller problem partitions, and hence has a longer runtime. A refined measure shows the WAC algorithm is roughly *twice* as reliable as TMR. This result is valid for special-purpose units that run in real time without interference, using fixed-size hardware arrays. However, the TMR method is slightly better according to the metric of mean time to first failure (MTTF).

Checksums have been designed for high-level computations as well. Convolution algorithms, for example, are made fault tolerant with systematic encodings [97] that "leave data symbols in an unaltered form after manipulation by polynomial multiplication." The advantage of the approach is simplicity of the resulting hardware system, although the encoding is highly domain-dependent (see [97] for the mathematical detail). This allows the fundamental convolution algorithm to execute unmodified, yet protects against errors. A coding circuit computes parity symbols concurrently with the data manipulations. A self-checking comparator then checks the result of the computational and parity steps.

4.2. Search Algorithms

We now turn our attention to search algorithms. The search problem is to select a specified element from a set, where the choice is based either on an exact or an heuristic match. Researchers have considered fault tolerance of both sequential and parallel search algorithms. Results consist of many different algorithms, each algorithm designed under a different fault model. The variety in fault models is a prominent reason for the many different results. A general fault model is built on Byzantine failures. Given primitive operations that can deceptively lie up to E times, the research result is an algorithm and complexity bounds for ordered sequential search [98]. Architecture-specific models, on the other hand, correspond to the vulnerabilities of specific hardware systems. One such model uses a network of fail-stop processors that never lie, but may be infinitely postponed [11, 54]. Alternatively, a system crash is a serious fault that may corrupt the system by the failure to complete updates to the the search structures [22, 42, 117]. Fault tolerance requires restoration of the data structures to a consistent state. A recent hardware development is the parallel associative memory, where the dominant faults are loss of a block of memory, or random bit changes. We now discuss the specific fault models and the algorithms developed for them.

4.2.1. Binary Search with Faulty Comparisons

Work by Rivest [98] solves a search problem under the theoretical model of a sequence of primitive operations subject to faults. The problem is to select an element (or neighborhood) from an ordered set (or function) by use of a binary search procedure. The primitive operation is to ask "yes/no" membership questions, for example, "is the value less than x." The fault model allows up to E wrong answers, which may attempt to deceive the algorithm. Since questions are posed sequentially, the measure of efficiency is the number of questions, rather than the complexity of the questions or the time needed to answer them. Since this work is somewhat theoretical, it should be mentioned that other researchers have considered more general unreliable components [43, 44] and communication over faulty channels [119].

In Rivest's work the redundancy is the additional queries that must be posed to compensate for the incorrect answers. A trivial algorithm would repeat each question $2E+1$ times (if E errors can occur), and could be based upon the lower bound of $\log_2 n$ questions posed by bisection (binary search) with error-free information for n elements. A surprisingly efficient "adversary answering strategy" uses only $E \log_2 \log_2 n + E \log_2 E$ more comparisons than the bisection algorithm. The strategy computes a weight for each possible element according to the number of erroneous answers that may still be supplied. At each step the algorithm picks the next question ("is x ε T") such that the "yes" and "no" weights are closest to being equal. The significance of this result is that establishes a precise bound, which fortunately is quite small, on the number of additional comparisons. This can have practical benefits when questions are expensive. This work could be extended in several ways, including design of pattern recognizers that operate under data, program, or component faults. The most serious limitations with this approach are the requirement of an ordered domain, the sequential execution of the queries, and the assumption that all questions are equally expensive. These restrictions are relaxed in the engineering approaches described below.

4.2.2. Heuristic Search with Fail-Stop Processors

Fail-stop systems have the property of partial-correctness. They assume that a component either operates correctly or does not operate at all. Thus, all available data can be completely trusted, since failures are manifest as lost computational and storage capacity. When the broken processors stop providing service, the computation and data in the processor are lost. Fault tolerance routines must efficiently complete the computation despite possible loss of programs and information. Several methods have been developed to address the possible loss of partially complete computations. In particular, when a stopped processor is found, the computations and data it had been working on must be reassigned. Alternatively, some robust parallel computations produce a satisfactory answer despite loss of some processors' information. This is possible in some search problems because of redundancy in the search spaces, as will be described shortly.

One difficulty with fail-stop processors is locating the faulty PEs. A centralized technique can periodically poll the other processors. If a processor does not receive an answer to a periodic "are you still working" message, then the requesting processor finds a new subcontractor to perform the work. Unfortunately, it has been observed that some subtasks take a long time to complete execution [11]. This presents the complication that it is impossible to distinguish long-running tasks from broken processors. The use of watchdog timers establishes an ad-hoc time-bound on each subtask, and timeout indicates that the process failed.

This simple technique suffers from the inefficiency that subprocesses created by the crashed processor no longer have a parent to receive the answer. This is the general "orphan" problem in distributed computation, where task T_1 creates task T_2, which in turn creates T_3. If T_2 crashes then the task T_3 cannot return an answer. All subprocesses of the crashed processor become orphans, since there is no parent waiting for their reply. Efficiency can be improved by keeping the orphans, rather than killing them. The intermediate results may be salvaged by use of specialized runtime structures, such as the dynamic call tree of subroutine invocation [64]. A "twin task" is allocated to inherit the offspring of a crashed processor, and the necessary parent/child linkages are retained through grandparent pointers.

The research misses several important issues. In particular, it ignores the efficiency of the parallel implementation. For example, the analysis does not mention the branching factor. Moreover, the programmer is required to state parallelism explicitly. Thus the algorithm properties are not fully exploited. These issues have been addressed by other researchers. Specifically the structure of the search space can improve fault tolerance and efficiency. For example, if a search space is redundant there can be several valid solutions to problem. Each parallel processor can simultaneously explore a different plausible solution. The fault tolerant algorithm can entirely ignore the failed processor, since the other processors can still compute an answer. Recovery of the lost work is therefore not required.

The success of this approach requires that answer-producing computations be uniformly distributed throughout the computing device. A technique to ensure that the answers adhere to this property is called *randomization*, partitioning the search space so each partition is expected to have an answer [54]. Each partition can be processed using only local information, making it unnecessary to maintain global data structures! The exact randomization depends upon cutoff functions of the particular problem instance, since these determine how deep a search will proceed before it is abandoned as futile. The initial distribution of answers is also important. If there are good cutoff functions and uniformly distributed answers, then randomized search trees [103] provide a method to select the next successor. In these cases, the randomized search is as efficient as depth first search, because the cutoff functions prevent long and fruitless searches. Randomization also helps when answers are scarce or when the cutoff function does not prune away bad paths. In both cases, all answers must be found, because the cost of missing a solution can be large. In this situation, selection of the next node to evaluate is made with local information. It is efficient because there is no global information, there are neither global checkpoints or access delays [54]. However, this presumes that faults are only omissions, not incorrect output. Also, there cannot be global damage by the faults.

4.2.3. Ordered Search with Dictionary Machines

Search has been approached by specialized hardware. One specialized task is the dictionary task, which is composed of a sequence of query and update operations. The query operator is quite simple, generally the retrieval of the record that corresponds to a given value. Many architectures have been proposed for this task, including large associative memories. The primary approach to fault tolerance is to maintain multiple copies of the data. However, recent work explores the idea of retrieval based on hash codes for the special case of imprecisely defined access (as in vision recognition tasks, for example). In both cases, it is important to note that data retrieval is highly data dependent. This problem cannot be solved efficiently by systolic arrays [21] or their fault-tolerant variants. Therefore, the fault tolerance methods for systolic processing cannot be directly applied to the dictionary task.

Implementations of dictionary machines include tree-based architecture [15] and hypercube machines [87, 106]. A particular technique designs fault tolerance directly into the algorithms by use of two binary tree machines (each a "half-tree"), each of which stores a complete copy of the data. A pair of redundant root processors, one for each half-tree, issues commands to both trees. Update operations execute in both halves. Queries return an answer to the respective roots, which vote and signal an error if there is a disagreement [4]. Error correction currently requires reloading the defective tree from the operational tree. This simple scheme has several defects.

A major problem stems from use of a "compress" operation that rebalances each half-tree after every update to prevent a skewed tree from growing within the half-tree. As a result, an entire subtree can become polluted by the "compress" operation due to propagation of errors between processors in a subtree. The data structures do not support local correctability within a region or even within a half-tree. Since errors must be corrected by reloading the faulty subtree from the fault-free subtree, the machine only tolerates 1 fault between two "query" operations. Significant performance improvements would be achieved if the software could isolate the area around the fault and then repair the damage selectively. Parallel k-fault tolerant data structures would be an appropriate way to improve this fault tolerant dictionary machine. Such structures will be described for sequential machines in section 4.2.4 . Even given these difficulties, the cost for fault tolerance in the basic approach is only about 100% additional hardware, which is better than the 3X replication necessary with a triply modular redundant (TMR) approach. The effectiveness of their approach has not been compared with error correcting codes, perhaps because such codes are vulnerable to transient processor faults.

A second approach to the problem considers data retrieval without update [28], where faults occur as errors in the input key and failures in some portion of the memory. Fault-tolerant retrieval is formulated as a constraint optimization problem, similar to a neural network. These retrieval networks are constructed as sets of associations. For image retrieval, recall is by presentation of a stimulus vector, with auto-associative evaluation based on the generalized inverse of the input. This is effective for retrieval of unknown objects (such as images) embedded in heavy noise. For text retrieval, the use of hashing methods has the property that memory corruption results in only a modest decrease in retrieval performance. This is largely because the memory is sparse. However, the authors have not reported details of the hash functions, and their work does not consider large databases. It seems that use of well-protected error-correcting codes would be a more direct way to solve the problem.

4.2.4. Crash Resistance — Ordered Storage Structures

Robust data structures — such as crash-resilient binary trees [117] and AVL trees [36] — improve the reliability of sequential systems. Several principles form the foundation for this work. Known relationships (such as orderings) between data elements can be exploited for fault detection. Additional links are added [22, 99, 115, 114], to make the structure crash-resistant. Algorithms operate on the redundant data structures to perform multiple traversals, and an error is detected if the traversals indicate different data orderings. The error can then be corrected by voting among the multiple traversals to determine the majority result. Error detectable and correctable search trees require only 1 more probe per search than the underlying balancing technique [82].

As an example of fault tolerant data structures, the *k-spiral list* achieves a theoretical lower bound on correctability by use of a data portion and k links. K-1 links point forward to different successor elements, and one points back to a predecessor. A link can be changed atomically, yet several changes are required to insert or delete an element. If the processor crashes during these updates, it is necessary to recover the error-free *before* instance from the errorful *after* instance. The correctability of such a structure is described formally; if c_1 components are changed in the new node, c_2 other changes are made, p pointers are added to the new node, and r errors occur:

> "For an instance in main storage, recovery by an r-correction routine from a crash during an insertion operation can be guaranteed iff (a) $c_1 + c_2 \leq 2r + 1$ or (b) $c_2 \leq 2r + 1$".

Since this bound cannot always be achieved, a weaker condition is needed. For example, in the chained and threaded binary tree, deletion first uses a sequence of rotate operations to move the node into a leaf position, and then deletes the leaf [117]. A partially complete deletion cannot be "undone" within the tight bounds given above. Fortunately, a relaxed correctability theorem produces a satisfactory algorithm. Instead of producing the *before* instance, it is satisfactory to make a new instance that will produce the right answer when the crashed operation is reapplied. Rotations conform to this definition because they retain the inorder sequence of the tree. Moreover, this correctable structure is resilient to a greater number of errors, under the assumption that only one error occurs within a bounded number of components [116]. Thus, fault tolerant algorithm design modifies the original algorithm so it functions correctly despite faults.

These ideas have recently been extended to searching a B-tree where several indices have been corrupted [42]. However, robust parallel data structures have not yet been described in the literature. Nevertheless, the above approach should be parallelizable for dictionary machines or text-search applications (such as [106]).

A second extension is to create *resilient execution structures* to maintain execution information about the dynamic structure of a parallel system, since they can be used to recover from execution faults [65]. Although these structures do not depend upon the algorithm properties, the ability to construct these representations depends upon the programming paradigm. In particular, under the functional paradigm, programs do not destructively modify data, and they are furthermore deterministic in the sense that the same functions and parameters determine a unique function output. Therefore, a dynamic graph of parameters represents a state which can be monitored by simple numbering techniques. This resilient execution structure can be manipulated into a consistent state without extensive "undo" operations.

In summary, the method of algorithm-based fault tolerance is motivated by the working examples described above, and additionally by problems with the other approaches. Neverthless, some readers may prefer the traditional general-purpose fault-tolerance, due to the greater difficulty of designing for the algorithmic method. However, there are many specialized cases where the general-purpose cannot provide the necessary performance.

Despite the attraction of general-purpose methods they can suffer from performance degradation, attributable to the overhead of redundancy management methods such as synchronization and voting [59]. This can even introduce real-time errors. Non-deterministic choices in NMR systems can cause problems with voting methods unless special attention ensures that all replicants make same choice [85].

The overhead of checkpoint methods is a serious problem for interactive or real-time tasks. It has been observed that it takes 19 seconds to checkpoint a system of 1000 ethernet-connected microprocessors, with an additional 20 seconds to recover [113]. This is satisfactory for a general noninteractive fault-tolerant environment, but it is too slow for high-availability systems. Fault detection uses redundant subnetworks, with signature-voting to detect inconsistencies, and checkpoints for recovery. The major problem with the approach is the need to quiese each subsystem prior to voting, so the error-free states will be identical. This also demands deterministic subsystems. Because it takes 20 seconds to freeze and checkpoint the network, these operations can only be performed every half hour. Hence the system may

lose up to 30 minutes of computation if it must revert to a checkpoint! The 20 second delays and the 30 minute work loss should be improved. Algorithm-specific approaches are one promising means to solve these problems.

5. Conclusion

This paper has presented a variety of approaches of fault tolerance, with an eye towards massively parallel computer systems. The method is not general, and has thus far been applied only to specialized problems. It results in significant performance advantages when it is applicable. Research is underway to improve the generality of the approach, though the three-pronged approach of predicates, program analysis and program transformations.

This paper has described important characteristics of algorithms and has cited research where these properties were utilized to reduce the cost fault tolerance. The method of using characteristics of algorithms, applications and architectures has been studied through a dozen examples. These are summarized in figure 5-1, which shows the invariants, the costs and the method by which they work. The key point of the figure is that algorithm invariants are exploited to achieve low-cost fault tolerance in many cases. This is done by finding an invariant property and building fault tolerance around the property. One such invariant is the ordering of the data elements. As shown in the first line of the figure ("faulty comparisons"), the method of *adversary answering* exploits the characteristic of "yes/no" membership questions. Likewise, the sixth line ("dictionary machines") shows use of a similar invariant, where the fault-tolerance method is to vote between the two subtrees.

The primary difficulty is finding the invariants. Although some "generic" methods are known, such as checkpoints and checksums, in general a painstaking analysis must be performed to determine these properties. There as an opportunity for major technological advances in the form of automated analysis tools. These might include dataflow compilers combined with token-managers, or dynamic programming formulations that determine the cheapest way to retain the necessary amount of redundancy. An area of particular interest is parallel processing of AI systems on MPPs. Little work has been done in the area, but the power of AI, when delivered on an MPP, suggests this will become more commonplace in the coming years. It is therefore appropriate to consider the application characteristics in development of fault tolerance for these systems.

EXAMPLE	CLASS	INVARIANTS	COST/PERFORMANCE	METHOD	CHARACTERISTIC	REFERENCE
Faulty comparison	Algorithm/ program	Ordered set or program	Eloglog(n)+O(Elog(E))+log(n) continuous function	Adversary Answering	Yes/No questions on set membership.	[98]
Search/ Crystal	Algorithm/ OS	Atomicity of query: state update iff query succeeds	Depends on query cost, delay, and failure rate	Restart sub-task if crash	Search problems, nondeterministic	[11]
Search/ randomized backtrack	Algorithm/ program	Distribution of solutions, de-composable search	Depends on cutoff functions and distribution of the answers	Use of randomized search trees	Save less state, keep list of unexamined ids	[103]
Crash recoverable structures	Algorithm/ data struct and program	Numbers of new and old links: c1+c2<-2r+1	Constant cost: may also improve performance by faster access	Redundant forward links	Search problems of keyed data; data structures	[22] [117]
B-Tree search of indices	Algorithm/ data struct	Relationships between index elements	Low storage overhead, no data structure change	Check suspect data elements	Search problems of indexed data	[42]
Dictionary machines	Algorithm/ hardware & program	Inorder within each tree; voting on answer	100% hardware, whereas triple modular redundant would be alot more.	Vote on the 2 half-trees	Data structures have constant properties	[4]
Bitonic sorting, UDD & RDC	Reconfigure/ software	Communication flow is maintained	UDD and RDC are best of any known distributed alg	Redistribute data	Processor array, MIMD hardware	[122]
Array algorithms	Reconfigure and remap	Flow of data, space and time of execution	Guaranteed performance result at large development cost	Remap alg to RR/RCR props	Formal; processor array & algs	[40]
Manufacture of large computing structures.	Hardware/ distributed algorithms	Topology and resource allocation accord-ing to algorithm	Aprox. 70% hardware util-ation. Depends on the defect rate	Builds from a spanning tree to map a def-ective to a usable array.	Signal processing tasks	[61]
Matrix and vector algorithms.	Algorithm and data.	Algebraic relation-ship between data and checking bits: checksums.	Problem dependent. Best for some problems and metrics.	Encode data & validate at retrieval.	Algebraic tasks, i.e. defined on group or ring.	[128]
Matrix algorithms.	Algorithm	Property of matrix method: orthogonality.	Can be executed concurrently with the algorithm.	Concurrent checking of ortho prop.	Restricted forms of matric algebra.	[17]
Signal processing algorithms.	Algorithm	Property of the domain: Parseval's theorem.	Check inexpensively between iterations.	Check for violation of the invariant.	Most signal processing tasks.	[4]
Conjugate gradient theorem	Algorithm	Useful for matrix solution of linear system, but costly	Corollary is inexpensive to test.	Check for violation of invar.	Uses data redundancy to preserve info.	[6]
Partial differential equations	Algorithm	Local computation, small sensitivity to isolated errors.	Small cost to rediscretize the problem.	Bypass recovery.	Numerical algorithm	[88]
General spawning of subtasks.	Programming methodology & OS	Task structure	Low cost -- retains the work of subprocesses through grandparent links, called functional checkpointing.	Slice recovery w/ level stamps	Maintain call structure	[66]
Graph algorithms.	Theoretical	Graph adjacencies	High development cost, low execution cost	Maintain and check graph.	Built upon the graph models.	[13]

Figure 5-1: Examples of Invariants in use for Fault Tolerance

References

[1] Abraham, J. A.
Fault Tolerance Techniques for Highly Parallel Signal Processing Architectures.
SPIE Highly Parallel Signal Processing Architectures, 614:49-65, 1986.

[2] Anderson, T., Lee, P. A.
Fault Tolerance -- Principles and Practice.
Prentice Hall, USA, 1981.

[3] Anderson, T.
Resilient Computing Systems.
Wiley Interscience, USA, 1985.

[4] Annapareddy, Narasimha R.
Algorithm-Based Fault Detection Techniques for Specific Applications.
Technical Report, University of Illinois, Department of Electrical Engineering, June, 1987.
SRC Technical Report No. T87054, Contract No. 86-12-109.

[5] Avizienis, A.
The N-Version Approach to Fault-Tolerant Computing.
IEEE Transactions on Software Engineering, SE-11(12)December 1985.

[6] Aykanat C., F. Ozguner.
A Concurrent Error Detecting Conjugate Gradient Algorithm on a Hypercube Multiprocessor.
In *International Symposium on Fault-Tolerant Computing*. 1987.

[7] Babaoglu, Ozalp, and Rogerio Drummond.
Streets of Byzantium: Network Architectures for Fast Reliable Broadcasts.
IEEE Transactions on Software Engineering, SE-11(6)1985.

[8] Babb, R. G., D. DiNucci.
Design and Implementation of Parallel Programs with Large-Grain Data Flow.
The Characteristics of Parallel Algorithms.
MIT Press, 1987.

[9] Babb, C.
Tracing Tokens for Fault Tolerance in Large Grain Dataflow.
Private Communication.
July 1987

[10] Baiardi, F., and M. Vanneschi.
Forward Recovery to Enhance Parallelism by Temporary Violation of Invariants.
In *Fifth Symposium on Reliability in Distributed Software and Database Systems*. IEEE, 1986.

[11] Bal, H. E., R. van Renesse, A. Tanenbaum.
A Distributed, Parallel, Fault Tolerant Computing System.
Technical Report, Vrije Universiteit Amsterdam, October 1985.
IR-104.

[12] Banerjee, P.
A Theory of Algorithm-Based Fault Tolerance in Array Processing Systems.
University of Illinois, Urbana, Illinois, 1985.

[13] Banerjee, P., Abraham, J. A.
Bounds on Algorithm-Based Fault Tolerance in Multiple Processor Systems.
IEEE Transactions on Computers, C-35(4):296-306, April 1986.

[14] Banerjee, P., Abraham, J. A.
A Probabilistic Model of Algorithm-Based Fault Tolerance in Array Processors for Real-Time Systems.
In *Proceedings Real-Time Systems Symposium*, pages 72-78. IEEE, December 1986.

[15] Reddy, Narasimha A. L., P. Banerjee.
A Fault-Tolerant Dictionary Machine.
In *Third International Conference on Data Engineering*, pages 104-109. IEEE, 1987.

[16] Reddy, A. L. Narasimha, P. Banerjee.
Algorithm-Based Concurrent Error Detection in Array Processors for Signal Processing Applications.
Technical Report, University of Illinois, Coordinated Science Lab, 1988.

[17] Banerjee, P., Rahmeh, J., Stunkel, C., Nair, S., Roy, K., Abraham, J.
An Evaluation of System-Level Fault Tolerance on the Intel Hypercube Multiprocessor.
To appear in IEEE Transactions on Computers.

[18] Barbara, D., H. Garcia-Molina.
The Vulnerability of Voting Mechanisms.
In *Symposium on Reliability in Distributed Software and Database Systems*. 1985.

[19] Best, E., and Cristian, F.
Comments on "Self-Stabilizing Programs: The Fault-Tolerant Capability of Self-Checking Programs".
IEEE Transactions on Computers, C-34(1)1985.

[20] Birman, Kenneth P., Joseph, Thomas A., Raeuchle, Thomas, and Abbadi, Amr El.
Implementing Fault-Tolerant Distributed Objects.
IEEE Transactions on Software Engineering, SE-11(6)1986.
Reprinted in "4th Symposium on Reliability in Distributed Software and Databases", page 124-133.

[21] Bisiani, R., A. Forin.
Multilanguage Parallel Programming.
In *International Conference on Parallel Processing*. 1987.

[22] J.P. Black, D.J. Taylor, D. E. Morgan.
A Compendium of Robust Data Structures.
In *International Symposium on Fault-Tolerant Computing*. 1981.

[23] Bokhari, S. H.
Assignment Problems in Parallel and Distributed Computing.
Kluwer Academic Publishers, Norwell, Mass., 1987.

[24] Briatico, D., Ciuffoletti, A., Simoncini, L.
A Distributed Domino-Effect Free Recovery Algorithm.
In *Fourth Symposium on Reliability in Distributed Software and Database Systems*. 1985.

[25] Brown, G. M., M. G. Gouda, C-L. Wu.
A Self-Stabilizing Token System.
In *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences*, pages 218-223. 1987.

[26] Campbell, Roy H., and Brian Randell.
Error Recovery in Asynchronous Systems.
IEEE Transactions on Software Engineering, SE-12(8)1986.

[27] Carter, W. C.
Hardware fault tolerance.
Resilient Computing Systems.
John Wiley & Sons, 1985.

[28] Char, J. M., V. Cherkassky, H. Wechsler, G. Zimmerman.
Distributed and Fault-Tolerant Computation for Retrieval Tasks Using Distributed Associative Memories.
IEEE Transactions on Computers, C-37(4):484-490, April 1988.

[29] Chor, B., B. A. Coan.
A Simple and Efficient Probabilistic Byzantine Agreement Algorithm.
In *Symposium on Reliability in Distributed Software and Database Systems.* 1985.

[30] Chung, F. R. K., Leighton, F. T., Rosenberg, A. L.
Diogenes: A Methodology for Designing Fault-Tolerant VLSI Processor Arrays.
April, 1983.
MIT VLSI Memo No. 83-142.

[31] Clarke, E. M., Christos N. Nikolaou.
Distributed Reconfiguration Strategies for Fault-Tolerant Multiprocessor Systems.
IEEE Transactions on Computers, C-35(8):771-784, August 1982.

[32] Contessa, A.
An approach to fault tolerance and error recovery in a parallel graph reduction machine: MaRS - a case study.
In *ACM Computer Architecture News; Vol. 16, No. 3.* June 1988.

[33] Cristian, Flaviu.
A Rigorous Approach to Fault-Tolerant Programming.
IEEE Transactions on Software Engineering, 11(1)1985.

[34] Cristian, F.
Reasoning about Programs with Exceptions.
In *International Symposium on Fault-Tolerant Computing.* 1983.

[35] Cristian, F., H. Aghili, R. Strong, D. Dolev.
Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement.
In *International Symposium on Fault-Tolerant Computing.* 1985.

[36] Davis, I. J.
A Locally Correctable AVL Tree.
In *International Symposium on Fault-Tolerant Computing.* 1987.

[37] Dixon, Graeme, Santosh Shrivastava.
Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems.* 1987.

[38] Echtle, K.
Fault Masking and Sequence Agreement by a Voting Protocol with Low Message Number.
In *Symposium on Reliability in Distributed Software and Database Systems.* 1987.
Algorithm for Concurrency and Synchronization.

[39] Ezhilchelvan, P. D.
Early Stopping Algorithms for Distributed Agreement under Fail-Stop, Omission, and Timing Fault Types.
In *Symposium on Reliability in Distributed Software and Database Systems.* 1987.

[40] Fortes, J. A. B., and C. S. Raghavendra.
Dynamically Reconfigurable Fault-Tolerant Array Processors.
In *International Symposium on Fault Tolerant Computing*. 1984.

[41] Fortes, J. A. B., Raghavendra, C. S.
Gracefully Degradable Processor Arrays.
IEEE Transactions on Computers, C-34(11):1033-1044, November 1985.

[42] Fujimura, K., P. Jalote.
Robust Search Methods for B-Trees.
In *International Symposium on Fault Tolerant Computing*. 1988.

[43] Gacs, P.
Reliable Computation with Cellular Automata.
Journal of Computer and System Sciences, 32(1):15-78, 1986.

[44] Gacs, P.
Self-correcting Two-Dimensional Arrays.
Technical Report, Boston University, September 5, 1987.
BUCS Tech Report #87002.

[45] Gargano, L.
Fault-Tolerant Minimal Broadcast Networks with a Logarithmic Number of Faults.
Technical Report, Columbia University Computer Science, January 1988.

[46] Di Giandomenico, F., M.L. Guidotto, F. Grandoni, L. Simoncini.
A Gracefully Degradable Algorithm for Byzantine Agreement.
In *Symposium on Reliability in Distributed Software and Database Systems*. 1987.

[47] Harper, R. E., J. H. Lala, J. J. Deyst.
Fault Tolerant Parallel Processor Architecture Overview.
Technical Report, Charles Start Draper Laboratory, November 10, 1987.
This paper has been "cleared" through the authors' affiliations.

[48] Hassan, A. S. M., Agarwal, V. K.
A Fault-Tolerant Modular Architecture for Binary Trees.
IEEE Transactions on Computers, C-35(4):356-361, April 1986.

[49] Hastad, J., Tom Leighton, Mark Newman.
Reconfiguring a Hypercube in the Presence of Faults.
In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 274-284. ACM, 1987.

[50] Hayes, J. P.
A Graph Model for Fault-Tolerant Computing Systems.
IEEE Transactions on Computers, TC-25(9):875-884, September, 1976.

[51] Hecht, H., M. Hecht .
Use of Fault Trees for the Design of Recovery Blocks.
In *International Symposium on Fault-Tolerant Computing*. 1982.

[52] Huang, Kuang-Hua and Jacob A. Abraham.
Fault-Tolerant Algorithms and Their Application to Solving LaPlace Equations.
In *International Conference on Parallel Processing*. 1984.

[53] Jalote, Pankaj, Roy H. Campbell.
Fault Tolerance Using Communicating Sequential Processes.
In *International Symposium on Fault Tolerant Computing*. 1984.

[54] Janakiran, V. K., D. P. Agrawal, R. Mehrotra.
Randomized Parallel Algorithms for Prolog Programs and Backtrackig Applications.
In *International Conference on Parallel Processing*. 1987.

[55] Johnson D. B., W. Zwaenepoel.
Sender-Based Message Logging.
In *International Symposium on Fault-Tolerant Computing*. 1987.

[56] Kar, Gautam, Nikolaou, Christos N.
Assigning Processes to Processors: a Fault-Tolerant Approach.
In *International Symposium on Fault Tolerant Computing*. 1984.

[57] Hutchinson, J., C. Koch, J. Luo, C. Mead.
Computing Motion Using Analog and Binary Resistive Networks.
IEEE Computer, 21(3):52-63, 1988.

[58] Kopetz, H., W. Merker.
The Architecture of MARS.
In *International Symposium on Fault-Tolerant Computing*. 1985.

[59] Krishna, C.M., Kang Shin.
Synchronization and Fault-Masking in Redundant Real-Time Systems.
In *International Symposium on Fault Tolerant Computing*. 1984.

[60] Krishna, C. M., Kang G. Shin.
On Scheduling Tasks With a Quick Recovery from Failure.
In *International Symposium on Fault Tolerant Computing*. 1985.

[61] Lee, Myoung Sun.
Self-Configuration on the Massively-Defective Cellular Array.
University of Michigan, Michigan, 1986.

[62] Lee, Myong Sung, G. Frieder.
Massively Fault-tolerant Cellular Array.
In *International Conference on Parallel Processing*. 1986.

[63] Li, H. F., D. Pao, R. Jayakumar.
Dynamic Reconfiguration for Fault-Tolerant Systolic Arrays.
In *Parallel Computation*. 1987.

[64] Lin, Frank C. J., R. M. Keller.
Distributed Recovery in Applicative Systems.
In *International Conference on Parallel Processing*. 1986.

[65] Lin, Frank C. H., Robert M. Keller.
Gradient Model: A Demand-Driven Load Balancing Scheme.
In *Sixth International Conference on Distributed Computing Systems*. 1986.

[66] Lin, Frank C.H., and Robert M. Keller.
The Gradient Model Load Balancing Method.
IEEE Transactions on Software Engineering, SE-13(1)1987.

[67] Lipovski, G. J., M. Malek.
Parallel Computing. Theory and Comparisons.
John Wiley and Sons, U.S.A., 1987.

[68] Lowrie, M. B., Fuchs, W. K.
Reconfigurable Tree Architectures Using Subtree Oriented Fault Tolerance.
IEEE Transactions on Computers, C-36(10):1172-1182, October 1987.

[69] Luk, F. T., H. Park.
An Analysis of Algorithm-Based Fault Tolerance Techniques.
Journal of Parallel and Distributed Computing, 5:172-184, 1988.

[70] Maehle, E., Moritzen, K. and K. Wirl.
A Graph Model for Diagnosis and Reconfiguration and its Application to a Fault-Tolerant Multiprocessor System.
In *International Symposium on Fault-Tolerant Computing*. 1986.

[71] Mahmood, Aamer, and McCluskey, E. J.
Concurrent Error Detection Using Watchdog Processors -- A Survey.
IEEE Transactions on Computers, 37(2)1988.

[72] McMillian, Bruce M., N. M. Ni, A. H. Esfahanian.
Design of a Reliable Parallel Environment Through Constraint Predicate Specification.
Technical Report, Michigan State University, College of Engineering, December 1987.

[73] Mead, C. A., Conway, L. A.
Introduction to VLSI Systems.
Addison-Wesley, Philippines, 1980.

[74] Metze. G, A. Mili.
Self-Checking Programs: An Axiomatisation of Program Validation by Executable Assertions.
In *International Symposium on Fault-Tolerant Computing*. 1981.

[75] Mili, Ali.
Self-Stabilizing Programs: The Fault-Tolerant Capability of Self-Checking Programs.
IEEE Transactions on Computers, C-31(7)1982.

[76] Mili, Ali.
Towards a Theory of Forward Error Recovery.
IEEE Transactions on Software Engineering, SE-11(8)1985.

[77] Mills, R.
An Algorithmic Taxonomy of Production System Machines.
Technical Report, Columbia University, Computer Science Department, 1987.
CUCS-340-88.

[78] Mohan, Chilukuri, and Larry Wittie.
Local Reconfiguration of Management Trees in Large Networks.
In *Distributed Computing Systems*. 1985.

[79] Moldovan, D. I., Fortes, J. A. B.
Partitioning an Mapping Algorithms into Fixed Size Systolic Arrays.
IEEE Transactions on Computers, C-35(1):1-12, January 1986.

[80] Morenos, Jaime H.
Replication and Pipelining in Multiple-Instance Algorithms.
In *International Conference on Parallel Processing*. 1986.

[81] Morris, Derek S.
A Fault-Tolerant Software Construction Technique Based on Redundant Abstract Data Types.
1987.

[82] Munro, J. Ian.
Fault Tolerance and Storage Reduction in Binary Search Trees.
Information and Control, 62:210-218, 1984.

[83] Nair, V.S.S., Abraham, J. A.
Average Checksum Codes for Fault-Tolerant Matrix Operations on Processor Arrays.
In *2nd International Conference on Supercomputing*. IEEE, May 4-7, 1987.

[84] Najjar, W., J. L. Gaudiot.
Distributed Fault-Tolerance in Data Driven Architectures.
In *International Conference on Supercomputing*. 1987.

[85] Natarajar, N., and Jian Tang.
Synchronization of Redundant Computation in a Distributed System.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems*. IEEE, 1987.

[86] Nikolaou, C., Kar, G.
Allocation and Reallocation of Processes in Distributed Computer Systems.
Current Advances in Distributed Computing.
Computer Science Press, 1987.

[87] Omondi, A. R., J. D. Brock.
Implementing a Dictionary on Hypercube Machines.
In *International Conference on Parallel Processing*. 1987.

[88] Osawa, G, T. Kawai, H. Aiso.
Fault Tolerant Scheme on Partial Differential Equations.
In *International Conference on Parallel Processing*. 1986.

[89] Patel, J. H., L.Y. Fung.
Multiplier and Divider Arrays with Concurrent Error Detection.
In *International Symposium on Fault-Tolerant Computing*. 1982.

[90] Perry, K. J.
Randomized Byzantine Agreement.
In *Symposium on Reliability in Distributed Software and Database Systems*. 1985.

[91] Pippenger, N.
Reliable Computation by Formulas in the Presence of Noise.
IEEE Transactions on Information Theory, IT-34(2)1988.

[92] Pradhan, D. K.
Fault Tolerant Computing: Theory and Techniques, Volume 2.
Prentice-Hall, Englewood Cliffs, N.J., 1986.

[93] Pradhan, D. K., and K. Matsui.
A Graph Theoretic Solution to Load Leveling and Fault-Recovery in Distributed Systems.
In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*. 1981.

[94] Ralston, T., P. Rabinowitz.
A First Course in Numerical Analysis.
McGraw-Hill Book Company, USA, 1978.

[95] Raynal, M., (D. Beeson *translator*.
Algorithms for Mutual Exclusion.
MIT Press, Cambridge, Mass., 1986.

[96] Reardon, J.
Private Communication.
Telephone.
April 18, 1988

[97] Redinbo, G.
Fault-Tolerant Convolution Using Real Systematic Cyclic Codes.
In *International Symposium on Fault-Tolerant Computing*. 1987.

[98] Rivest, R. L., A. R. Meyer, D. J. Kleitman, K. Winklmann.
Coping with Errors in Binary Search Procedures.
Journal of Computer and System Sciences, 20:396-404, 1980.

[99] Seth, S.C., R. Muralidhar.
Analysis and Design of Robust Data Structures.
In *International Symposium on Fault-Tolerant Computing*. 1985.

[100] Lilien, L., S. M. Shatz.
Software Redistribution in Distributed Mission-Oriented Systems after Site Crashes and Network Partitioning.
In *International Conference on Supercomputing*. 1987.

[101] Shing, K. G., T-H Lin, Y-H Lee.
Optimal Checkpointing of Real-Time Tasks.
In *Fifth Symposium on Reliability in Distributed Software and Database Systems*. IEEE, 1986.

[102] Shrivastava, S. K.
Reliable Computer Systems.
Springer Verlag, Berlin, 1985.

[103] Smith, D. R.
Random Trees and the Analysis of Branch and Bound Procedures.
Journal of the Association for Computing Machinery, 31(1):163-188, January 1984.

[104] Jonathan M. Smith.
A Survey of Software Fault Tolerance Techniques.
Technical Report CUCS-325-88, Columbia University Computer Science Department, 1988.

[105] Jonathan M. Smith and Gerald Q. Maguire, Jr.
The RB Language.
Technical Report CUCS-338-88, Columbia University Computer Science Department, 1988.

[106] Stanfill, Craig, B. Kahle.
Parallel Free-Text Search on the Connection Machine System.
Communications of the ACM, 29(12):1229-1239, December, 1986.

[107] Stankovic, J.A., D. Towsley.
Dynamic Reallocation in a Highly Integrated Real-Time Distributed System.
In *Sixth International Conference on Distributed Computing Systems*. 1986.

[108] Strom, Robert E. and Shaula Yemini.
Optimistic Recovery: An Asynchronous Approach to Fault-Tolerance in Distributed Systems.
In *International Symposium on Fault Tolerant Computing*. 1984.

[109] Strom, Robert E., and Shaula Yemini.
Typestate: A Programming Language Concept for Enhancing Software Reliability.
IEEE Transactions on Software Engineering, SE-12(1)1986.

[110] Strom, Robert E., David F. Bacon, Shaula A. Yemini.
Volatile Logging in n-Fault-Tolerant Distributed Systems.
November, 1987.

[111] Strom, R., D. Bacon, S. Yemini.
Volatile Logging in n-Fault-Tolerant Distributed Systemsa.
In *International Symposium on Fault-Tolerant Computing*. 1988.

[112] Tamir, Yuval and Carlo H. Sequin.
Error Recovery in Multicomputers Using Global Checkpoints.
In *International Conference on Parallel Processing*. 1984.

[113] Tamir, Y., Gafni, E.
A Software-Based Hardware Fault Tolerance Scheme for Multicomputers.
In *Proceedings of the 1987 International Parallel Processing Conference*, pages 117-120. IEEE, Pennsylvania, 1987.

[114] Taylor, Seger.
Robust Storage Structures for Crash Recovery.
IEEE Transactions on Computers, C-35(4):, April 86.

[115] Taylor, D. J., J. P. Black, .
Guidelines for Storage Structure Error Correction.
In *International Symposium on Fault-Tolerant Computing*. 1985.

[116] Taylor, D. J.
How Big Can an Atomic Action Be?
In *Symposium on Reliability in Distributed Software and Database Systems*. 1986.

[117] Taylor, D. J.
Crash Recovery for Binary Trees.
In *International Symposium on Fault-Tolerant Computing*. 1987.

[118] Tsubotani, Hideaki, Noriaki Monden, Minoru Tanaka, and Tadao Ichikawa.
A High Level Language-Based Computing Environment to Support Production and Execution of Reliable Programs .
IEEE Transactions on Software Engineering, Se-12(2)1986.

[119] Uhlig, D.
On Reliable Networks from Unreliable Gates.
LNCS-269: Parallel Algorithms and Architectures.
Springer Verlag, May, 1987, pages 155-162.

[120] Uhr, L.
Multi-Computer Architectures for Artificial Intelligence.
John Wiley and Sons, New York, 1987.

[121] Uyar, M. Umit, Anthony Reeves.
Fault Reconfiguration for the Near Neighbor Problem In a Distributed MIMD Environment.
In *Distributed Computing Systems*. 1985.

[122] Uyar, M. Umit and Anthony P. Reeves.
Fault Reconfiguration in a Distributed MIMD Environment with a Multistage Network.
In *International Conference on Parallel Processing*. 1985.

[123] Uyar, M. Umit.
Dynamic Fault Reconfiguration in Multiprocessor Systems.
Cornell University, Ithica NY, June 1986.

[124] Varman, P. J., Ramakrishnan, I. V., Fussell, D. S.
Fault-Tolerant VLSI Sorters.
Circuits Systems Signal Processing, 6(2):153-174, 1987.

[125] Varman, P. J., and I. V. Ramakrishnan.
A Fault-Tolerant VLSI Matrix Multiplier.
In *International Conference on Parallel Processing*. 1986.

[126] Vinter, Ramamritham, Stemple.
Recoverable Actions in Guttenberg.
In *Sixth International Conference on Distributed Computing Systems*. 1986.

[127] vonNeumann.
Papers of John von Neumann on Computing and Computer Theory.
MIT Press, Charles Babbage Institute Reprint Series, Cambridge, Mass., 1987.

[128] Wakerly, J.
Error Detecting Codes, Self-Checking Circuits, and Applications.
Elsevier North-Holland, New York, 1978.

[129] Yanney, Raif M., John P. Hayes.
Distributed Recovery in Fault-Tolerant Multiprocessor Networks.
IEEE Transactions on Computers, C-35(10):871-879, October 1986.

[130] Yemini, S., D. M. Berry.
A Modular Verifiable Exception-Handling Mechanism.
ACM Transactions on Programming Languages and Systems, 7(2):215-241, April, 1985.

[131] Youn, h. Y., Singh, A.
On Area Efficient and Fault Tolerant Tree Embeddings in VLSI.
In *Proceedings of the 1987 International Parallel Processing Conference*, pages 170. IEEE, Pennsylvania, 1987.