Computer Science Journal of Moldova, vol.**13**, no.3(39), 2005

Approaches to Software Based Fault Tolerance – A Review

Goutam Kumar Saha

Abstract

This paper presents a review work on various approaches to software based fault tolerance. The aim of this paper is to cover past and present approaches to software implemented fault tolerance that rely on both software design diversity and on single but enhanced design.

1 Introduction

Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. The purpose of fault tolerance is to increase the dependability of a system. The objective of a fault-tolerant system is to mask faults (or to detect errors to switch to an alternate module) and continue to provide service despite faults. Fault tolerant systems must provide their specified services despite the occurrence of faults in the systems's components [1]. A failure occurs when an actual running system deviates from the specified behaviour. The cause of a failure is called an error. A fault is the root cause of a failure. In other words, an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors. Similarly, a single error may lead to multiple failures [116]. All fault tolerance techniques must use some form of redundancy to tolerate faults. Depending on the class of faults [76] redundant devices, networks, data or applications are used. Software fault tolerance relies either on design diversity or on single design using robust data structure. The only type of fault possible in software is a design fault introduced during the software development. Software faults are what we commonly call "bugs". According to [74], software faults are the root cause in a high percentage of operational system failures. The consequences of these failures depend on the application and the particular characteristic of the faults. The immediate effects can range from minor inconveniences (e.g., having to restart a hung personal computer) to catastrophic events (e.g., software in an aircraft that prevents the pilot from recovering from an input error) [75]. From a business perspective, operational failures caused by software faults can translate into loss of potential customers, lower sales, higher warranty repair costs, and losses due to legal actions from the people affected by the failures. There are four ways of dealing with software faults: prevention, removal, fault tolerance, and input sequence workarounds. Fault prevention is concerned with the use of design methodologies, techniques, and technologies aimed at preventing the introduction of faults into the design. Fault removal considers the use of techniques like reviews, analyses, and testing to check an implementation and remove any faults thereby exposed. The proper use of software engineering during the development processes is a way of realizing fault prevention and fault removal (i.e., fault avoidance). The use of fault avoidance is the standard approach for dealing with software faults and the many developments in the software field target the improvement of the fault avoidance techniques. [74,76] states that the software development process usually removes most of the deterministic design faults. This type of fault is activated by the inputs independently of the internal state of the software. A large number of the faults in operational software are state-dependent faults activated by particular input sequences. Given the lack of techniques that can guarantee that complex software designs are free of design fault, fault tolerance is sometimes used as an extra layer of protection. Software fault tolerance is the use of techniques to enable the continued delivery of services at an acceptable level of performance and safety after a design fault becomes active. Single version technique aims to improve the fault tolerance of a single piece of software by adding extra measures into the design aiming the error detection, containment, and tolerating errors caused by the design faults. Multi-version fault tolerance technique uses multiple versions (or variants) of a piece of software in a structured way to ensure that design faults in one version do not cause system failures. This is based on masking the design bugs. A characteristic of the software fault tolerance techniques is that they can be applied at any level in a software system: procedure, process, full application program, or the whole system including the operating system [4]. Such technique can also be applied selectively to those components that are deemed to have design faults due to their complexity [5]. Whatever measure we may take to remove software design bugs, software cannot be made free of design bugs. Though at present software errors have been attributed to be the main cause of most system failures. Transient errors often disrupt proper functioning (e.g., program hanging, wrong answer, false branching, data and code errors etc.) of an application program during the execution of an application system. But recent studies [2,3] have suggested that soft errors or transient bit-errors are increasingly responsible for system malfunctioning. Nowadays, computer systems are becoming more complex and are optimized for price and performance and not for availability. This makes soft errors an even more common case. Move towards denser, smaller, and low voltage transistors has the potential to increase these transient errors. Most system software architectures assume faith in underlying hardware, and software make no provisions to deal with hardware faults. [101] predicts that soft error rate (SER) per chip of logic circuits will increase nine orders of magnitude from 1992 to 2011 and at that point magnitude will be comparable to the SER per chip of unprotected memory elements. Researches in [104] have analyzed the effect of soft errors on system software. Software fault tolerance techniques have also been proposed in [102,103]. In this survey paper, we not only investigate various approaches to tolerate software design bugs but also we investigate the consequence of soft error on the system as a whole and current research into proposed recovery mechanisms along with transient fault tolerance. Multi version approach is basically on the assumption that software built differently might fail differently and thus, if one of the redundant versions fails, at least one of the others might provide an acceptable output. Recovery blocks, N-version programming, N self-checking programming, consensus recovery blocks, and $t/(n-1)$ techniques are also reviewed. Current research in software engineering focuses on establishing patterns in the software structure and trying to understand the practice of software engineering. We expect that software based fault tolerance research will be benefited by this research on enabling greater predictability of the dependability of software.

2 Various Approaches to Software Fault Tolerance

In this section, we investigate various software based fault tolerant approaches that rely on design diversity (multiple version) as well as on single design.

2.1 Design Diversity Based Software Fault Tolerance

Design Diversity Based or Multiple version based software fault tolerance is based on the use of at least two versions (or "variants") of a piece of software, executed either in sequence or in parallel. The versions are used as alternatives (with a separate means of error detection), in pairs (to implement detection by replication checks) or in larger groups (to enable masking through voting). The rationale for the use of multiple versions is the expectation that components built differently (i.e, different designers, different algorithms, different design tools, etc) should fail differently [6,7]. Therefore, if one version fails on a particular input, at least one of the alternate versions should be able to provide an appropriate output. This section covers some of these "design diversity" approaches to software reliability and safety. Basically multiple-version approach is to mask software design bugs. Two critical issues in the use of multi-version software fault tolerance techniques are the guaranteeing of independence of failure of the multiple versions and the development of the output selection algorithms.

Design diversity is "protection against uncertainty" [16]. In the case of software design, the uncertainty is in the presence of design faults and the failure modes due to those faults. The goal of design diversity techniques applied to software design is to build program versions that fail independently and with low probability of coincidental failures. If this goal is achieved, the probability of not being able to select a good output at a particular point during program execution is greatly reduced or eliminated. Due to the complexity of software, the use of design diversity for software fault tolerance is today more of an art rather than a science. The methodology of multiple-version software design was carried out by Algirdas Avizienis and his colleagues at UCLA starting in the 1970s [6,17,18,19,20,21,22,23,24]. Although focused mainly on software, their research considered the use of design diversity concepts for other aspects of systems like the operating system, the hardware, and the user interfaces. Assuming that the development is rigorous and design diversity is adequately applied to the product, there is still the common error source of the identical input profile. [25] points out that experiments (e,g, [26,27,28]) have shown that the probability of error manifestations are not equally distributed over the input space and the probability of coincident errors is impacted by the chosen inputs. Certainly data diversity techniques could be used to reduce the impact of this error source, but the problem of quantifying the effectiveness of the approach still remains. The cost of using multi-version software is also an important issue. A direct replication of the full development effort, including testing, would certainly be an expensive proposition. In some applications where only a small part of the functionality is safety critical, development and production cost can be reduced by applying design diversity only to those critical parts [16].

- **The Recovery Block Scheme**

The Recovery Block Scheme (RBS) technique [8,9] combines the basics of both the checkpoint and restart approach with multiple versions of a software component such that a different version is tried after an error is detected. Checkpoints are created before a version executes. Checkpoints are needed to recover the state after a version fails to provide a valid operational starting point for the next version if an error is detected. The acceptance test need not be an output-only test and can be implemented by various embedded checks to increase the effectiveness of the error detection. Also, because the primary version will be executed successfully most of the time, the alternates could be designed to provide degraded performance in some sense (e.g., by computing values to a lesser accuracy). Actual execution of the multiple versions can be sequential or in parallel depending on the available processing capability and performance requirements. If all the alternates are tried unsuccessfully, the component must raise an exception to communicate to the rest of the system its failure (or crash) to complete its function. Note that such a failure occurrence does not imply a permanent failure of the component, which may be reusable after changes in its inputs or state. The much possibility of coincident faults is the source of much controversy concerning all the multi-version software fault tolerance techniques.

- **The N-Version Programming Scheme**

The N-Version programming Scheme (NVPS) [7] is a multiple-version technique in which all the versions are designed to satisfy the same basic requirements and the decision of output correctness is based on the comparison of all the outputs. The use of a generic decision algorithm (usually a voter) to select the correct output is the fundamental difference of this approach from the Recovery Blocks approach, which requires an application dependent acceptance test. Since all the versions are built to satisfy the same requirements, the use of N-version programming requires considerable development effort but the complexity (i.e., development difficulty) is not necessarily much greater than the inherent complexity of building a single version. Design of the voter can be complicated by the need to perform inexact voting. [43] presents a generic two step structure for the output selection process. The first step is a filtering process where individual version outputs are analyzed by acceptance tests for likelihood of correctness, timing, completeness, and other characteristics. [44] presents four generalized voters for use in redundant systems: Formalized Majority Voter, Generalized Median Voter, Formalized Plurality Voter, and Weighted Averaging Techniques. The Weighted Averaging Technique combines the version outputs in a weighted average to produce a new output. The weights can be selected a-priori based on the characteristics of the individual versions and the application. When all the weights are equal this technique becomes a mean selection technique. The weights can also be selected dynamically based on the pair-wise distances of the version outputs [48] or the success history of the versions measured by some performance metric [44,47]. Other voting techniques have been proposed. For example, [45] proposed a selection function that always produces an acceptable output through the use of artificial intelligence techniques. [49, 50, 51] discuss the voting or majority output problem in some detail.

- **The N Self-Checking Programming Scheme**

The N Self-Checking Programming Scheme (NSCPS) [10,11,12] is the use of multiple software versions combined with structural variations of the Recovery Blocks and N-Version Programming. N Self-Checking programming uses acceptance tests. Here the versions and the acceptance tests are developed independently from common requirements. This use of separate acceptance tests for each version is the main difference of this N Self-Checking model from the Recovery Blocks approach. Similar to N-Version Programming, this model has the advantage of using an application independent decision algorithm to select a correct output.

- **The Consensus Recovery Blocks Scheme**

The Consensus Recovery Blocks Scheme (CRBS) [13] approach combines N-Version Programming and Recovery Blocks to improve the reliability over that achievable by using just one of the approaches. The acceptance tests in the Recovery Blocks suffer from lack of guidelines for their development and a general proneness to design faults due to the inherent difficulty in creating effective tests. The use of voters as in N-Version Programming may not be appropriate in all situations, especially when multiple correct outputs are possible. In that case a voter, for example, would declare a failure in selecting an appropriate output. Consensus Recovery Blocks uses a decision algorithm similar to N-Version Programming as a first layer of decision. If this first layer declares a failure, a second layer using acceptance tests similar to those used in the Recovery Blocks approach is invoked. Although obviously much more complex than either of the individual techniques, the reliability models indicate that this combined approach has the potential of producing a more reliable piece of software.

- **The $t/(n-1)$-Variant Programming Scheme**

The $t/(n-1)$-Variant Programming Scheme (VPS) was proposed in [14]. The main difference between this approach and the ones mentioned above is in the mechanism used to select the output from among the multiple variants. The design of the selection logic is based on the theory of system-level fault diagnosis [15], which is beyond the scope of this paper. Basically, a $t/(n-1)$-VPS architecture consists of n variants and uses the $t/(n-1)$ diagnosibility measure to isolate the faulty units to a subset of size at most $(n-1)$ assuming there are at most t faulty units [14]. Thus, at least one non-faulty unit exists such that its output is correct and can be used as the result of computation for the module. $t/(n-1)$-VPS compares favorably with other approaches in that the complexity of the selection mechanism grows with order $O(n)$ and it can potentially tolerate multiple dependent faults among the versions. It also has a lower probability of failure than N Self-Checking Programming and N-Version Programming when they use a simple voter as selection logic.

2.2 Single-Design Software Fault Tolerance Approach

Single-design fault tolerance is based on the use of redundancy applied to a single version of a piece of software to detect and recover from faults. Among others, single-version software fault tolerance techniques include considerations on program structure and actions, error detection, exception handling, checkpoint and restart, process pairs, and data diversity [29].

- **Software Engineering Aspects**

Software architecture gives us the basis for implementation of fault tolerance. The use of modularizing techniques to decompose a problem into manageable components is as important to the efficient application of fault tolerance as it is to the design of a system. The modular decomposition of a design should consider built-in protections to keep aberrant component behavior in one module from propagating to other modules. Control hierarchy issues like visibility (i.e., the set of components that may be invoked directly and indirectly by a particular component) and connectivity (i.e., the set of components that may be invoked directly or used by a given component) should be considered in the context of error propagation for their potential to enable uncontrolled corruption of the system state. Partitioning is a technique for providing isolation between functionally independent modules [31]. Advantages of using partitioning in a design include simplified testing, easier maintenance, and lower propagation of side effects [30]. System closure is a fault tolerance principle stating that no action is permissible unless explicitly authorized [32]. An atomic action among a group of components is an activity in which the components interact exclusively with each other and there is no interaction with the rest of the system for the duration of the activity [33]. The advantage of using atomic actions in defining the interaction between system components is that they provide a framework for error confinement and recovery. There are only two possible outcomes of an atomic action: either it terminates normally or it is aborted upon error detection. If an atomic action terminates normally, its results are complete and committed. If a failure is detected during an atomic action [76], it is known beforehand that only the participating components can be affected.

- **Error Detection Mechanisms**

Effective application of fault tolerance techniques in single version systems requires that the structural modules have two basic properties: self-protection and self-checking [34]. The self-protection property means that a component must be able to protect itself from external contamination by detecting errors in the information passed to it by other interacting components. Self-checking means that a component must be able to detect internal errors and take appropriate actions to prevent the propagation of those errors to other components. The degree (and coverage) to which error detection mechanisms are used in a design is determined by the cost of the additional redundancy and the run-time overhead. Note that the fault tolerance redundancy is not intended to contribute to system functionality but rather to the quality of the product. Similarly, detection mechanisms detract from system performance. Actual usage of fault tolerance in a design is based on trade-offs of functionality, performance, complexity, and safety. Anderson [33] has proposed a classification of error detection checks, some of which can be chosen for the implementation of the module properties mentioned above. The location of the checks can be within the modules or at their outputs, as needed. The checks include replication, timing, reversal, coding, reasonableness, and structural checks.

- The use of **Assertions** [105] that is logic statements inserted at different points in the program that reflects invariant relationships between the variables of the program can also be used for fault tolerance. However it can lead to different problems, since assertions are not transparent to the programmer and their effectiveness largely depends on the nature of the application and on the programmers ability.
- **Control Flow Checking** [106] is to partition the application program in basic blocks (that is, branch-free parts of code). For each block a deterministic signature is computed and faults can be detected by comparing the run-time signature with a precomputed one. In most control-flow checking techniques one of the main problems is to tune the test granularity that should be used.
- **Replication checks** make use of matching components with error detection based on comparison of their outputs. This is applicable to multi-version software fault tolerance.
- **Timing checks** are applicable to systems and modules whose specifications include timing constraints, including deadlines. Based on these constraints, checks can be developed to look for deviations from the acceptable module behavior. Watchdog timers are a type of timing check with general applicability that can be used to monitor for satisfactory behavior and detect "lost or locked out" components.
- **Reversal checks** use the output of a module to compute the corresponding inputs based on the function of the module. An error is detected if the computed inputs do not match the actual inputs. Reversal checks are applicable to modules whose inverse computation is relatively straightforward.
- **Coding checks** use redundancy in the representation of infor-

mation with fixed relationships between the actual and the redundant information. Error detection [109, 110] is based on checking those relationships before and after operations. Checksums are a type of coding check. Similarly, many techniques developed for hardware (e.g., Hamming, M-out-of-N, cyclic codes) can be used in software, especially in cases where the information is supposed to be merely referenced or transported by a module from one point to another without changing its contents. Many arithmetic operations preserve some particular properties between the actual and redundant information, and can thus enable the use of this type of check to detect errors in their execution.

- **Reasonableness checks** use known semantic properties of data (e.g., range, rate of change, and sequence) to detect errors. These properties can be based on the requirements or the particular design of a module.

- The **Data Structural checks** use known properties of data structures. For example, queues, lists, and trees can be inspected for number of elements in the structure, their links and pointers, and any other particular information that could be articulated. Structural checks could be made more effective by augmenting data structures with redundant structural data like extra pointers, embedded counts of the number of items on a particular structure, and individual identifiers for all the items [34, 35, 36, 37, 38]. Another fault detection tool is run-time checks [15]. These are provided as standard error detection mechanisms in hardware systems (e.g., divide by zero, overflow, underflow). Although they are not application specific, they do represent an effective means of detecting design errors. Error detection strategies can be developed in an ad-hoc fashion or using structured methodologies. Fault trees have been proposed as a design aid in the development of fault detection strategies [39]. Fault trees can be used to identify general classes of failures and conditions that can trigger those failures. Fault trees represent a top-down approach which, although not guaranteeing complete coverage, is very helpful in documenting assumptions, simplifying design reviews, identifying omissions, and allowing the designer to visualize component interactions and their consequences through structured graphical means. Fault trees enable the designer to perform qualitative analysis of the complexity and degree of independence in the error checks of a proposed fault tolerance strategy.

- **The Exception Handling**

The task exception handling is the interruption of normal operation to handle abnormal responses. Exceptions are signaled by the implemented error detection mechanisms as a request for initiation of an appropriate recovery. The design of exception handlers requires that consideration be given to the possible events triggering the exceptions, the effects of those events on the system, and the selection of appropriate mitigating actions [15]. [8] lists three classes of exception triggering events for a software component: interface exceptions, internal local exceptions, and failure exceptions. This knowledge of error containment is essential to the design of effective exception handlers.

- **The Checkpoint and Restart**

For single-design software there are few recovery mechanisms. The most often mentioned is the checkpoint and restart mechanism (e.g., [15]). As mentioned in previous sections, most of the software faults remaining after development are unanticipated, state-dependent faults [76]. This type of fault behaves similarly to transient hardware faults: they appear, do the damage, and then apparently just go away, leaving behind no obvious reason for their activation in the first place [40]. Because of these characteristics, simply restarting a module is usually enough to allow successful completion of its execution [40]. A restart, or backward error recovery has the advantages of being independent of the damage caused by a fault, applicable to unanticipated faults, general enough that it can be used at multiple levels in a system, and conceptually simple [33]. There exist two kinds of restart recovery: static and dynamic. A static restart is based on returning the module to a predetermined state. This can be a direct return to the initial reset state, or to one of a set of possible states, with the selection being made based on the operational situation at the moment the error detection occurred. Dynamic restart uses dynamically created checkpoints that are snapshots of the state at various points during the execution.

- **The Process Pairs**

A process pair uses two identical versions of the software that run on separate processors [15]. The recovery mechanism is checkpoint and restart. Here the processors are labeled as primary and secondary. At first the primary processor is actively processing the input and creating the output while generating checkpoint information that is sent to the backup or secondary processor. Upon error detection, the secondary processor loads the last checkpoint as its starting state and takes over the role of primary processor. As this happens, the faulty processor goes offline and executes diagnostic checks. The main advantage of this recovery technique is that the delivery of services continues uninterrupted after the occurrence of a failure in the system.

- **The Data Diversity**

The last line of defense against design faults is to use "input sequence workarounds". Data diversity can be seen as the automatic implementation of "input sequence workarounds" combined with checkpoint and restart [76]. Again, the rationale for this technique is that faults in deployed software are usually input sequence dependent. Data diversity has the potential of increasing the effectiveness of the checkpoint and restart by using different input re-expressions on each retry [41]. The goal of each retry is to generate output results that are either exactly the same or semantically equivalent in some way. In general, the notion of equivalence is application dependent. [41, 42] presents three basic data diversity models: (i) Input Data Re-Expression, where only the input is changed; (ii) Input Re-Expression with Post-Execution Adjustment, where the output is also processed as necessary to achieve the required output value or format; (iii) Re-Expression via Decomposition and Recombination, where the input is broken down into smaller elements and then recombined after processing to form the desired output. Data diversity is compatible with the Process Pairs technique using different re-expressions of the input in the primary and secondary.

- **Fault Tolerance in Operating Systems**

Any application level software relies on the correct behavior of the operating system. Software fault tolerance can be applied to the design of operating systems [32,76]. However, in general, designing and building operating systems tends to be a rather complex, lengthy and costly endeavor. For safety critical applications it may be necessary to develop custom operating systems through highly structured design processes [31] including highly experienced programmers and advanced verification techniques in order to gain a high degree of confidence on the correctness of the software. Another approach to the development of fault tolerant operating systems for mission critical applications is the use of wrappers on off-the-shelf operating systems to boost their robustness to faults. A problem with the use of off-the-shelf software on dependable systems is that the system developers are not sure if the off-the-shelf components are reliable enough for the application [52]. It is known that the development process for commercial off-the-shelf software does not consider de facto standards for safety or mission critical applications and the available documentation for the design and validation activities tend to be rather weak [53]. A point in favor of using commercial operating systems is that they often include the latest developments in operating system technology. Also, widely deployed commercial operating systems could have fewer bugs overall than custom developed software due to the corrective actions performed in response to bug complaints from the users [54]. Because modifications to the internals of the operating system could increase the risk of introducing design faults, it is preferred to apply techniques that use the software as is. A wrapper is a piece of software put around another component to limit what that component can do without modifying the component's source code [52]. Wrappers monitor the flow of information into and out of the component and try to keep undesirable values from being propagated. In this manner, the wrapper limits the component's input and output spaces. Wrappers have been used as middleware located between the operating system and the application software [55, 56, 57]. The wrappers (called "sentries" in the referenced work) encapsulate operating system services to provide application-transparent fault tolerant functionality and can augment or change the characteristics of the services as seen by the application layer. In this design the sentries provide the mechanism to implement fault tolerance policies that can be dynamically assigned to particular applications based on the individual fault tolerance, cost and performance needs. [53] proposed the use of wrappers at the microkernel level for off-the-shelf operating systems. The wrappers proposed by these researchers aim at verifying consistency constraints at a semantic level by utilizing information beyond what is available at the interface of the wrapped component. Their approach uses abstractions [76] (i.e., models) of the expected component functionality.

3 Assessment of Fault Tolerance by Fault Injection

Software fault injection (SFI) [76] is the process of testing software under anomalous circumstances involving erroneous external inputs or internal state information. The main reason for using software fault injection is to assess the goodness of a design [65]. Basically, SFI tries to measure the degree of confidence that can be placed on the proper delivery of services. Since it is very hard to produce correct software, SFI tries to show what could happen when faults are activated. The collected information can be used to make code less likely to hide faults and also less likely to propagate faults to the outputs either by reworking the existing code or by augmenting its capabilities with additional code as done with wrappers [65]. SFI can be used to target both objectives of the dependability validation process: fault removal and fault forecasting [62]. In the context of fault removal, SFI can be used as part of the testing strategy during the software development process to see if the designed algorithms and mechanisms work as intended. In fault forecasting, SFI is used to assess the fault tolerance robustness of a piece of software (e.g., an off-the-shelf operating system). The use of SFI has two important advantages over the traditional input sequence test cases [59]. First, by actively injecting faults into the software we are in effect accelerating the failure rate and this allows a thorough testing in a controlled environment within a limited time frame. Second, by systematically injecting faults to target particular mechanisms we are able to better understand the behavior of that mechanism including error propagation and output response characteristics. There exist two basic models of software injection: fault injection and error injection. Fault injection simulates software design faults by targeting the code. Here the injection considers the syntax of the software to modify it in various ways with the goal of replacing existing code with new code that is semantically different [65]. This “code mutation” can be performed at the source code level before compilation if the source code is available. The mutation can also be done by modifying the text segment of a program’s object code after compilation. Error injection, called “data-state mutation” in [65], targets the state of the program to simulate fault manifestations. Actual state injection can be performed by modifying the data of a program using any of various available mechanisms: high priority processes that modify lower priority processes with the support of the operating system; debuggers that directly change the program state; message-based mechanisms where one component corrupts the messages received by another component; storage-based mechanisms by using storage (e.g., cache, primary, or secondary memory) manipulation tools; or command-based approaches that change the state by means of the system administration and maintenance interface commands [59]. An important aspect of both types of fault injection is the operational profile of the software [65]. Fault injection is a dynamic-type testing because it must be used in the context of running software following a particular input sequence and internal state profile. A large amount of work has been done in the area of assessing software robustness by many researchers. Examples of reported works include [54, 58, 60, 61, 63, 64].

4 Software and Hardware Fault Tolerance

System fault tolerance is a vast area of knowledge well beyond what can be covered in a single paper. The concepts presented in this section are purposely treated at a high level with details considered only where regarded as appropriate. Readers interested in a more thorough treatment of the concepts of computer system fault tolerance should consult additional reference material [15, 66, 67].

- **Fault Tolerance in Computer System**

Computer fault tolerance is one of the means available to increase dependability of delivered computational services. Dependability is a quality measure encompassing the concepts of reliability, availability, safety, performability, maintainability and testability [68]. (i) Reliability is the probability that a system continues to operate correctly during a particular time interval given that it was operational at the beginning of the interval. (ii) Availability is the probability that a system is operating correctly at a given time instant. (iii) Safety is the probability that the system will perform in a non-hazardous way. A hazard is defined as “a state or condition of a system that, together with other conditions in the environment of the system, will lead inevitably to an accident” [69]. (iv) Performability is the probability that the system performance will be equal to or greater than some particular level at a given instant of time. (v) Maintainability is the probability that a failed system will be returned to operation within a particular time period. Maintainability measures the ease with which a system can be repaired. (vi) Testability is a measure of the ability to characterize a system through testing. Testability includes the ease of test development (i.e., controllability) and effect observation (i.e., observability).

The primary concern for fault tolerant designs is the ability to continue delivery of services in the presence of faults in the system. A fault is an anomalous condition occurring in the system hardware or software. [66,70] presents a general fault classification which is excellent for understanding the types of faults that fault tolerant designs are called upon to handle. A *latent fault* is a fault that is present in the system but has not caused errors; after errors occur, the fault is said to be active. Permanent faults are present in the system until they are removed; transient faults appear and disappear on their own with no explicit intervention from the system. *Symmetric faults* are those perceived identically by all good subsystems; asymmetric faults are perceived differently by the good subsystems. A *random fault* is caused by the environment (e.g., heat, humidity, vibration, etc.) or by component degradation; generic faults are built-in faults accidentally introduced during design or manufacturing of the system. *Benign faults* are detectable by all good subsystems; malicious faults are not directly detectable by all good subsystems. The fault count classification is relative to the modularity of the system. A *single fault* is a fault in a single system module; a group of multiple faults affects more than one module.

The *time classification* is relative to the time granularity. Coincident multiple faults appear during the same time interval; distinct-time faults appear in different time intervals. Independent faults are faults originating from different causes or nature. Common mode faults, in the context of multiple faults, are faults that have the same cause and are present in multiple components.

The main use of fault tolerance in these systems is to provide added value and prevent nuisance faults from affecting the perceived dependability from a user perspective. The design of systems with fault tolerance capabilities to satisfy particular application requirements is a complex process loaded with theoretical and experimental analysis in order to find the most appropriate tradeoffs within the design space. [66] offers a high-level design paradigm extracted from the more detailed description presented in [70]. System properties to be considered include dependability (i.e., reliability, availability, maintainability, etc), performance, failure modes, environmental resilience, weight, cost, volume, power, design effort, and verification effort.

Every fault tolerant design must deal with one or more of the following aspects [33, 71,76]:

- Detection: A basic element of a fault tolerant design is error detection. Error detection [96, 97, 98, 99, 100] is a critical prerequisite for other fault tolerant mechanisms.

- Containment: In order to be able to deal with the large number of possible effects of faults in a complex computer system it is necessary to define confinement boundaries for the propagation of errors. Containment regions are usually arranged hierarchically throughout the modular structure of the system. Each boundary protects the rest of the system from errors occurred within it and enables the designer to count on a certain number of correctly operating components by means of which the system can continue to perform its function.

- Masking: For some applications, the timely flow of information is a critical design issue. In such cases, it is not possible to just stop the information processing to deal with detected errors. Masking is the dynamic correction of errors. In general, masking errors is difficult to perform inline with a complex component. Masking, however, is much simpler when redundant copies of the data in question are available.

- Diagnosis: After an error is detected, the system must assess its health in order to decide how to proceed. If the containment boundaries are highly secure, diagnosis is reduced to just identifying the enclosed components. If the established boundaries are not completely secure, then more involved diagnosis is required to identify which other areas are affected by propagated errors.

- Repair/reconfiguration: In general, systems do not actually try to repair component-level faults in order to continue operating. Because faults are either physical or design-related, repair techniques are based on finding ways to work around faults by either effectively removing from operation the affected components or by rearranging the activity within the system in order to prevent the activation of the faults.

- Recovery and Continued Service: After an error is detected, a system must be returned to proper service by ensuring an error-free state. This usually involves the restoration to a previous or predefined state, or rebuilding the state by means of known-good external information.

Redundancy in computer systems is the use of resources beyond the minimum needed to deliver the specified services. Fault tolerance is achieved through the use of redundancy in the hardware, software, information, or time domain [68,71,73]. In what follows we present some basic concepts of hardware redundancy to achieve hardware fault tolerance. Good examples of information domain redundancy for hardware fault tolerance are error detecting and correcting codes [72]. Time redundancy is the repetition of computations in ways that allow faults to be detected [68, 76].

5 Implemented Structures

Here, we present few examples of fault tolerant architectures that are implemented in various important applications.

Fault-tolerance in *Maintainable Real-Time System* (MARS) [82,83] is based on fail-silent components running in dual active redundancy and on sending each message twice on the two actively redundant real-time busses. MARS is a fault –tolerant distributed real-time architecture for hard real-time application.

CRAK [84, 85, 86] is a kernel module that implements checkpoint / restart for Linux. Checkpoint / restart is an operating system feature that creates a file describing a running process. Checkpoint / restart is a mechanism for fault tolerance. Applications may be checkpointed periodically. Once the application state has been committed to stable storage, the application may be restarted and reconfigured to work around the fault.

Safety critical fault tolerant architectures are used on the flight control computers of the fly-by-wire systems of two types of commercial jet transport aircraft. The first computer is used on the *Boeing 777* airplane [76]. The second computer is used on the AIRBUS A320/A330/A340 series aircraft. The fly-by-wire system of the Boeing 777 airplane departs from old-style mechanical systems that directly connect the pilot's control instruments to the external control surfaces. A fly-by-wire system enables the creation of artificial airplane flight characteristics that allow crew workload alleviation and flight safety enhancement, as well as simplifying maintenance procedures through modularization and automatic periodic self-inspection [77,78, 79, 80, 81]. Software diversity was to be achieved through the use of different programming languages targeting different lane processors. The final and current implementation uses only one programming language with the executable code being generated by three different compilers still targeting dissimilar lane processors. The lane processors are dissimilar because they are the single most complex hardware devices, and thus there is a perceived risk of design faults associated with their use. The requirements for the flight control computer on the Airbus A320/A330/A340 [76] include many of the same considerations as in the B777 fly-by-wire system [87, 88]. The selected architecture, however, is much different. The basic building block is the fail-stop control and monitor module.

Active-stream / Redundancy-stream Simultaneous Multithreading (AR_SMT) [89, 90, 91] exploits several recent microarchitectural trends (e.g., simultaneous multithreading [94, 95], control flow and data flow prediction and hierarchical processors) to provide low-overhead, broad coverage of transient faults and restricted coverage of some permanent faults. Program-level time redundancy is used here. Time redundancy [92, 93, 107, 108] is a fault tolerant technique in which a computation is performed multiple times.

Self-stabilization [115] is an *optimistic* way of looking at system fault tolerance, because it provides a built-in safeguard against transient failures that might corrupt the data in a distributed system. Although the concept was introduced by Dijkstra in 1974 [111], and Lamport [112] showed its relevance to fault tolerance in distributed systems in 1983, serious work only began in the late nineteen-eighties. A good survey of self-stabilizing algorithms can be found in [113]. Herman's bibliography [114] also provides a fairly comprehensive listing of most papers in this field. Because of the size and nature of many ad hoc and geographically distributed systems, communication links are unreliable. The system must therefore be able to adjust when faults occur. But 100% fault tolerance is not warranted. The promise of self-stabilization, as opposed to fault masking, is to recover from failure in a reasonable amount of time and without intervention by any external agency. Since the faults are transient (eventual repair is assumed), it is no longer necessary to assume a bound on the number of failures. A fundamental idea of self-stabilizing algorithms is that the distributed system may be started from an arbitrary global state. After a finite amount of time the system reaches a correct global state, called a *legitimate* or *stable* state. An algorithm is self-stabilizing if (i) for any initial illegitimate state it reaches a legitimate state after a finite number of node moves, and (ii) for any legitimate state and for any move allowed by that state, the next state is a legitimate state. A self-stabilizing system does not guarantee that the system is able to operate properly when a node continuously injects faults in the system (Byzantine fault that generates wrong and random answer) or when communication errors occur so frequently that the new legitimate state cannot be reached. While the system services are unavailable when the self-stabilizing system is in an illegitimate state, the repair of a self-stabilizing system is simple; once the offending equipment is removed or repaired the system provides its service after a reasonable time.

[117] have proposed a *transient fault tolerant design* that takes good advantage of the resource for parallel executions found in a superscalar processor. The design in [117] delivers fault tolerance by carrying out multiple executions of the same instruction in lower performance.

The approaches in [118-140] are all low-cost but effective software fault tolerance tool that do not rely on software design diversity. These approaches are all based on enhanced single-version programming (ESVP) schemes for fault tolerance against operational faults, transient and permanent errors etc. Replicated or transformed code [118,119,125] and data have also been used. [120,121,126,127,129] propose other important single-version fault tolerance approaches to design reliable applications using robust data structure for tolerating erroneous control flow and program hanging. [122,136] propose various low-cost software based fault tolerance approaches for designing microprocessor based commodity applications on inserting NO-Operation (NOP) instructions and run time consistency checking for those extra NO-Operation codes. The approaches in [123,124,128] describe various self-checking and assertion based approaches for designing reliable computing by enhancing basic computing logic. The approaches in [125,130] rely on code and data redundancy for detecting run-time error detection and recovery thereof. [138] proposes a single version software implemented transient fault tolerant approach for designing a reliable application using a multiprocessor with lower overhead on execution time. [131, 132, 133, 134, 135, 137, 139,140] propose various enhanced single-version software implemented transient fault tolerant computing schemes using fault masking with an affordable time redundancy (< 3) and program state verification. Specific knowledge of the application allows the use of a suite of math, logic and heuristic checks on the data, the data processing flow and the results. These techniques also provide more efficient error handling, and system recovery.

The approaches in [118-140] are useful specifically for designing low cost reliable computing applications against operational, transient and permanent errors that might occur during the execution time of applications. ESVP does not aim to tolerate software design bugs. ESVP needs only one reliable machine to execute an application with an enhanced processing logic. It does not rely on multiple versions of software and machines. The designers also feel comfortable while implementing these simple approaches to their applications without any extra cost and hardware.

6 Conclusion

A review on software fault tolerance is presented in this paper. It covers fault tolerant computing schemes that rely on the single-design as well as on the multiple-design. Single version software fault tolerance techniques discussed include system structuring and closure, atomic actions, inline fault detection, exception handling, assertion, and checkpoint and restart. Process pairs exploit the state dependence characteristic of most software faults to allow uninterrupted delivery of services despite the activation of faults. Similarly, data diversity aims at preventing the activation of design faults by trying multiple alternate input sequences. Multiple-version techniques are based on the assumption that software built differently should fail differently and thus, if one of the redundant versions fails, at least one of the others should provide an acceptable output. Because of our present inability to produce error-free software, software fault tolerance is and will continue to be an important consideration in software systems.

The root cause of software design errors is the complexity of the systems. Compounding the problems in building correct software is the difficulty in assessing the correctness of software for highly complex systems. Current research in software engineering focuses on establishing patterns in the software structure and trying to understand the practice of software engineering aiming at better predicting the software dependability. Multiple-version schemes are costlier (O(2.67)) than a single-version software fault tolerance approach because of lower software development cost.

References

[1] L. Spainhower and T.A. Gregg, *IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: a Historical Perspective*, IBM Journal of Research & Development, Vol.43, No. 5/6, 1999.

[2] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D.D. Mannaru, A. Riska and D. Milojicic, *Susceptibility of Commodity Systems and Software to Memory Soft Errors*, IEEE Transactions on Computers, Vol. 53, No. 12, December 2004, pp. 1557–1568.

[3] H. Kopetz, H. Kantz, G. Grilnsteidl, P. Puschner and J. Reisinger, *Tolerating Transient Faults in MARS*, Proc. of the 20th Symposium on Fault Tolerant Computing, June 1990, UK.

[4] Brian Randell, *System Structure for Software Fault Tolerance*, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 220–232.

[5] Michael R. Lyu, editor, *Software Fault Tolerance*, John Wiley & Sons, 1995.

[6] Algirdas Avizienis, *The N-Version Approach to Fault-Tolerant Software*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, December 1985, pp. 290–300.

[7] Algirdas Avizienis, *The Methodology of N-Version Programming*, in R. Lyu, editor, Software Fault Tolerance, John Wiley & Sons, 1995.

[8] Brian Randell and Jie Xu, *The Evolution of the Recovery Block Concept*, in Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 1–21.

[9] Brian Randell, et al, editors, *Predictably Dependable Computing Systems*, Springer, 1995.

[10] J.C. Laprie, et al, *Hardware- and Software-Fault Tolerance: Definition and Analysis of Architectural Solutions*, Digest of Papers FTCS-17: The Seventeenth International Symposium on Fault-Tolerant Computing, July 1987, pp. 116–121.

[11] Jean-Claude Laprie, et al, *Definition and Analysis of Hardware- and Software-Fault- Tolerance Architectures*, IEEE Computer, July 1990, pp. 39–51.

[12] J.C. Laprie, et al, *Architectural Issues in Software Fault Tolerance*, in Software Fault Tolerance, Michael R. Lyu, editor, Wiley, 1995, pp. 47–80.

[13] R. Keith Scott, James W. Gault, and David F. McAllister, *Fault-Tolerant Software Reliability Modeling*, IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987, pp. 582–592.

[14] Jie Xu and Brian Randell, *Software Fault Tolerance: t/(n-1)-VariantProgramming*, IEEE Transactions on Reliability, Vol. 46, No. 1, March 1997, pp. 60–68.

[15] Dhiraj K. Pradhan, *Fault-Tolerant Computer System Design*, Prentice-Hall, Inc., 1996.

[16] Peter Bishop, *Software Fault Tolerance by Design Diversity*, in R. Lyu, editor, Software Fault Tolerance, John Wiley & Sons, 1995.

[17] A. Avizienis, et al, *The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software*, Digest of Papers: The Fifteenth Annual International Symposium on Fault-Tolerant Computing (FTCS 15), Ann Arbor, Michigan, June 19–21, 1985, pp. 126–134.

[18] Algirdas Avizienis and Jean-Claude Laprie, *Dependable Computing: From Concepts to Design Diversity*, Proceedings of the IEEE, Vol. 74, No. 5, May 1986, pp. 629–638.

[19] Algirdas Avizienis, *In Search of Effective Diversity: A Six-Language Study of Fault-Tolerant Flight Control Software*, Digest of Papers FTCS-18: The Eighteenth International Symposium on Fault-Tolerant Computing, June 27–30, 1988, pp. 15–22.

[20] Algirdas Avizienis, *Software Fault Tolerance*, Information Processing 89, Proceedings of the IFIP 11 th World Computer Congress, 1989, pp. 491–98.

[21] Algirdas Avizienis, *Dependable Computing Depends on Structured Fault Tolerance*, Proceedings of the 1995 6^{th} International Symposium on Software Reliability Engineering, Toulouse, France, 1995, pp. 158–168.

[22] Algirdas Avizienis, *The Methodology of N-Version Programming*, in R. Lyu, editor, Software Fault Tolerance, John Wiley & Sons, 1995.

[23] Algirdas Avizienis, *Toward Systematic Design of Fault-Tolerant Systems*, Computer, April 1997, pp. 51–58.

[24] Thomas C. Bressoud, *TFT: A Software System for Application-Transparent Fault Tolerance*, Digest of Papers: Twenty-Eight Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23–25, 1998, pp. 128–137.

[25] F. Saglietti, *Strategies for the Achievement and Assessment of Software Fault-Tolerance*, IFAC 1990 World Congress, Automatic

Control. Vol. IV, IFAC Symposia Series, Number 4, 1991, pp. 303–308.

[26] J. C. Knight, et al, *A Large Scale Experiment in N-Version Programming*, Digest of Papers FTCS-15: The 15th Annual International Conference on Fault Tolerant Computing, June 1985, pp. 135–139.

[27] J. C. Knight and Nancy G. Leveson, *An Experimental Evaluation of the Assumption of Independence in Multiversion Programming*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, pp. 96–109.

[28] Dave E. Eckhardt, et al, *An Experimental Evaluation of Software Redundancy as a Strategy for Improving Reliability*, IEEE Transactions on Software Engineering, Vol. 17, No. 7, July 1991, pp. 692–702.

[29] Michael R. Lyu, editor, *Software Fault Tolerance*, John Wiley & Sons, 1995.

[30] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, The McGraw-Hill Companies, Inc., 1997

[31] *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B, RTCA, Inc, 1992.

[32] Peter J. Denning, *Fault Tolerant Operating Systems*, ACM Computing Surveys, Vol. 8, No. 4, December 1976, pp. 359–389.

[33] T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice/Hall, 1981.

[34] Russell J. Abbott, *Resourceful Systems for Fault Tolerance, Reliability, and Safety*, ACM Computing Surveys, Vol. 22, No. 1, March 1990, pp. 35–68.

[35] David J. Taylor, et al, *Redundancy in Data Structures: Improving Software Fault Tolerance*, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 585–594.

[36] David J. Taylor, et al, *Redundancy in Data Structures: Some Theoretical Results*, IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 595–602.

[37] J. P. Black, et al, *Introduction to Robust Data Structures*, Digest of Papers FTCS-10: The Eleventh Annual International Symposium on Fault-Tolerant Computing, October 1–3, 1980, pp. 110–112.

[38] J. P. Black, et al, *A Compendium of Robust Data Structures*, Digest of Papers FTCS-11: The Eleventh Annual International Symposium on Fault-Tolerant Computing, June 24–26, 1981, pp. 129–131.

[39] Herbert Hecht and Myron Hecht, *Fault-Tolerance in Software, in Fault-Tolerant Computer System Design*, Dhiraj K. Pradhan, Prentice Hall, 1996.

[40] Jim Gray, *Why Do Computers Stop and What Can Be Done About It?* Proceedings of the Fifth Symposium On Reliability in Distributed Software and Database Systems, January 13–15, 1986, pp. 3–12.

[41] Paul E. Ammann and John C. Knight, *Data Diversity: An Approach to Software Fault Tolerance*, IEEE Transactions on Computers, Vol. 37, No. 4, April 1988, pp. 418–425.

[42] Victor F. Nicola, *Checkpointing and the Modeling of Program Execution Time*, in Software Fault Tolerance, Michael R. Lyu, Ed, Wiley, 1995, pp. 167–188.

[43] Tom Anderson, *A Structured Mechanism for Diverse Software*, Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems, January 1986, pp. 125–129.

[44] Paul R. Lorczack, et al, *A Theoretical Investigation of Generalized Voters for Redundant Systems*, Digest of Papers FTCS-19: The Nineteenth International Symposium on Fault-Tolerant Computing, 1989, pp. 444–451.

[45] P. R. Croll, et al, *Dependable, Intelligent Voting for Real-Time Control Software*, Engineering Applications of Artificial Intelligence, vol. 8, no. 6, December 1995, pp. 615–623.

[46] Judith Gersting, et al, *A Comparison of Voting Algorithms for N-Version Programming*, Proceedings of the 24^{th} Annual Hawaii International Conference on System Sciences, Volume II, January 1991, pp. 253–262.

[47] J. M. Bass, *Voting in Real-Time Distributed Computer Control Systems*, PhD Thesis, University of Sheffield, October 1995.

[48] R. B. Broen, *New Voters for Redundant Systems*, Transactions of the ASME, Journal of Dynamic Systems, Measurement, and Control, March 1975, pp. 41–45.

[49] John P. J. Kelly, et al, *Multi-Version Software Development*, Proceeding of the Fifth IFAC Workshop, Safety of Computer Control Systems, October 1986, pp. 43–49.

[50] Kam Sing Tso and Algirdas Avizienis, *Community Error Recovery in N-Version Software: A Design Study with Experimentation*, Digest of Papers FTCS-17: The Seventeenth International Symposium on Fault-Tolerant Computing, July 6–8, 1987, pp. 127–133.

[51] F. Saglietti, *Software Diversity Metrics: Quantifying Dissimilarity in the Input Partition*, Software Engineering Journal, January 1990, pp. 59–63.

[52] Jeffrey M. Voas, *Certifying Off-the-Shelf Software Components*, IEEE Computer, Vol. 31, June 1998, pp. 53–59.

[53] Frédéric Salles, et al, *MetaKernels and Fault Containment Wrappers*, Digest of Papers: Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, June 15–18, 1999, pp. 22–29.

[54] Philip Koopman, et al, *Comparing Operating Systems Using Robustness Benchmarks*, Proceedings of the 1997 16^{th} IEEE Symposium on Reliable Distributed Systems, October 1997, pp. 72–79.

[55] Mark Russinovich, et al, *Application Transparent Fault Management in Fault Tolerant Mach*, Digest of Papers: The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23), Toulouse, France, June 22–24, 1993, pp. 10–19.

[56] Mark Russinovich, et al, *Application Transparent Fault Managemet in Fault Tolerant Mach*, in Foundations of Dependable Computing System Implementation, Gary M. Koob and Clifford G. Lau, editors, Kluwer Academic Publishers, 1994, pp. 215–241.

[57] Mark Russinovich and Zary Segall, *Fault-Tolerance for Off-The-Shelf Applications and Hardware*, Digest of Papers: The Twenty-Fifth International Symposium on Fault-Tolerant Computing, Pasadena, CA, June 27–30, 1995, pp. 67–71.

[58] Jean Arlat, et al, *Fault Injection for Dependability Validation: A Methodology and Some Applications*, IEEE Transactions on Software Engineering, Vol. 16, No. 2, February 1990, pp. 166–182.

[59] Ming-Yee Lai and Steve Y. Wang, *Software Fault Insertion Testing for Fault Tolerance*, in Software Fault Tolerance, Michael R. Lyu, editor, John Wiley & Sons, 1995, pp. 315–333.

[60] Wei-lun Kao, et al, *FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults*, IEEE Transactions on Software Engineering, Vol. 19, No. 11, November 1993, pp. 1105–1118.

[61] J. C. Fabre, et al, *Assessment of COTS Microkernels by Fault Injection*, Proceedings IFIP DCCA-7, 1999, pp. 19–38.

[62] Dimitri Avresky, et al, *Fault Injection for the Formal Testing of Fault Tolerance*, Digest of Papers of the Twenty-Second In-

ternational Symposium on Fault-Tolerant Computing, Boston, Massachusetts, July 8–10, 1992, pp. 345–354.

[63] Ravishankar K. Iyer and Dong Tang, *Experimental Analysis of Computer System Dependability*, in Fault Tolerant Computer System Design, Dhiraj K. Pradhan, Prentice Hall, 1996, pp. 282–392.

[64] Inhwan Lee and Ravishankar K. Iyer, *Software Dependability in the Tandem GUARDIAN System*, IEEE Transactions on Software Engineering, Vol. 21, No. 5, May 1995, pp. 455–467.

[65] Jeffrey M. Voas and Gary McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, John Wiley & Sons, Inc., 1998.

[66] N. Suri, et al, *Advances in Ultra-dependable Distributed Systems*, IEEE Computer Society Press, 1995.

[67] Brian Randell, et al, editors, *Predictably Dependable Computing Systems*, Springer, 1995.

[68] Barry W. Johnson, *An Introduction to the Design and Analysis of Fault-Tolerant Systems*, in Fault-Tolerant Computer System Design, Dhiraj K. Pradhan, Prentice Hall, Inc., 1996, pp. 1–87.

[69] Nancy G. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[70] Algirdas Avizienis, *A Design Paradigm for Fault Tolerant Systems*, Proceedings of the AIAA/IEEE Digital Avionics Systems Conference (DASC), Washington, D.C., 1987.

[71] Victor P. Nelson, *Fault-Tolerant Computing: Fundamental Concepts*, IEEE Computer, July 1990, pp. 19–25.

[72] Stephen B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice Hall, 1995.

[73] Jaynarayan H. Lala and Richard E. Harper, *Architectural Principles for Safety-Critical Real-Time Applications*, Proceedings of the IEEE, Vol. 82, No. 1, January 1994, pp. 25–40.

[74] Timothy C. K. Chou, *Beyond Fault Tolerance*, IEEE Computer, April 1997, pp. 47–49.

[75] Charles B. Weinstock and David P. Gluch, *A Perspective on the State of Research in Fault-Tolerant Systems*, Software Engineering Institute, Special Report CMU/SEI-97-SR-008, June 1997.

[76] Wilfredo Torres-Pomales, NASA Report (No. L-18034) on Software Fault Tolerance, 2000.

[77] Brian D. Aleska and Joseph P. Carter, *Boeing 777 Airplane Information Management System Operational Experience*, AIAA/IEEE Digital Avionics Systems Conference, Vol. II, 1997, pp. 3.1-21 – 3.1-27.

[78] Robert J. Bleeg, *Commercial Jet Transport Fly-By-Wire Architecture Considerations*, AIAA/IEEE 8^{th} Digital Avionics Systems Conference, October 1988, pp. 399–406.

[79] Andy D. Hills and Dr. Nisar A. Mirza, *Fault Tolerant Avionics*, AIAA/IEEE 8^{th} Digital Avionics Systems Conference, October 1988, pp. 407–414.

[80] Gordon McKinzie, *Summing Up the 777's First Year: Is This a Great Airplane, or What?*, Airliner, July – September 1996, pp. 22–25.

[81] Y.C. Yeh, *Triple-Triple Redundant 777 Primary Flight Computer*, Proceedings of the 1996 IEEE Aerospace Applications Conference, Vol. 1, 1996, pp. 293–307.

[82] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz, *The Real-Time Operating System of MARS*, ACM SIGOPS Operating Systems Review, Vol. 23, No. 3, July 1989, pp. 141–157.

[83] H. Kopetz, A. Damm, Ch. Koza, M. Mulazzani, W. Schwabl, Ch. Senft and R. Zainlinger, *Distributed Fault-Tolerant Real-Time Systems: The MARS Approach*, IEEE Micro, Vol. 9, No. 1, February 1989, pp. 25–40.

[84] Hua Zhong and Jason Nieh, *CRAK: Linux Checkpoint / Restart As a Kernel Module*, Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2002.

[85] E. Roman, *A Survey of Checkpoint/ Restart Implementations*, Lawrence Berkeley National Laboratory – Checkpoint Survey Report, July 2002.

[86] J. Duell, P. Hargrove and E. Roman, *Requirements for Linux Checkpoint/ Restart*, Report of the Lawrence Berkeley National Laboratory, 2002.

[87] Dominique Briere and Pascal Traverse, *AIRBUS A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems*, Digest of Papers FTCS-23: The Twenty-Third International Symposium on Fault-Tolerant Computing, June 1993, pp. 616–623.

[88] Pascal Traverse, *Dependability of Digital Computers on Board Airplanes*, Dependable Computing for Critical Applications, Volume 4, A. Avizienis, J.C. Laprie, editors, 1991, pp. 134–152.

[89] E. Rotenberg, *Ar-smt: Coarse-grain time redundancy for high performance general purpose processors*, Univ. of Wisc. Course Project (ECE753), May 1998.

[90] E. Rotenberg, Q. Jacobson, Y. Sazeides and J. Smith, *Trace Processors*, Proc. 30th Intl. Symp. On Microarchitecture, December 1997.

[91] E. Rotenberg, *AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors*, Technical Report of the Univ. of Wisconsin – Madison, 1998.

[92] J.H. Patel and L.Y. Fung, *Concurrent error detection in ALU's by recomputing with shifted operands*, IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982, pp. 589–595.

[93] B.W. Johnson, *Fault-Tolerant Microprocessor–Based Systems*, IEEE Micro, December 1984, pp. 1609–1624.

[94] D. Tullsen, S. Eggers and H. Levy, *Simultaneous Multithreading: Maximizing on-chip parallelism*, Proc. of the 22^{nd} Intl. Symp. On Computer Architecture, June 1995, pp. 392–403.

[95] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo and R. Stamm, *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor*, Proc. of the 23^{rd} Intl. Symp. On Computer Architecture, May 1996, pp. 191–202.

[96] A. Damm, *The Effectiveness of Software Error-Detection Mechanisms in Real-Time Operating Systems*, Proc of the 16^{th} Intl. Symp. On Fault Tolerant Computing, Vienna, July 1986, pp.171–176.

[97] A. Damm, *Experimental Evaluation of Error-Detection and Self-Checking Coverage of Components of a Distributed Real-Time System*, PhD Thesis, Technisch Naturwissenschaftliche Fakultat, Technische Universitat Wien, Vienna, Austria, 1988.

[98] A. Damm, *Self-Checking Coverage of Components of a Distributed Real-Time System*, Proc. of the 4^{th} Intl. Conf. On Fault-Tolerant Computing Systems, Germany, 1989, pp. 308–319.

[99] T. Sato and I.Arita, *Tolerating Transient Faults in Microprocessors*, Proc. of the 13^{th} Symposium on Parallel Processing, 2001, Japan.

[100] J. Reisinger, *Failure Modes and Failure Characteristics of a TDMA driven Ethernet*, Research Report 8/89, Institut fur Technische Informatik, Technische Universtat Wien, Vienna, Austria, 1989.

[101] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger and L. Alvisi, *Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic*, Proc. of the Intl. Conf. On Dependable Systems and Networks, Maryland, June 2002, pp. 389–398.

[102] D. Milojicic, A. Messer, J. Shau, G. Fu, P. Alto and A. Munoz, *Increasing Relevance of Memory Hardware Errors: a Case for recoverable Programming Models*, Proc. of the ACM SIGOPS European Workshop, Denmark, Sept. 2000, pp. 97–102.

[103] M. Rebaudengo, M.S. Reorda, M. Torchiano and M. Violante, *Soft-Error Detection Through Software Fault Tolerance Techniques*, Proc. of the IEEE Intl. Symp. On Defect and Fault Tolerance in VLSI Systems, New Mexico, Nov. 1999, pp. 210–218.

[104] C. da Lu and D.A. Reed, *Assessing Fault Sensitivity in MPI Applications*, In Supercomputing, Pittsburg, Nov. 2004, pp. 37.

[105] M.Z. Rela, H. Madeira and J.G. Silva, *Experimental Evaluation of the Fail Silent Behavior in Programs with Consistency Checks*, Proc. of the Intl. Symp. On Fault Tolerant Computing, Japan, June 1996, pp. 394–403.

[106] S. Yau and F. Chen, *An Approach to Concurrent Control Flow Checking*, IEEE Transactions on Software Engineering, Vol. 6, No. 2, March 1980, pp. 126–137.

[107] Y. Tamir and E. Gafni, *A Sofware-Based Hardware Fault Tolerance Scheme for Multicomputers*, Proc. of the Intl. Conf. on Parallel Processing, Illinois, August 1987, pp.117–120.

[108] Hoang Pham (Ed), *Fault- Tolerant Software Systems*, IEEE Computer Society Press, 1992.

[109] Kuang-Hua Huang and Jacob A. Abraham, *Algorithm-Based Fault Tolerancefor Matrix Operations*, IEEE Transactions on Computers, Vol. C-33, No. 6, June 1986, pp. 518–528.

[110] Gam D. Nguyen, *Error- Detection Codes: Algorithms and Fast Implementation*, IEEE Transanctions on Computers, Vol. 54, No.1, January 2005, pp. 1–11.

[111] E. W. Dijkstra, *Self-stabilizing systems in spite of distributed control*, Communications of the ACM, Vol.17, No.11, November 1974, pp.643–644.

[112] L. Lamport, *Solved problems, unsolved problems, and non-problems in concurrency*, In Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, 1984, pages 1–11.

[113] M. Schneider, *Self-stabilization*, ACM Computing Surveys, Vol.25, No.1, March 1993, pp.45–67.

[114] T. Herman, *A comprehensive bibliograph on self-stabilization, a working paper*, Chicago J. Theoretical Comput. Sci., http://www.cs.uiowa.edu/ftp/selfstab/bibliography.

[115] W. Goddard, S.T. Hedetmiemi, D.P. Jacobs and P.K. Srimani, *Self-Stabilizing Distributed Algorithm for Strong Matching in a System Graph*, Proc. of the High Performance Computing, Hyderabad, 2003.

[116] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[117] J. Ray, J.C. Hoe and B. Faisafi, *Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery*, IEEE Micro, 2001.

[118] Goutam Kumar Saha, *Algorithm Based EFT Errors Detection in Matrix Arrays*, Internatinal Journal System Analysis Modelling Simulation, Gordon and Breach, Vol.36, No. 1, USA, 1999, pp.117–135.

[119] G.K. Saha, *Fault Tolerant Computing: A Self-Detection & Recovery Technique*, Internatinal Journal Computers & Electrical Engineering, Elsevier Science Pub., UK, Vol.24, No.5, 1998.

[120] Goutam K Saha, *Transient- Fault Tolerant Processing in a RF Application*, Internatinal Journal System Analysis Modelling Simulation, Gordon and Breach, USA, Vol.38, 2000, pp.81–93.

[121] Goutam K Saha, *EMI Control by Software for a RF Communication System*, in Book: Electromagnetic Environments and Consequences, Part II, Edited by D.J. Serafin, EUROEM, France, 1995, pp.1870–1875.

[122] Goutam Kumar Saha, *A Software Tool for Fault Tolerance*, accepted & in press, Internatinal Journal – Information Science & Engineering, 2005.

[123] Goutam K Saha, *Algorithm Based Fault Tolerant Computing for a Scientific Application*, Internatinal Journal System Analysis Modelling Simulation, Gordon and Breach, USA, Vol.34, No.4, 1999, pp.509–523.

[124] Goutam K Saha, *EMP- Fault Tolerant Computing: A New Approach*, Internatinal Journal of Microelectronic Systems Integration, Plenum Publishing Corporation, USA, Vol.5, No.3, 1997, pp.183–193.

[125] G.K. Saha, *Designing an EMI Immune Software for Microprocessor Based Traffic Control System*, Proc. 11th IEEE Internatinal Symposium EMC'95, Zurich, 1995, pp.401–404.

[126] Goutam Kumar Saha, *Fault Tolerant Processing Technique for a Temperature Measurement System*, in Desk Book: Industrial Measurements & Automation, TIMA'96- American Society of Instrumentation, HCK Publication, Madras, pp.56–58.

[127] G.K. Saha, *Virtual N-Versions Programming for Fault Tolerant Computing*, Internatinal Journal Computers & Electrical Engineering, Elsevier Science Pub., UK, Vol.24, No.4, 1999.

[128] G.K. Saha, *Using Software to Control ESD/EMP in Microprocessor-Based System*, RF Design (EMC Test & Design), Argus Inc., USA, November 1995, pp. 8-11.

[129] G.K. Saha, *EMI Protection by Software for a Microcomputer Based Process Controller*, Proc. IEEE sponsored Symp. ISEMC'94, SP Brazil, 1994.

[130] Goutam Kumar Saha, *Transient Software Fault Tolerance Through Recovery*, ACM Ubiquity, ACM Press, Vol. 4, No. 29, USA, 2003.

[131] Goutam Kumar Saha, *Fault Tolerance in Distributed System*, CSI HardCopy, Vol. 40, December 2003, Kolkata, Computer Society of India.

[132] Goutam Kumar Saha, *Single-Version Software Fault Tolerance in Process Control System*, Proc of the Modelling & Simulation, MS'2004, AMSE Press, France, 2004.

[133] Goutam Kumar Saha, *Beyond the Conventional Techniques of Software Fault Tolerance*, ACM Ubiquity, ACM Press, Vol. 4, No. 47, USA, 2004.

[134] Goutam Kumar Saha, *Fault Management in Mobile Computing*, ACM Ubiquity, ACM Press, Vol. 4, No. 32, USA, 2003.

[135] Goutam Kumar Saha, *Software Fault Tolerance in Industrial Automation*, Journal Industrial Automation, IED, Mumbai, April 2004.

[136] Goutam Kumar Saha, *A Software Fix Towards Fault Tolerant Computing*, ACM Ubiquity, ACM Press, Vol. 6, No. 16, USA, May 2005.

[137] Goutam Kumar Saha, *A Technique of Designing in Application System with Fault Tolerance*, Journal of the AMSE, France, December 2004.

[138] Goutam Kumar Saha, *Fault Tolerance Application Using a Multiprocessor*, to appear in IEEE Potentials, 2005.

[139] Goutam Kumar Saha, *Transient Software Fault Tolerance Using Single Version Algorithm*, accepted paper, ACM Ubiquity, ACM Press, 2005.

[140] Goutam Kumar Saha, *Software Implemented Fault Tolerance – The ESVP Approach*, to appear in IEEE Potentials, 2005.

G.K. Saha,

Received July 12, 2005

Centre for Development of Advanced Computing,
Kolkata
Mailing Address: CA- 2 / 4B, Baguitai, Deshbandhu Nagar,
Kolkata-700059, INDIA
E–mail: *gksaha@rediffmail.com*