U.S. DEPARTMENT OF COMMERCE • National Bureau of Standards

NBSIR 87-3568

Institute for Computer Sciences and Technology

On the Measurement of Fault-Tolerant Parallel Processors

John W. Roberts
Alan Mink
Robert J. Carpenter

Advanced Systems Division

May 1987

CMRF

COMPUTER MEASUREMENT RESEARCH FACILITY FOR HIGH PERFORMANCE PARALLEL COMPUTATION

Sponsored by the
Defense Advanced Research Projects Agency
under ARPA order number 5520,
July 23, 1985, and July 23, 1986.

ON THE MEASUREMENT OF FAULT-TOLERANT PARALLEL PROCESSORS

John W. Roberts
Alan Mink
Robert J. Carpenter

Advanced Systems Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Gaithersburg, MD 20899

Preparation of this report was sponsored by the
Strategic Computing Initiative
Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209
ARPA Order No. 5520, July 23, 1985,
as amended July 28, 1986.

U.S. Department of Commerce, Malcolm Baldrige, Secretary

National Bureau of Standards, Ernest Ambler, Director

May 1987

TABLE OF CONTENTS

ON THE MEASUREMENT OF FAULT-TOLERANT PARALLEL PROCESSORS

John W. Roberts
Alan Mink
Robert J. Carpenter

A number of measurement techniques can be used to determine how well computers detect and recover from faults. In addition to qualitative measures, quantitative measures can be obtained relating to fraction of faults detected and corrected, recovery time, and degradation of performance during and after recovery.

Key words: Computers; Fault detection effectiveness; Fault recovery effectiveness; Fault tolerant; Performance measurement.

1. Background

Fault tolerance in a computer system is the ability to detect erroneous states in computations or in hardware, and to deal with these errors so that "correct" operation can continue. While limited capability for error detection and correction is commonplace, a much smaller set of computer systems detects and correctly handles errors with a high degree of assurance. This smaller set, known as fault-tolerant systems, applies various techniques to meet the specialized needs of a wide range of users.

1.1 The need for fault-tolerant processing

The principal need for fault tolerance arises in the areas of the solution of large problems, control systems demanding high reliability, and applications demanding availability.

Large Scale applications are those which use enormous amounts of computation (e.g., weather forecasting and three-dimensional fluid flows), and thus require long run times. A fault-tolerant system which is good at both detecting and recovering from errors is virtually a necessity for the solution of large-scale problems that have long running times, with some assurance that the results are correct. As an example, consider a system which has a normal error rate of one per billion operations. If an attempt is made to run a program requiring one hundred billion operations on this machine, the results are almost sure to be incorrect. Comparison of the results from multiple runs can show errors, but can not be used to determine which are the correct results unless the program is run many times. In this case, it is important not only that the system be very good at detecting errors, but also that it be able to continue operation after the detection of errors, without having to restart programs from the beginning.

High-reliability applications are those for which it is important that the system remain functional as long as possible in the presence of hardware failures. These applications include manufacturing process controllers, and aircraft and spacecraft control systems. For some of the functions of these systems, such as the processing of incoming sensor information, a loss of small amounts of incoming data may not be harmful as long as overall operation is able to continue. For these uses, mean time between system failures (MTBF) and mean time to next failure (MTTF) are the parameters of greatest importance.

High-availability systems, such as telephone switching centers, strive to compensate for hardware failures in order to minimize the fraction of the time that the system is unavailable because it is awaiting repairs or engaged in the fault recovery process. The parameters of interest are mean time between failures and mean recovery/repair time. Reduced-level performance may be available during recovery, and if so should be characterized. Such systems feature redundancy of many or all components, and may allow on-line repair or replacement of failed components.

The original design of any particular fault-tolerant system determines to what degree it incorporates error detection and recovery facilities. Since fault tolerance always exacts a cost in price or performance, a potential user should search for a system that conforms adequately to the needs of the intended tasks. If applications that do not require fault tolerance are also targeted for such an architecture, estimates should be made of the penalties (cost, performance, etc.) that may be incurred. Development of measurement techniques to determine the performance of fault-tolerant systems will aid users in this search, and also help manufacturers to categorize their machines in a uniform manner.

1.2 Approaches to achieve fault tolerance

Current approaches to fault-tolerant computing are based on redundancy. Redundancy allows detection of malfunctions (physical errors), but usually cannot detect design errors, which are replicated in each redundant component. Malfunctions are assumed to occur in some random manner not affecting all copies. Redundancy is not a viable approach to detection of software faults, since software logic faults will exist in each of the duplicate units. Most software reliability efforts are directed toward fault avoidance, concentrating on aspects of design and implementation (e.g., design reviews, design specifications, testing, etc.). Detection and correction of software errors at execution time doesn't enter this fault avoidance model because machine operation (at software design time) is assumed correct.

One approach to fault tolerance which *does* address design errors, and therefore includes software errors, is based on diversity [AVIZ84]. For diversity in software, independent organizations implement similar but distinct versions of the software from the same specification. Both versions then execute simultaneously while their outputs are compared. A software fault (in specification, design or implementation) is indicated if the outputs differ. This approach does not guarantee that both versions will not produce identical erroneous results.

At present, a major problem in fault-tolerant systems centers about the decision algorithm which is responsible for the selection of the correct output (if any). In the space shuttle [ZORP85], redundancy is used to detect malfunctions in the four primary processing units, while diversity is used in the fifth, secondary processing unit (which has the same hardware design, but a different software design), as a software consistency check on the primaries.

1.2.1 The assumptions of this report. Henceforth in this paper, *fault-tolerant computing* is understood to refer to the detection and correction of hardware faults (physical failures), rather than faults caused by mistakes in logical or software design. Our baseline model of fault-free operation is a program (either correct or incorrect) and computer hardware operating correctly to execute that program.

1.2.2 Loosely-coupled fault-tolerant architectures. Loosely-coupled fault-tolerant architectures are generally made by coupling a number of fairly conventional complete computers into a system. Alternate communication paths must be provided between all members of the systems, and the individual computer designs must be enhanced to include some means to detect erroneous operation and localize its effects. Communications between members of the system are handled by *messages*, with a protocol to detect any errors in the exchange of messages. The responsibility for managing the recovery from faults in the computers is assigned to the software system. Figure 1 illustrates such a system.

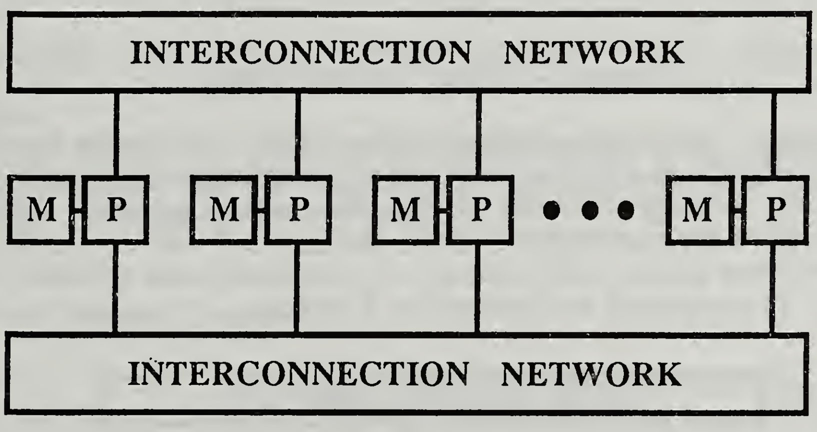

Figure 1 - Loosely-coupled system.

1.2.3 Tightly-coupled fault-tolerant architectures. Tightly-coupled systems require specially-designed computing subunits. Shared memory is used, and the hardware must be designed to avoid erroneous transfers to memory. These systems frequently use multiply redundant processors operating in a lock-step mode, with some sort of voting or selection technique to assure real-time selection of the correct data. Dual *mirrored* memory units are used with special schemes to assure that the correct data has been transferred. A mirrored memory system uses two (or more) copies of each memory location for redundancy. Data must be correctly altered in one instance of the memory before the same change can be allowed in the other (*mirror*) instance(s). Error-checking is again used when altering the second instance. Figure 2 illustrates such a system.

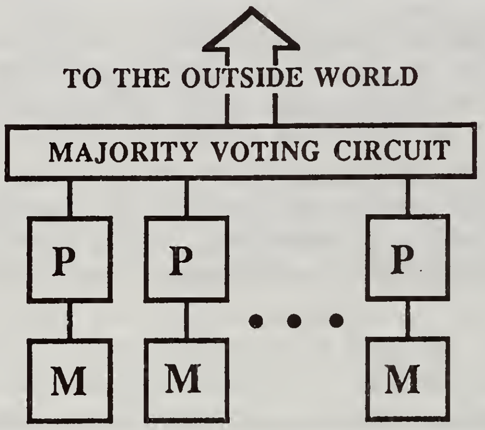

Figure 2 - Tightly-coupled system.

1.2.4 Mixed systems. Some manufacturers choose tight coupling to detect faults, but omit the greater redundancy required for hardware correction of the faults. These systems use recovery techniques similar to loosely-coupled systems. They therefore represent a mixture of the architectures shown in Figures 1 and 2. (Doubly) redundant units are used to detect faults. Additional sets of components are included to allow migration of tasks. This approach is illustrated in Figure 3.

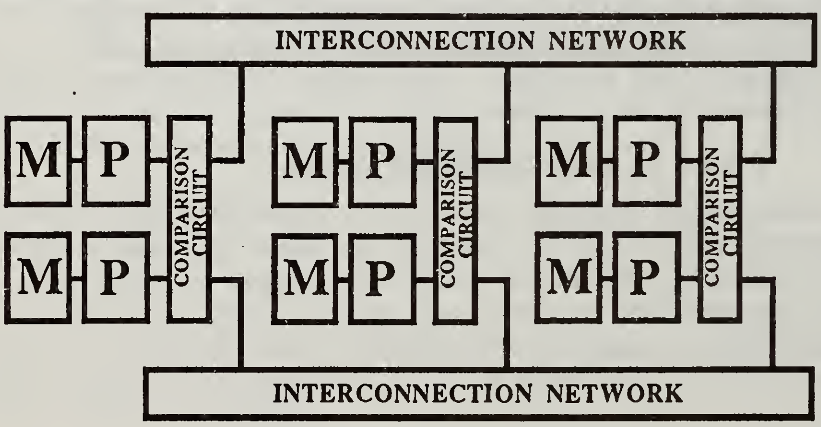

Figure 3 - Mixed-architecture system

1.3 Stages of Fault-tolerant Operations

For the purposes of this report, we have divided fault-tolerant operations into two stages: detection and recovery. Other taxonomies, such as [SIEW84], have chosen a finer division:

- **confinement** - the system attempts to localize the effects of faults so that recovery will be possible.
- **detection** - the faulty operation is detected.
- **masking** - operations are performed redundantly, to detect and mask erroneous results.
- **retry** - errors are detected and operations retried until an error-free result is obtained.
- **diagnosis** - the specific fault involved is diagnosed to allow reconfiguration or repair.
- **reconfiguration** - required (automatic) reconfiguration is used to eliminate the faulty portion of the hardware.
- **recovery** - the state of the machine is backed up to that most recently recorded before the error, and the intervening operations are repeated. This may require the correction of erroneous intermediate results.
- **restart** - the system completely restarts the program from the beginning. This is a drastic recovery technique suitable for batch or transaction systems.
- **repair** - since fault-tolerant machines can only tolerate a limited number of simultaneously defective parts, any defective parts are repaired or replaced automatically or by service personnel before the tolerance limit is exceeded.
- **reintegration** - once a part in a high-availability system is repaired or replaced, it must be reintegrated into the system without disturbing the jobs in progress.
- **reporting** - after a fault is detected, a report is made to the user both to help in evaluation of the fault handling of the system and to alert the user to losses of redundancy in the system.

1.4 I/O in Fault-tolerant Systems

The protection of I/O (input/output) operations poses several unique problems in the design of fault-tolerant systems. These difficulties are due largely to the differences between I/O and other types of data transfers.

The responsibilities of a fault-tolerant I/O system are related to what the designer considers the nature and extent of such a system to be. Communications with other devices and the interface with the users would normally be considered part of I/O. If the fault-tolerant system is made up of a group of loosely-coupled computers, however, communications between the computers within the system would probably not be considered I/O. Similarly, some would consider disk and tape access as I/O, while others would not. One possible working definition is that an interaction between two integrated, self-sufficient entities would be classified as I/O, while interactions within such an entity would not. This definition fits well with the concepts of fault tolerance, since a fault-tolerant system maintains control over a specific area in a community of computational devices, within which it is responsible for detection and correction of faults. The fault-tolerant system is expected to interact with other systems in a correct manner, but it is not responsible for the actions of the other systems. Using this definition, one may be able to visualize a machine designed as a hierarchy of fault-tolerant systems within fault-tolerant systems, with the protected, internal communications at one level regarded as I/O interactions at a lower level.

In general, I/O operations are regarded as the transfer of information in the form of messages in accordance with a specific protocol, which may be implemented chiefly in hardware, but which is usually controlled by a fairly complex software structure. If there is to be any assurance of reliable operation, some handshaking mechanism, which uses attention and acknowledgement signals sent in both directions, must be employed. The capabilities of the handshaking algorithm may vary greatly. A minimal algorithm might simply indicate readiness to receive transfers, with little or no checking of the correctness of the data. A more complex algorithm might check for errors and require retransmissions as needed, but not allow later revisions of transfers or provide highly reliable acknowledgements. Advanced algorithms might provide for the same degree of controllability as is found for intrasystem transfers, at which point the device with which communications are established could be considered part of the fault-tolerant domain.

Many potential problems with I/O transfers are associated with timing and synchronization. Depending on the transfer protocol and the output device, it may not be possible to reverse the effects of an erroneous output. An input which is lost because of an error or because the receiving device is busy recovering from a fault when the signal comes in may be lost permanently. In contention among several devices for a resource, an accurate record of the current owner may be lost, resulting in overwrite errors or blocking of access to the resource. Damage to software reference tables may cause errors in connections. Consistency, which refers to the maintenance of correct records of resource conditions from the viewpoint of all interested parties, is likely to be compromised by such failures. In order to have reliable I/O operations, a fault-tolerant system must have carefully chosen hardware and software structures that take these problems into account. A sophisticated handshaking algorithm is necessary to insure that data transfers can take place without causing errors in either communicating device.

Other than the unique problems associated with timing and controllability, fault tolerance in I/O transfers bears many similarities to fault tolerance in other operations of the system. Much of what is described for fault detection and recovery for internal operations therefore applies to I/O as well. Issues specific to I/O are also mentioned elsewhere, though a complete description of the issues in I/O fault tolerance is beyond the scope of this paper.

2. Fault Detection Measurement Techniques

The first step in determining the capabilities of a fault-tolerant computing system is to evaluate its fault detection processes. The measurement equipment should be able to determine when faults are detected, and whether or not all significant faults are detected. In order to fully evaluate the detection process, the measurement equipment must independently detect the occurrence of all faults and also observe the detection of faults by the system. Many of the details of the measurement techniques are dependent on the architecture of the system under test.

2.1 Detection of errors

A fundamental problem in the design of fault-tolerant systems is to allow them to detect errors while the system is performing useful work. Different error detection techniques are required for the storage and transmission of data, and for the transformation of data.

2.1.1 Errors in the storage and transmission of data. Since information is not intentionally changed, detection of errors in storage and transmission of data is usually handled by information redundancy. This usually takes the form of error codes which come in a large range of complexity, from simple parity and checksums to Hamming codes, AN codes and polynomial codes (e.g., CRC). These codes can be used to allow all errors of up to **n** bits to be detected, and all errors of up to **k** bits to be corrected, where n > k and k >= 0. These techniques have been successfully applied to storage devices (e.g., parity for detection or one-bit error correction in primary memories and magnetic tape units) and data transmission mechanisms (e.g., checksums and CRC appended to data packets). While many systems use redundant information for (forward) error correction, some system use the cheaper but slower approach of *retransmission* for correction of data transmission errors.

2.1.2 Errors in the transformation of data. Detection of errors in the portions of a computer system which intentionally transform data is more difficult. This activity is usually associated with the processing units. Since the data is being transformed, addition of redundant information in the form of error detection codes is not generally applicable. The transformation of the error code information does not produce the correct new code to accompany the transformed data. A new error code must be computed from the transformed data, after it is transformed (correctly or incorrectly). Error detection for transformations is usually handled by redundancy (replication) of data transformation devices. With the assumption that errors are caused by random and independent events, multiple components are highly unlikely to be affected in exactly the same **manner** at the same **time**. Errors are detected by comparison, which requires only two units (more can be used), whose outputs are compared. An error is detected when there is a difference between the outputs of the the units. If only two units are used, the correct output (if any) is not known and further action must be taken to resolve the problem. If three or more units are compared, majority voting may be used to determine the correct output, as discussed more fully below.

2.1.3 Detection of only major failures. If the fault detection is needed only to identify units which have been incapacitated by a catastrophic hard failure (rather than faults which produce erroneous results but allow the unit to continue operation), a simplified detection approach can be taken, based on timeouts [SERL84, ZORP85]. A common technique is for each operating unit (or program) to periodically inform another unit that it is still active. This other unit may be a backup unit or a dedicated monitoring device. If an excessive interval elapses without receipt of this notification, it is assumed that the unit (or program) has failed and a backup strategy is invoked.

2.2 Induced Faults for Testing

In addition to being difficult to implement in a fairly general manner, techniques that make use of only naturally-occurring faults are unsatisfactory as methods for evaluation of the fault detection capabilities of a system. The main drawback lies in the low probability of encountering an adequate number of natural faults of a specific type during the course of the measurement process. Fault-detection tests would therefore take an unreasonably long time to complete if measurement depended on natural faults. This can be avoided if the measurement system, under user supervision, is somehow able to induce, inject, or emulate a comprehensive selection of faults in the system under test. The frequency of occurrence of normally-rare faults can be made high enough to allow the collection of reasonable data on detection and recovery. Knowing when a fault is present is vital to permit the measurement equipment to determine whether the fault is detected by the system being tested.

A drawback of the induced fault method compared to the use of naturally-occurring faults is that there may be little correlation between the distribution and characteristics of the induced faults and naturally occurring ones. This problem can be minimized if an effort is made using theory and experiment to characterize natural faults, and to emulate them with induced faults. Induced faults may also be used to force situations that are considered extremely unlikely but extremely troublesome, to determine whether detection and recovery can take place in these circumstances.

A further difficulty with induced faults is the need to get access to a desired location in the system under test and emulate the desired fault. The techniques available are highly dependent on the architecture of the system tested, and it may not be possible to induce a particular fault. Most of the fault injection techniques used to date involve permanently or temporarily forcing one or a set of signal lines to a fixed logic level, forcing switches open or closed, disabling logic elements or check bit generators, and creating unpredictable faults by putting signal spikes on power supply lines, etc. Faults induced using these methods may bear little resemblance in their characteristics to naturally-occurring faults.

2.3 Simulation vs. Emulation of Faults

It is not always necessary to induce a real fault. If a false "fault-detected" signal can be introduced into the system, many of the fault isolation and recovery mechanisms can be observed. Such signals may be injected, using hardware drivers, in the control circuitry somewhere above the lowest level of system hardware. It may also be possible to induce these signals using software techniques. High-level error reporting should then occur as expected. Systems which recover by switching out suspected components and later testing them for readmission can be thereby be observed in operation. For the most realistic models of recovery from normal faults, however, it is best to use real faults, whether natural or induced. Real faults must also be used to investigate fault detection and correction at the lowest hardware levels of the machine.

When a fault has been introduced, it is necessary to measure its exact extent and the time of its detection. One must observe the faults and notifications of faults as they appear.

2.4 Direct Observation of Faults and Fault Detection

Observation of the detection of faults is simplest if the system under test provides notification to the outside world whenever a fault has been detected. Many systems, however, are not set up to report faults from which the system recovers without loss of performance (i.e. error correction in memories). Even if notification is available, the type and chronology of the fault may not be reported completely enough for certain measurements, for example collection of statistics on address transformation errors. For many measurements, more direct methods of observation of fault detection are desirable.

A more difficult technique for observing fault detection is to locate internal "fault detected" signals within the system being measured. If the architecture of the particular machine under test permits, probes can be connected to the lines on which these signals are found, and the positive indications recorded. This has the advantage of allowing one to learn of fault detections that would otherwise never be reported outside of the system, and to be more selective in the types of faults reported. One major disadvantage is that intrusive examination of a system is highly dependent on its architecture; thus certain observations may prove to be difficult or impossible. In addition, the desired signal may not exist as such. For instance, a fault-tolerant memory controller may algorithmically correct a single-bit read error without any indication that a fault was detected.

Some types of induced signals, particularly noise on power supplies, create unpredictable faults. An inherent weakness of the above methods is that they rely on fault detection in the system under test itself to identify the types of faults detected. This detection facility may not be perfect, and is itself the subject of many important measurements. The most direct way around this problem is to independently look for faults within the system under test by probing address and data lines, looking at control signals, etc. With compact design and large-scale integration, this approach ranges from difficult to impossible in the general case, but there are some some machines for which it is practical. Increased concern over the importance of testing and measurement could prompt designers of future systems to add features to facilitate direct observation of internal system operation.

2.5 Indirect Methods of Observation

Often it is desirable to observe machine performance for a particular type of fault detection, but no direct method is practical, because of the difficulty of gaining access to the internal components of the system under test. Fortunately, one may still be able to obtain useful results using indirect methods that provide only partial information on the internal functions of the system. These methods include diagnostics along with knowledge of hardware failures, and mathematical modeling.

Diagnostics in this case are user-specified test programs run as jobs on the system being tested, for which the correct results are already known, and designed specifically to isolate permanent hardware failures that result in faults. They may attempt to approximate the programming environment that would be encountered in normal use, or may make heavy use of a specific system component being tested. A given diagnostic program may be used, for instance, to test the arithmetic unit of a processor or the integrity of a local memory. If fault detection and correction are functioning as specified, the only faults thus located may be those which are not detected or reported to the user by the system being tested. In many systems, however, it may be possible for the user to disable fault detection or recovery, thus allowing diagnostic programs to identify nearly all transient and permanent hardware faults of a given type. With this knowledge, the user may be able to run more common programs and obtain reasonable measures for fault detection and correction parameters.

A major difficulty with the use of diagnostics both by system being tested and by the measurement equipment is the response of the diagnostics to transient failures. While permanent failures can remain stable long enough to be detected and often located by the diagnostic programs, this is often not the case for transient failures. A transient failure may appear and cause an error, then disappear before it can be located by the recovery diagnostics. It may appear or disappear during or after the running of a diagnostic program, rendering its conclusions invalid. Since permanent failures do remain to be detected, however, there are many situations in which it is reasonable for the system or the measurement equipment to use diagnostic programs for the detection of hardware failures, as long as the difficulties that can be caused by transient failures are taken into account.

Where it is not possible to obtain complete information on a needed parameter, the application of mathematical models to the information at hand may make it possible to produce a reasonable reconstruction of the data. For instance, suppose the measurement equipment can detect faults of hypothetical type A but not of type B. Mathematical analysis, taking into consideration the architecture of the system under test, suggests that there should be a correlation between faults of types A and B. Tests may be run in order to seek supporting evidence for the correctness of this model. One may then observe the rate, grouping, etc. of faults of type A, and produce a reasonable estimate of the characteristics of faults of type B. It is important to keep in mind that there may be significant factors not taken into account by the model, and that this limits the accuracy or validity of the conclusions that may be drawn. A specific example is an address bus for which faults can be detected only in the upper bits of the address. Since all the address bit signals go through many identical processes, it may be reasonable to assume that the bit error rate on the lower bits is not greatly different from the rate observed on the upper bits. It is necessary to take into account, however, the facts that permanent hardware failures are not necessarily evenly distributed and that there are certain operations that the upper and lower address bits do not have in common, such as cache control and memory management, which can reduce the error correlation among the address signals.

If something is known about the timing of execution in the system under normal conditions, it may be possible to put this information to use in the analysis of corrected but unreported faults. While some fault-tolerant systems are designed with tightly-coupled hardware control with the objective of avoiding any delays from fault-correction procedures, many other hardware-based systems and all software-based fault-tolerant systems will experience delays as recovery from a fault takes place, and often exhibit a reduced level of performance during recovery. If the time to complete a task with no errors present is well characterized, and if a good estimate for the recovery time is available, then it may be possible from consistently increased execution time to deduce that a number of errors are being detected and corrected. The practicality of this technique depends on the consistency of the normal speed of execution, the length of time required for recovery, and the knowledge that other factors are *not* interfering with execution. This approach seems precarious at best, but it may be possible to apply it to a wider range of systems than other approaches. In certain systems, access to *HALT* or *WAIT* signals that are associated with detection of an error can make the correctness of interpretation of the measurements more certain.

The observations that can be made and the alternative mechanisms that may be available are highly dependent on the architecture of the specific machine under test. Signals that are buried inside the integrated circuits of one system may be available to the outside world in another. Some machines allow the user to control or inhibit the operation of certain internal functions in order to facilitate testing. Such capability can be extremely useful in testing the various features of a fault-tolerant system independently of one another.

2.6 Observation of Software Fault Detection Techniques

A number of fault-tolerant systems make use of software techniques to detect faults. With this method, different tradeoffs are chosen than with hardware detection systems. Hardware design time and hardware investment may be considerably reduced, but software design is likely to be more complicated, and the time spent running fault detection and recovery routines may cause execution to be slower than for systems which use hardware fault detection. In addition, certain types of faults involving short-term events or the correct execution of routines may be more difficult for the system to detect. Different methods must also be employed to observe the performance of the fault detection mechanisms.

As previously described, the techniques employed for software fault detection include the interchange of signals indicating continued correct operation, periodic synchronization and comparison of intermediate results and/or processor state, and comparison of multiple copies of global variables. In many systems hardware for the detection of faults is also under the control of supervisory processes, such as schedulers which assign multiple copies of tasks to different processors. To varying degrees, these software functions can be controlled by the user. The user may therefore have the opportunity to override the normal operations of the fault detection software in order to better observe the performance of its individual parts.

Since the fault detection process in these systems proceeds at the speed of program execution, observation may proceed at a similar pace. Reporting of faults may very well be one of the functions of the software, or the user may be able to include real-time notification or logging of faults with only a slight penalty to the rate of execution of the fault-tolerant software. If hardware-level observation of software-detected faults is desirable and practical, it is useful to note that fault detection is a major event from the viewpoint of the processor, with signals on address, data, and control lines, which in many systems can be monitored by measurement equipment.

As in the case of hardware fault-detection systems, it is necessary for the measurement hardware to be made aware of faults as they occur, in some way that is independent of system fault detection, in order to be able to evaluate the fault detection mechanism itself. Techniques that rely on hardware fault reporting will not work for software-based systems, because the signals of interest do not exist. To obtain a reasonable estimate of the fault rate, it may be necessary to use the indirect methods previously described, or, preferably, to carefully control the induced faults.

For systems with software-based fault detection, there are several software-oriented options available for induction of faults. It may be possible to make a change in the value of a reference variable that will be interpreted as a fault. Forcing suspension or termination of one of a set of redundant tasks would have a similar effect. It may also be possible in such systems to insert error notices at a higher level in the control structure of the software, to test the upper levels of the fault detection and recovery mechanisms.

3. Measurement of Fault Detection

An abstract model of a computing system might consider all activities of a computer system to consist of logical manipulation of data, storage of data, transfer of data within the computing system, and input/output. Fault detection can therefore be considered as it relates to each of these activities. The problems associated with fault tolerance for I/O operations as compared to internal operations are mentioned here and throughout the paper.

3.1 Explanation of measurement entries

This section of the paper is essentially a list of many of the faults to which a fault-tolerant system may be subject, and a description of ways in which the detection of these faults by the system under test might be evaluated. The faults that will be of principal interest to a given user depend on the architecture of the system, the applications that are anticipated, and the user's interpretation of fault tolerance. The techniques to be employed are similarly a function of these factors. Other fault types and evaluation techniques may be added as needed. An attempt has been made to keep the fault categories fairly general, so they can be applied in a wide range of situations.

3.1.1 Sample measurement entry.

Description of fault: This is a collection of general background information concerning a particular type of fault. Among the descriptions that may be included are the precise the nature of the fault, possible causes of the fault, ways in which the fault can effect system operation, and methods that may be used by the fault-tolerant system to detect this type of fault.

Architectures affected: This is a brief summary or description of the system architectures that are susceptible to this type of fault. In many cases a given architecture will be inherently immune to certain faults, and no further consideration is needed for these faults. The range of architectures considered to be affected by a given fault depends on whether the description of the fault is broadly or strictly interpreted, so the user must decide in advance how to define the set of faults, in order to accurately apply these techniques.

Test method: This is a collection of techniques relating to analysis by the measurement equipment of the ability of the system under test to detect faults of this type. In order to perform this function, the measurement equipment must encounter faults, determine that faults of the appropriate type have occurred, derive the needed information concerning the faults, and determine whether the system under test has correctly detected and identified the faults. (Some of the identification activity may be considered part of the recovery process rather than the identification process.) To make sure that errors are encountered, many of the entries include guidelines on possible ways to induce faults. Many fault-injection techniques are not completely controllable, and will produce a range of faults, not just the specific type desired. In any event, it is often necessary for the measurement equipment to directly observe the faults as they appear, in order to determine whether all such faults are detected by the system under test. Where permanent hardware failures are involved, one approach is to run diagnostics to locate the failures, so it is known that a fault will occur whenever a certain operation is attempted. Techniques are given that allow the measurement equipment to determine whether the system under test has detected a fault. If notification is not automatically provided to the outside world, intrusive techniques may be employed.

The principle measurement to be taken for most of these parameters is the incidence of fault detections by the system under test, compared to the incidence of faults as determined by the measurement equipment. This may be presented as a ratio, giving the percentage of faults of this type which the system under test detects or fails to detect (or falsely detects.) It may be more useful, though more complicated, to produce a log describing each detection or failed detection, so that analysis can lead to discovery of specific correctable problems.

There may be some interest in measuring the interval between the appearance of the fault and detection. A useful measurement would be difficult for several reasons. First, the instant at which a fault "occurs" may be subject to considerable debate. Second, the interval to detection will depend mostly on the detection technique employed. For instance, lockstep hardware systems will detect the fault almost immediately if at all. Checkpointing systems may not detect the fault until the next "full" comparison, which may take place over a wide range of intervals from the appearance of the fault. The time to detection may depend only slightly on the type of the fault. Interval measurements are therefore useful mainly as a way of evaluating response characteristics of various detection techniques, and not as a way to characterize performance relative to specific faults.

Special apparatus: This section lists certain types of equipment, including passive probes and various active devices, that may be useful for observation of faults and of fault detection within the system under test.

Limitations of test: There are many situations in which a particular measurement technique may not provide completely satisfactory results. There may be a small known error in the results, or certain types of faults that can not be reliably detected by the measurement equipment. It is important to know the limitations of the tests in order to avoid drawing erroneous conclusions from the results.

3.2 Detection of transmission errors

In the broadest sense, transmission refers to all operations which cause a transfer of data from one location to another with no permanent transformation of the data. This includes all in-system communications, transfers between registers, etc. Since errors in the lowest-level transmissions may not be distinguishable from other circuit errors, there is a tendency to look mainly at higher-level communications in this category.

3.2.1 Inter-module communication errors.

Description of fault: This term refers to errors in the communications links among resource modules and processors in a fault-tolerant system. Such links can be between redundant components jointly performing a fault-tolerant task, or between groups of components performing different parts of a job. If the fault is of a type that prevents normal communications procedures, it should be detected quickly by hardware or software checking. Address errors can often be detected by the handshaking procedure, though certain types of address errors are very difficult to handle, as described in the next entry. Data transmission errors have been widely studied, and may be detected by several means, including the transmission of redundant information.

Architectures affected: The modularity and the communications approach chosen in the design of a particular system determines the types of intermodule communications errors to which it is prone. Systems with separate processor and memory modules will have to deal with memory addressing errors. Module addressing errors can become more of a problem in common-bus systems than in other systems. Links between redundant modules performing identical tasks must allow the modules to compare results effectively without introducing new errors.

Test method: Intermodule errors may be easier to observe than internal communications errors, because there is a better chance that the signal lines will be available for the attachment of probes. In a shared bus or ring system, the signal lines can be checked at several points, to determine whether any errors have been introduced. It may also be possible to induce errors by forcing lines high or low, breaking connections, inverting signals, etc. Direct notification of errors by the system under test is the simplest means of observing fault detections, though these signals may be hard to access.

Special apparatus: Probes to observe communication paths and error signal lines (if available) will be needed. To induce errors by driving the signal lines, appropriate drivers will be required. Care must be taken to insure that the system line drivers are not damaged. Signal modification devices may have to be inserted *in series* with signal lines.

Limitations of test: Evaluation of the detection of all possible types of faults is not practical, even using induced faults. Care must be taken in the selection of a suitable subset of faults to be evaluated. The architecture of the system tested may limit the range and accuracy of the tests that may be performed.

3.2.2 Address errors.
Description of fault: Because of the heavy emphasis that is traditionally placed on errors in storage and transmission of data, there is a tendency in designing a computer system to use extensive error detection and correction for the data, but to use less protection for the address lines and circuitry. This is unfortunate, since address errors can cause a system failure at least as easily as can data errors. Because of the intricate processes of address generation, translation, transmission, and recognition (many of which *transform* the address information), it is much more difficult to fully protect the address handling process than the data process. However, where there is sufficient motive for reliability, it should be possible, though expensive, to create a reasonable degree of protection from address errors.

Address errors may be caused by incorrect address generation in the processor, incorrect translation in the memory management circuitry, errors in transmission, and incorrect interpretation of the addresses in the addressed devices. This class of faults includes incorrect handling of addresses in cache memory.

The main types of addressing errors are: addressing the wrong resource, addressing nonexistent resources, addressing the wrong page, and address errors within a page. Among the in-page errors are: compensating errors, multiple addresses mapped to one location, and one address mapped to multiple locations (unstable address bits). These errors can be caused by faults in the processor or addressing logic, or in the address signal lines.

Faults in resource selection could possibly be detected by fault-tolerant resource/memory management devices, by general-purpose address monitors, or by standard use of handshaking procedures in address and data transfers. Page faults, where a "page" represents a set of addresses logically viewed as a group for purposes of address or memory management, could be detected by memory management devices. Monitors looking at "sequential address" and other flag signals from the processors could detect the specific types of faults to which these signals are related, but since a considerable fraction of the addresses sent out would not be thus protected, such an approach might be considered only marginally helpful in making a system highly fault-tolerant.

Compensating errors are those which cause parts of the system to perform outside of specifications, but which produce no error in results. A common example would be crossed address lines running to a homogeneous random access memory. Though data would be written to and read from the wrong addresses, the results would be correct as seen from outside the memory. Such errors, while not directly causing system failures, can complicate fault detection by producing readings inconsistent with system results and dependent on where the readings are taken.

Failures in which several addresses map to one location cause faults when a value is overwritten by a write to another address. Similar faults appear on reading. Such failures can probably be detected by diagnostics. Faults caused by unstable address lines can damage the contents of several locations, and make the fetching of needed data unreliable.

For highly fault-tolerant designs, methods similar to those used to protect data can be adapted to protect addresses. Two or more processors working in parallel can compare output addresses to detect/correct address errors caused by the processors, as described in the section on processor errors. Address check bits can detect errors in transmission. Redundant address detectors can be used at the intended locations to provide fault-tolerant address decoding. Redundancy would therefore be employed in receivers rather than transmitters, so there would have to be changes in the response algorithm. Such an approach might not be considered worthwhile unless it were part of a complete system of fully redundant resources, with the redundant parts of each resource comparing actions several times in the process of serving the processor.

Architectures affected: Essentially all computer systems use addressing for low-level internal data manipulation. Many systems also use explicit addressing in higher level communications.

Test method: Other than by notification, addressing errors are most likely to be found by incorrect system output, as when user diagnostics are run. Comparison of address lines at source and destination can locate physical failures in the conductors. Induced errors may be used only if the address lines are available for external manipulation without damaging the system under test. Detection can be observed by fault signals or by perceived operation of the recovery system.

Special apparatus: Probes to observe address and fault signal lines may be used. It may be practical to design an address manager emulator and connect it to the processor and address lines, to evaluate the address management of the system. Error signal drivers may be used if possible.

Limitations of test: Many address signals will not be available for outside observation, especially in systems with VLSI components. It may be necessary in some cases to obtain approximate readings using indirect measurement techniques.

3.3 Detection of data storage errors

Storage of data in the interactive parts of the system involves registers as well as local and shared memories. Since different techniques for storage, retrieval, and fault detection are used for each, the different types of data storage are considered separately.

3.3.1 Faults in processor registers, transient.

Description of fault: With the wide range of architectures used in fault-tolerant systems, the term "processor register" can be used to apply to the data, address, and flag registers closely associated with the processing circuitry, as in a microprocessor, or it can be used in reference to flip-flops, state machine registers, and generally all of the temporary storage used by the support circuitry in and around each processing element.

The fault-tolerant system can detect register faults (without necessarily identifying them) by use of hardware redundancy. Repeated tests can determine whether or not a given fault is transient.

If the transient fault is in a register of a conventional microprocessor, evaluation will be very difficult. Incorrect results may be the only sign of such faults, and testing and diagnostics will probably be unable to identify the register as the source of the fault.

If, however, the fault is in a register with built-in fault detection or a register that is directly accessible by the system designer, there is a good chance that such faults can be isolated and specifically dealt with.

Architectures affected: In the general sense, all digital computing systems function as state machines, and require temporary storage for the "machine state". All architectures are therefore susceptible to register faults. Even with a narrowed interpretation of "processor register", the vast majority of machines use registers that fit within the definition.

Test method: For registers not directly accessible to the measurement equipment, evaluation is probably not practical. For registers that are accessible, direct observation will work, but will be awkward for looking at many registers simultaneously. Induced errors can be used to make the pattern of faults more predictable. The error-detected signals in the system under test will provide an indication that the error has been detected.

Special apparatus: Probes into system, error signal drivers.

Limitations of test: It is not practical to individually evaluate many of the registers of interest.

3.3.2 Faults in processor registers, permanent.

Description of fault: The registers of interest are the same as for the transient fault parameter, and similar techniques are used, for the most part. Register faults within a microprocessor are extremely difficult to observe directly. Observation is more practical for registers accessible from the outside. The presumed stability of the hardware failure, however, allows an additional set of techniques for detection and identification. The fault may be detected by physically redundant execution of a task. Temporal redundancy is ineffective as a detection technique in the system under test because the same error can appear each time, so redundancy of hardware is a necessity. Once a fault has been detected, diagnostics should be able to detect consistently erroneous performance of a given processor register, and thus determine the location of the fault.

Architectures affected: As is the case for transient register faults, essentially all machine architectures are affected.

Test method: The techniques used for transient register faults are also applied for permanent faults. Induced errors must remain stable for the duration of the test. User-supplied diagnostics may be used to verify that the faults are present.

Special apparatus: Same as for transient register faults, possibly with different drivers to induce errors.

Limitations of test: Most of the same limitations apply as for transient register faults. The indirect observation of faults internal to microprocessors, etc. is somewhat less difficult.

3.3.3 Errors in main memory and redundant copies.

Description of fault: Since memory elements make up a large percentage of the total logic element count in most systems, there is a good chance that many of the faults that occur in such systems will be in memory. For this reason, most systems have methods to detect and deal with faults in memory. While faults in logic elements, once detected, may be corrected by means of checkpointing or other methods, the permanent loss of the contents of a memory location can result in system failure. It is therefore vital not only to detect memory faults, but to recover from them.

Two major approaches to fault-tolerant memory design are the use of error detecting codes with storage of multiple copies of all data, and the use of error correcting codes. A combination of these methods may also be used. Error detecting codes and error correcting codes add a number of "check bits" to a group of data bits. There may also be a transformation of the data bits, but the net effect is that a certain number of bits are added to each item of data. These additional bits represent a function of the numeric value of the data bits. For error-detecting systems, the device which reads the data repeats the checking calculation, and compares the result to the stored check bits. For some of the currently used algorithms, this approach will fail to detect at worst about one in 2**N of all possible combinations of errors, where N is the number of check bits. Some methods also guarantee to detect *all* errors of up to a certain number of bits. Reliability of detection increases with the number of check bits but the overhead associated with handling the additional information also increases. A tradeoff between reliability and overhead must therefore be chosen. With larger blocks of data, this tradeoff is less severe, but memory systems generally restrict the size of the protected blocks to that of the largest blocks transferred to and from memory in a single transfer, both to simplify writing and to maintain a reasonable memory bandwidth. In most current fault-tolerant systems such transfers are of 64 bits or less. If errors are detected but not corrected, it is necessary to maintain multiple copies of each value stored, preferably in physically separate memory devices to reduce the risk that all copies will be damaged by a single fault. The alternate locations should also be protected by check bits.

Error correcting codes contain sufficient redundant information in the added bits to allow reconstruction of partially damaged data. The usual claim is that all errors of up to a certain number of bits can be corrected. As an additional service, the error handling circuitry may be able to detect and report a much larger set of errors than can be corrected. The overhead for a given degree of error correction is greater than for a corresponding degree of error detection, but one may be able to reduce the level of backup protection and still maintain the same level of reliability. Error correction schemes usually work best when bit errors are evenly distributed. Since this is not always the case, and since too many errors in one data block can prevent recovery, at least one redundant backup copy is still important for highly reliable systems.

Architectures affected: Most systems have large storage areas available to the processors, to which these considerations apply. Systems which use a single large shared memory are particularly susceptible to memory faults, since such faults can affect all the processors by preventing successful use of their memory space, or by garbling processor-to-processor communications handled through common memory.

Test method: Error detection systems, upon perceiving an error, will generally send out a hardware or software error notice, which may be readily available to outside devices, or which may be reachable by probes into the system. Error correcting systems sometimes send out error notices, but usually refrain from signaling when a correctable error has been detected. In fact, the error recovery mechanism may be implemented at a very low level in the hardware, and the higher levels of hardware may not be aware that a correctable error has occurred. Unreported error correction, in addition to making measurement more difficult, can leave a user uninformed in the event that the system has a large number of hidden permanent faults which effectively reduce system redundancy. Some sort of reporting is therefore desirable even for errors which are corrected.

Unless the system under test has specific hardware features that make it possible, inducing errors of only a few bits in a few memory locations may prove to be difficult. Blocking out entire memory locations can be accomplished by tampering with the address signals. Similarly, bit errors that are the same for all locations can be induced by driving the data lines. If the system features modular construction with removable, standard memory devices, it may be useful to temporarily replace one or more of the devices with units known to be defective, in order to observe the detection and recovery process.

Special apparatus: Probes into system, error signal drivers. Defective memory modules could be useful, as described above. If this test is sufficiently important, special hardware could be built that would recognize specific memory addresses and inject errors when these addresses are accessed. In a system with a simple addressing protocol, there is a good chance that such a device could be implemented using a single programmable array logic (PAL) device.

Limitations of test: It may be difficult to induce or observe the response to certain patterns of errors that are of importance in evaluating fault tolerance. Though memory is more likely to be accessible to the outside world than many other system components, some systems have memory that is not accessible because it is on-chip or because of space limitations.

3.3.4 Errors in local cache.

Description of fault: In order to reduce average memory latency and the loading on shared resources, systems with a large global memory often have a local cache memory associated with each processor. Copies of values from main memory are temporarily stored in the cache for fast access, in accordance with the particular cache control algorithm used. When the appropriate type of access is attempted, the memory controller looks in the cache before starting an access of main memory. Because of the close relation between cache and the memory access controller, problems with the cache can seriously affect all memory accesses. A cache that always reports a "miss" will slow the processor considerably, and the controller will waste time trying to update the cache. A cache that falsely reports a "hit" or stores a value incorrectly will cause an incorrect value to be received by the processor. While caches often use parity checksums on the data contents, there may be a tendency to leave cache address handling without proper protection. It is therefore important that cache controllers in fault-tolerant systems be designed with the ability to detect faults in the address and control functions of caches. This protection is made more difficult when transformation of addresses is employed.

The *cache controller* must keep track of the main memory addresses of all the values stored in cache, and handle updates to or from main memory. In a typical set associative caching scheme, part of the address of a value is correlated with its location in the cache. The remainder of the address bits are stored in a table in a dedicated fast random access memory. This memory has one entry for each block of locations in the cache. Along with each entry in this table may be a bit specifying whether or not the corresponding location holds a currently valid item from memory. The contents of this table can be protected by check bits. The cache control mechanism can also be made fault-tolerant. For most caching schemes, detection of cache faults is far more important than local recovery of cache data, since in the event of a fault the "entry valid" bit may be cleared, a cache miss indicated, the cache disabled, and a main memory access initiated, with loss only of performance, not complete failure.

The most obvious approach to fault detection in the cache controller is redundancy. It may be practical to place several copies with comparison circuitry in each controller. Diagnostics could help in the identification of permanent failures. Fault notification would be very useful in the recovery of the rest of the system.

Architectures affected: The systems affected by this parameter are those which use cache memory.

Test method: User diagnostic programs can be used to detect permanent failures, but transient faults are likely to be interpreted as memory faults. If the signal lines are available for probes, faults can be recorded directly, or additional devices can be connected in parallel with the cache controller and the outputs compared. (A large number of systems use the TMS 2150 set of integrated circuits to implement cache control.) It may be possible to induce faults by driving data and address lines, or signal lines such as the "entry valid" line. Some systems allow the cache to be disabled, partly to facilitate the testing of other parts of the memory system. Fault notification lines may often be found for simple cache faults such as parity errors. Notifications for more complex cache faults, rare in current systems, would be specific to the particular system being evaluated.

Special apparatus: Probes into system, error signal drivers, possibly duplicate hardware.

Limitations of test: Because certain failure modes merely slow down operation of the processor, rather than causing overt errors, complete characterization of faults and fault detection may not be practical.

3.4 Faults in data transformation elements

Data transformation brings about permanent changes in the data, with the possible incidence of consistent (permanent failures) or inconsistent (transient failures) errors in the output. Some form of redundant execution using different individual devices is necessary for reliable detection of errors.

3.4.1 Detection of faults within operating processors.

Description of fault: This parameter refers to the central portion of a system processing element, which is directly responsible for managing data stored in registers, manipulating and sending out data, fetching data and instructions, and controlling the sequence of execution of the program. In many modern systems, these functions are implemented on a single integrated circuit, a notable exception being bit slice processors, which are implemented using a small set of specialized integrated circuits. Memory management and mathematical operations, if performed outside the central processor, are considered separately.

Most current systems use extremely complex processors, with built-in features such as pipelines and internal cache to enhance performance. A complex microprocessor can have up to several hundred thousand logic elements, with more expected in future processors. At the same time, the number of signals available for interface with outside circuitry is extremely limited, almost always less than 200. (Simpler processors generally have far fewer pins or numerous processors on one integrated circuit, so the problem is essentially the same.) With the tremendous premium on signal lines, mechanisms allowing the user to monitor the detailed inner operations of the processor are usually neglected. (Bit slice processors and processors built from small- and medium-scale components, having more of their "internal" signals available to outside circuitry, can usually be retrofitted with additional circuitry more easily than can other types of integrated circuit processors.)

Because of the relative isolation from surrounding circuitry, any checking for processor errors in any system without redundant processors is generally the responsibility of the processors themselves. Some processors make use of on-chip fault detection and recovery mechanisms, but the extreme complexity of a processor and its even more complex state space make it impractical to provide really satisfactory coverage for all possible faults. Faults which are detected and corrected generally do not result in any notification outside of the processor, while faults that are detected but not corrected usually cause the processor to send out a specific error signal (which indicates error type, but not necessarily exact location), and halt normal operation so software fault recovery efforts can begin. Without external control mechanisms, a fault that causes a processor to lock up or run wild is likely to cause a system failure.

A method frequently used to check the internal operations of a processor is to execute a job on two or more processors and compare the results. If the results do not agree, then fault recovery operations are started. This method is effective for fault detection because, barring design flaws, two processors are unlikely to develop exactly the same set of transient or permanent hardware failures. A complex result on which two or more processors agree is therefore much more likely to be correct than a result produced by a single processor. The risk of common errors due to errors in input is minimized if memory, buses, etc. are also redundant. The comparison mechanism must also be highly reliable, or comparisons will not be valid.

The techniques used for fault recovery in redundant-processor systems depend on the interval between comparisons. Lockstep machines, which compare results in hardware every machine cycle, can suppress each fault as it appears, as previously described. Systems which check results only at the end of the task must repeat the entire task whenever an error appears. Some systems use a timer or program instructions and software comparison algorithms to compare results periodically throughout execution of the task. This is called barrier checkpointing, because a synchronization mechanism must be used to make sure that the processors have all reached the same point in the execution of the task, by blocking execution until the checking has been completed. A copy of the intermediate results and the processor state is stored in a safe place. In the event that an error is detected in a system with only two-way redundancy, execution is started on backup hardware at the most recently completed checkpoint. Comparisons of intermediate results may be made more often than checkpoint storage, with execution after fault detection always returning to the nearest true checkpoint. Checkpointing increases the complexity of the software, but is able to protect running tasks, usually with less hardware than a system having enough redundant hardware to suppress errors. Checkpoints must be used before operations where recovery from an error may be impractical, such as a write to global memory or disk, or an I/O access.

Without checkpointing, a system with only two processors running a given task faces great difficulties when a fault is detected, since it is difficult to tell which processor is at fault. As an alternative to checkpointing, some fault-tolerant systems use more than two processors to a task, with voting on the correct result. Such systems, if operating in lockstep mode, can be designed to recover from faults with no additional delay, which is useful for some time-critical applications. To a certain extent, an increase in the level of redundancy increases the fault tolerance of the task being executed. In theory, one can allocate different numbers of processors to tasks for different levels of security.

Use of three or more processors for recovery is sometimes combined with checkpointing to create two levels of recovery mechanisms. A variant of this approach is the use of a "warm backup". This is a set of one or more processors not specifically allocated to the task at hand. Error detection is performed using two processors with checkpointing. When an error is detected, the backup is used along with the original processors from the last checkpoint to determine which processor is in error, and what the correct result should be. The backup may be assigned to replace the erroneous processor until diagnostics can demonstrate that it is working correctly. Possible benefits of this approach are continued reliability in the event of the permanent failure of a processor, and less hardware overhead required for a given level of reliability, since several active processor pairs may share a common pool of backup processors.

As previously stated, the effectiveness of redundant processor techniques depends on the accuracy of the comparisons, and the correct handling of error signals. Checkpointing approaches also depend on the accuracy and reliability of storage, and completeness of intermediate results. While lockstep systems can generally get by with comparing the outputs of the processors, checkpoint systems should record the point of execution in the program, and the values of all registers, flags, and storage local to the processor that are relevant to the execution of the program. Storing all these variables in memory can take a considerable number of machine cycles, which can considerably slow down execution when there are numerous access to external devices. Custom processors with additional ports for comparison and checkpoint storage could help to reduce this overhead. Systems which use more than two processors on a task to implement fault recovery can lose recovery capability or even operability as a result of permanent failures in the processors. When there are as few as three processors, the loss of one processor will allow the system to detect faults, but not necessarily to recover from them. When there are two processors, the loss of one processor makes comparison fault detection impossible, though successful completion of diagnostic programs may indicate which one still functions correctly. The ability to reconfigure spare processors to replace damaged ones can help to alleviate the problem of processor loss. In theory, one could design a system which uses comparisons among processors to recover from faults, until the number of processors drops to two, at which point software checkpointing is used to maintain recovery capability.

Architectures affected: All digital computer systems use processors in some form. Commercial processors may have a certain amount of built-in ability to detect faults, but recovery is usually the responsibility of the user/programmer. Commercial multiprocessor systems employing lock-step fault detection and correction are available.

Test method: Processors that have many of the internal signals externally available, such as bit slice processors, may be fitted with probes to directly observe and induce faults. All other types of processors must be observed from the outside by their performance, using whatever signals are available. The options that may be available to the measurement equipment to detect faults are essentially the same as those of the fault-tolerant systems themselves: diagnostics, and temporal and hardware redundancy. The measurement equipment may carry out these fault detection procedures much more extensively than does the system under test in its own processor fault detection. This allows more effective detection of the induced faults, as a reference for comparison to the fault detection of the tested processors. The system comparison circuitry may be available for the attachment of probes. Faults may be induced by tampering with signals coming into or going out of the system comparison circuitry, which should cause the system to react as though one of the processors is at fault. The processors themselves can be made to operate incorrectly by such methods as modifying their instruction or data inputs, placing them in a HOLD state, and changing the power supply. It may be possible to substitute devices known to be defective for the corresponding ones in the system.

Special apparatus: Probes into the system and injected error drivers are desirable when practical. Depending on the degree of testing to be conducted, special equipment may range all the way up to external emulators for the processors and comparison devices. Microprocessor manufacturers can make test emulators available for their processors. Such emulators could be implemented using the target processors themselves, with many internal signal lines which can not be accessed in the commercial versions made available to the outside world.

Limitations of test: The chief limitation of these tests is the inability to directly access the inner workings of the processors. All of the methods described are ways of working around this problem. The extremely complex state space of a typical processor increases the difficulty. As for other measurements, what can be done depends on the particular system in question.

3.4.2 Detection of faults within coprocessors and controllers.

Description of fault: Most computer systems use processors which are designed for general-purpose use. In a given application, one often encounters functions which are to be performed many times, and which the processor could perform using lengthy routines. These operations were not built into the hardware because implementation of the functions would have exceeded available space or caused loss of generality. An example is floating point mathematics, which is used in many applications, but which requires a large, complex circuit for hardware implementation. If these functions can be completed using specialized hardware, or even run in parallel with execution of other parts of the program, overall speed of execution may be improved considerably. A common method of implementing these features is to use specialized processing devices called coprocessors and controllers, coupled to the general-purpose processors. They are generally subservient to the general processors, though they may generate interrupt signals to pass messages. Communications with the general processors may go through special buses, or through the normal processor and address/data lines. Commands from the general processors may be purely memory-mapped, or they may pass through additional control lines. Coprocessors and controllers are similar in the ways that they assist the general processors. The distinction between the two is that coprocessors, once initially instructed by the general processor, take over operations to execute instructions as they appear in the instruction stream, while controllers are repeatedly supervised by simple routines executed in the general processor. External devices for mathematical computation are commonly implemented as coprocessors, while controllers include processors to handle disk memory access and input/output for communications. Discrete memory managers, since they perform many of the same functions, probably should also be considered in this category.

Because of the importance of mathematics in a wide range of applications, many systems use mathematics coprocessors. These often contain more logic circuitry than the processors they assist, and are separate from the processors largely due to size limitations.

Receiver/transmitters for communications and disk controllers are included in many systems. Operation in parallel with operation of the general processor is a major incentive for the use of I/O controllers. These devices allow the system to handle slow communications without the need for continual supervision and consequent slowdown of the general processor.

External memory managers are used when the features of memory page protection and hardware translation of virtual addresses to physical addresses are desired, but the designer has chosen to use the circuit space in the general processor for other features. A memory manager is controlled by instructions from the general processor, and acts as an interface between the general processor and the memory address space.

Coprocessor/controller faults that can interfere with proper operation of the system include addressing errors, handshaking errors, failure to provide an output, errors in output, and unauthorized output. Because the devices are closely associated with the general processors, these faults can be particularly drastic and hard to detect. Addressing errors are discussed in another section. Handshaking provides an indication of whether or not interchanges are taking place in a proper manner. Failures thus indicated or failures in the handshaking mechanism itself should be detected by the processor and result in error messages. A failure in an output may cause failures in the general processor or external memory devices, or lead directly to incorrect outputs to the "outside world". Without having another source for reference, it is unlikely that the general processor will be able to detect such faults. For fault-tolerant applications, the most workable approach is to use redundant coprocessors and controllers, with checking of results before anything is sent outside of the protected area. If one tries to assign several coprocessors to each general processor, however, the timing problems caused by the delay for comparison could cause processor errors. In many systems, furthermore, the processors themselves need to be redundant to allow error checking, causing either a need for a large number of coprocessors, or the problems associated with combining the outputs of several parallel devices into one signal by voting, then running it to several other parallel devices. A more reasonable approach for checking coprocessors in such redundant-processor systems is to regard each processor/coprocessor set as a single processing unit, then combine several of these units using the redundancy techniques described for fault detection in processors. For controllers, which are likely to be shared among processors in a multiprocessor system, putting several together for error detection is probably the best method.

Architectures affected: A large percentage of systems use coprocessors or controllers, and are therefore susceptible to faults in these devices.

Test method: Since coprocessors and controllers are separate from the general processors, there are connections between them that may be accessible to probes from the measurement equipment. This may make it possible for the measurement equipment to pick up signals and directly observe faults and indications of faults. If all the signals can be reached, the user can attach one or more coprocessors/controllers in parallel and compare the outputs to that of the unit being tested. Emulators may also be used, as for general processors. Comparison circuitry may be checked in the same manner as for general processors. For an indirect approach, devices with easily predictable outputs, such as math coprocessors, can be tested using diagnostic programs. Diagnostics can sometimes be used with controllers, by keeping a log of the results of their outputs, and comparing to other systems. Errors may be injected by driving the data and control lines, removing devices, or substituting defective devices. Once errors are determined to be present using these techniques, the effectiveness with which the system detects faults can be evaluated.

Special apparatus: Probes into system, additional coprocessors/emulators, error signal drivers.

Limitations of test: If the signals and components are not readily accessible, direct observation of faults and induction of faults will probably be impractical. Because of their unique interactions with processor and memory, memory managers may prove very difficult to evaluate, other than by comparison of duplicates.

3.4.3 Detection of faults at the processor-board level.

Description of fault: There are several important ways in which evaluation of faults at the processor-board level differs from evaluation of faults within an enclosed processor, such as a microprocessor. These differences pertain to the design features that may be implemented, the number of signals available, and the complexity of the system.

Since the circuitry to support the operations of the processor is chosen during the design process, the designer can put in whatever hardware is considered desirable to aid in detection of faults and fault recovery. Even when commercial processors are used, with a careful design one may include a high level of fault checking and explicit notification of faults. Depending on the design goals of the particular fault-tolerant system, fault detection may involve the cooperation of a set of boards. For instance, if a triply redundant system, designed to keep running at all times with a high degree of reliability, is built with all three processors on a single board, then a failure in power supply, board circuitry, etc. that affects one processor is likely to affect the others as well. Furthermore, the user cannot remove one processor for replacement without disabling the others. In such a system, therefore, one might find the multiply redundant components spread out over several boards, with local links between the boards and distributed circuitry to check for inconsistencies. On the other hand, the triply redundant processor may be placed on a single board and considered as a single highly-reliable system element.

In contrast to the internal signals of a microprocessor, a board-level design may have hundreds of signal lines to which probes may be attached, including important signals for control, timing, and fault notification. For a given type of physical system layout, availability of signals for observation of faults and fault notification and for injection of errors is likely to be much better at board level than at processor level. (If what is considered the "processor" takes up the entire board, then the board-level descriptions apply.)

A setback to fault detection at board level is the greater number of logic elements involved than at processor level, and the extreme complexity of operation that is often encountered, with multiple clock phases, state machines, and complex handshaking procedures. While commercial mass-produced processors are likely to have a large percentage of their operational states and error conditions well described, this is much less likely to be the case for the custom-designed processor board containing them. It is also more likely that there will be design errors, which can hamper both operation of the system and observation of fault detection.

When there are several processors per board, either coupled together or independent of one another, observation of the interactions between the processors can be important for detection of faults, though these interactions can be more difficult to deal with than the signals on single-processor boards.

Architectures affected: Though this approach is not universal, many current systems are built with one or a small number of processors per circuit board, with at least some possibility of probe access to the board-level signals.

Test method: As described above, the relatively open coupling between components on the board can often make access to the needed signals fairly straightforward. It is of course necessary for those making the measurements to have sufficient knowledge of the design of the boards to be able to identify and locate the signals that will allow the measurement equipment to directly observe both faults and fault notifications. (This usually implies the necessity for good communication with the designers of the system and often involves proprietary design descriptive material.) The test equipment can then be set to look for faults caused by failures in signal lines, logic devices, etc. If resources, line loading, and timing considerations permit, it may be useful to connect an identical board in parallel (all inputs and outputs on the board) to the board being observed, in order to check all the signals on that board. (Some boards may have components, such as autonomous polyphase clocks, that do not initialize to a single defined state. This method would then be unworkable without disabling the corresponding components on the comparison board.) Errors can be induced as for other tests, with the distinction that the more open architecture at this hardware level can make placement of the induced errors more precise than at the processor level.

Special apparatus: Probes into system, error signal drivers. As described above, it may be possible to make use of a duplicate board, with the necessary signals brought out to make a parallel connection.

Limitations of test: The chief limitation is the extreme complexity of operation of a typical processor board as a whole, and the degree to which those conducting the tests must understand it before certain measurements may be made. This is in contrast to the situation for commercial microprocessors, which are widely known and well documented. The physical layout of the system, including board placement, method of integrated circuit mounting, and conductor layout (such as multilayer boards) can interfere with the attachment of probes. It may not be practical to attach another board in parallel to enhance fault detection in the measurement equipment.

3.5 Faults in computers in loosely-coupled systems

These may include internal two-way redundancy for fault detection, or may merely rely on failure to produce output or "I'm alive" messages in a timely manner.

3.5.1 Detection of faults at the component-computer level.

Description of fault: When a number of distinct computers are connected together in a network in such a way as to form a single fault-tolerant system, it is evident that messages passed between the component computers will play a significant role in the operation of the system, and that equipment to measure the occurrence of faults should pay attention to these communications. If the component computers are highly fault-tolerant in themselves, then the only issues of interest in the message passing pertain to faults in the communications system itself. Evaluation of such faults is described in the section on data transmission. If, however, the system performs processor fault-checking by means of interprocessor communications, then observing these messages is the key to observing fault tolerance in the component computers and in the system as a whole.

In redundant-processor loosely-coupled systems, all information used for comparison, and many of the fault detection notices, pass through the communications links. For the overall system to be fault-tolerant, these links and the comparison mechanisms must also be fault-tolerant. A possible performance problem arises from the fact that the links generally can not transfer data as rapidly as can be done between tightly-coupled processors. A comprehensive comparison of the states of two processors can therefore be a lengthy task. In order to maintain good system performance, the designer will probably seek to make the interprocessor state comparisons as infrequent as possible while meeting the requirements for recoverability. Checkpointing, as described in the section on processor faults, is a likely approach. It may also be possible to reduce the length of time required for each interchange by somehow reducing the data to be sent. As an example, a simple algorithm implemented in hardware or software could take in the pertinent state information of the component computer, and produce a much shorter check or signature. These check bits would then be transmitted, and compared to comparable check bits for another component computer. If a discrepancy were detected, the machines could compare states in much more detail.

Architectures affected: The architectures affected are the ones which are made up of loosely-coupled processors which use message passing as part of the fault detection process.

Test method: The fault detection techniques used by these systems present a number of opportunities for the evaluation of both faults and fault detection. In loosely-coupled multiprocessors, it is highly likely that the signals sent between processors are available for interception by outside equipment. These signals, if properly interpreted, contain the complete information used by the system under test for fault detection and notification among processors. Fault recovery procedures are likely to be largely software-based, and may be observable if probes can be attached to the internal signal lines of the component computers. It may be possible to use a spare component computer in the system to parallel the efforts of the processor being observed, in order to aid the measurement equipment in its observations. Faults may be induced by forcing errors in the component computers, or by disrupting communications between computers, in either a random or a systematic manner. Random disruptions would produce unpredictable, spurious fault notifications. Systematic disruptions could be used to test specific features of the fault detection mechanism.

Special apparatus: Probes for communications links, error signal drivers, possibly probes into component computers.

Limitations of test: The comparison messages may be sent in a format that is not readily interpreted by the measurement equipment. The data may be reduced, as previously described, and require another processor for interpretation. It may be possible to access only one half of a "conversation", and both parts may be needed to properly interpret the transaction.

3.6 Faults in I/O systems

The detection of faults in I/O systems is very similar in principle to detection of faults in internal transmissions and data storage, though the actual implementations are different. Completion of data transfers is assured by handshaking mechanisms, and correctness may be verified by redundant transmissions, inclusion of check bits, etc. Consistency could be checked by comparison among state models of the various devices. In any long-term operations, faults in I/O operations would be likely to eventually cause faults in the internal operations of the systems, which could be detected by the means previously described.

Measurement equipment may have an advantage over the system under test in the detection of faults, since it may be able to directly observe the devices at both ends of the communication path, allowing it to detect both errors in transmission and errors that occur at each end. Evaluation of fault recovery also benefits from this situation. I/O pathways are usually the most accessible part of any system, so connection of the measurement equipment should not be too difficult. Similarly, injection of errors should be straightforward.

4. Fault Recovery Techniques

Central to the objectives of many fault recovery mechanisms is the containment of faults. A processor or other device that puts out erroneous results can cause inappropriate response in other devices, corrupt databases, and ultimately lead to system failures. This propagation of errors must therefore be kept under control before repair of the damage can begin. The optimum form of containment depends on the source and type of the original fault, and the areas to which it is likely to spread. Several techniques call for precautionary measures that must be taken ahead of time.

As is the case for fault detection, which can use hardware or software techniques, fault recovery mechanisms can be implemented in hardware, software, or a combination of the two. Systems which use hardware recovery techniques generally also use hardware detection, and include such measures as choosing the correct answer by voting among three or more devices. (Such systems are the only ones that may be able to recover from faults with no additional delay in operation, though the delay for comparing results is always present.) Software recovery techniques may be used with either hardware or software fault detection. Software recovery will generally take longer than a comparable level of hardware recovery, but software techniques are much more adaptable to the needs of different situations, and a considerably wider range of operations may be performed. Software recovery may be limited to investigation of machine state, and analysis and correction of data. Performance of the recovery mechanism may be substantially enhanced, however, if the system allows the software to control the hardware at a lower level than is traditionally allowed in major systems. This low-level control may include the ability to switch processors and other components on and off, to reconfigure the system, and to conduct tightly-controlled tests of system components. In order to allow for such control by the software, appropriate hardware devices must be built into the system. Such an approach might best be considered a mixture of hardware and software fault recovery techniques.

4.1 Views of Fault Recovery

Because the users of fault-tolerant systems have many different requirements, there are many definitions or specifications for satisfactory fault recovery. An "ideal" form of fault recovery would allow the system to completely repair the damage caused by the fault and continue normal operation, with no resulting delays. This type of recovery is possible for certain systems experiencing certain types of faults, but can not be assured in the general case. For all other situations, some sort of tradeoff must be established, in accordance with the needs of the user. The factors involved in the tradeoff include correctness of result, volume of output, time to recovery, survival of specific processes, integrity of data structures, performance degradation during recovery, near-term and permanent degradation in system performance and redundancy, and system cost and complexity.

A traditional view of fault recovery is that avoidance of errors in output has absolute priority over all other considerations, with successful completion of the proposed job as a second priority. A single incorrect output is considered a failure of the program or even of the system. Emphasis is placed on fault detection and reliable response to detected faults. When a fault is detected, the system may suspend or cut back on normal operations in order to divert resources for recovery from the fault. Measures taken may include a number of approaches to reconstruct the previous fault-free state, and test and reconfigure the hardware. Efforts are made to "contain" the damage caused by a fault to the smallest possible architectural and data storage regions in the system, by such methods as checkpointing and checking for faults before any major activity such as an update to a database. If damage occurs from which recovery is not possible, then a failure of program or system is considered to have taken place, and execution is halted.

Emphasis on continued execution of specific processes, with secondary priority on correctness of results, is another view of fault recovery. There are processes such as machine controllers, for which continued operation is vital, as long as the output remains correct to within specified limits. Operating with minor damage to data structures, etc. can sometimes be less damaging than halting operations. It therefore becomes the responsibility of the fault recovery mechanism to provide for the best possible operation, and to determine whether damage is of such a degree that operation must be halted. There may be a need to minimize recovery time by such measures as immediately switching suspected components out of use, leaving any analysis for permanent failures for later. Continued output with reduced throughput may be acceptable in some cases. In some applications, a damaged database is "self-repairing" over time, as damaged elements are replaced or overwritten by new values.

A third view of fault recovery addresses the long-term ability of the system to maintain at least a minimum level of performance and redundancy. When a processor or other component suspected of being faulty is switched out of normal operation, there is a reduction in performance, redundancy for detection and handling of faults, or both. Studies appear to indicate that for most systems, the great majority of hardware failures are based on transient conditions [SIEW86]. Though a component that has just produced a faulty output can not be considered a satisfactory reference for the detection of new faults, a later test may show that the suspected component is now working properly, and may be added to the pool of usable components. Systems which emphasize optimized performance over long periods of time should have the ability to recertify faulty components. In order to minimize execution time of the affected process, execution will often be continued immediately using substitute components, while the testing process is put off until a more convenient time.

4.2 Fault Recovery Methods

After an error has been detected, there must be a choice of what action to take. The general approach is to somehow use or create an error-free version of the data. Alternative actions may be to terminate operations until repairs are made (e.g., a home computer with a faulty memory) or even to continue operation and ignore the error (e.g., a bad pixel in a graphics display). The action taken is usually based on the operational environment as envisioned at design time.

If the errors are assumed to be transient (rather than permanent or "hard"), then retry is a reasonable strategy (e.g., reread disk memory, or retransmit data). However, this approach is time-consuming and may not work. If hard errors are expected, use of error correcting codes (ECC) or a backup copy are reasonable strategies. Both require additional data and components. If a backup copy is to be used, a number of different steps are required for correction, such as diagnosis of the error source, disconnection, and switching to a backup data source.

Errors in data transformation can not be corrected by the use of ECC, since a correct ECC passed through the transformation stages will no longer be valid. It is often assumed that data transformation errors are **not** transient. For this reason *retry* is not the usual solution. Multiply redundant voting systems and recalculation in another portion of the system are commonly used.

4.2.1 Recovery in loosely coupled systems. Software recovery procedures, invoked after a fault is detected, include one or more of the following steps:

Fault location: Determine the unit that caused the fault. This may be self-evident from the fault, as when a processor fails to respond within a time limit, or when the detection mechanism is hardware voting and the result of the vote indicates the processor that is faulty. At the very least, the error itself will provide some indication of the level, if not the general identity, of the faulty unit. Some combination of hardware and software diagnostics can then be used to locate the source of the fault. Transient faults are almost impossible to locate by this delayed testing procedure, since they may not remain stable long enough to permit analysis. The application will dictate the importance of identifying the source of the fault and thus the method used to locate it.

Fault isolation: Isolate the unit that caused the fault from the rest of the system, either by reconfiguration (hardware switching) or reorientation (logical switching). This may be accomplished by the failure itself, as when a failed processor no longer participates in the system and therefore does not appear to exist. If the failure does not cause self-isolation, then the remaining system must, through some combination of hardware and software, switch out faulty units and utilize backup units, either ordinary operating units or standby units, to handle the failed workload.

Fault diagnosis: Hardware tests and software diagnostics can be used to determine whether the error is hard or transient, and if it is hard, to locate the faulty unit and replace it with a standby. Standby units can be "hot" (powered up and available for immediate use), or "cold" (not powered up or not immediately usable). A cold standby can be used when the perceived failure mode of the unit is due to deterioration with time, as opposed to time-independent events. If the fault is transient, the unit is placed back into available service. Depending on the type of fault and the complexity of the system, this step may or may not be included.

Workload rollback: Distribute the job that was interrupted onto standby units or among the remaining non-faulty units, and re-execute some portion of that workload, starting from a previously known/saved state.

Workload rollback is usually handled by *checkpointing*, in which a periodic "snapshot" is taken of the current program location and the current value of its variables. The checkpoint information (either complete or incremental) is passed to a backup version of the process (either on common secondary storage or to a designated processor). When a failure is detected, the backup copy is invoked and initialized to the state of the last checkpoint information set. Checkpoints are taken both periodically and at critical junctures of the program, such as an action which is irrevocable from the viewpoint of the fault control system (e.g., a write to a database or I/O device). If a fault is detected, execution is backed up to the most recently checkpointed state for which all was well.

The backup unit which picks up the operation of the faulty unit incurs two time delays. The first delay is the time required to detect the error and invoke the recovery mechanism so that the backup can begin execution from the last checkpoint. The second delay is the time required to repeat the computations that were performed between the time the last checkpoint state was saved and the time the fault occurred [SERL84, ZORP85, MARK85, NEST85]. The backup unit must be able to "backout", or avoid repetition of critical operations (e.g., database updates and communicating messages) which have occurred during this repeated computation period. This backtracking may be complex and time-consuming.

If the remaining non-faulty units, rather than standby units, are used as backup devices, then failures will result in a net reduction in system computing power. This happens because the recovery process uses part of the computational capacity, and because the backup versions, which now execute on some of the remaining non-faulty units, also use part of the computational capacity. If this reduction in computing power cannot be tolerated, special standby units must be provided for backup.

Systems with continuous data updates, such as certain signal processors and controllers, may be able to continue operations with incomplete information, as new information is gradually added to the system state. Continuous service programs may require only some reasonable point in their cycles at which to begin. Programs which are sensitive to existing machine state and for which no adequate records are preserved will have to be discontinued, or restarted at the beginning.

Some side effects may result from use of these checkpointing techniques, since it is likely that the process has executed for a period between the time the last checkpoint state was saved and the time the fault occurred. Operational features which may be affected include:

Data base consistency: Any modifications made to a data base by the failed process since the last checkpoint must be undone or otherwise accounted for, so as not to create an error in the database when that portion of the code is re-executed.

Communication consistency: Primary communications (messages sent or received from other processes to or from the failed process since the last checkpoint) must be "properly handled and accounted for". This may be more difficult than it sounds since primary communications can create a "domino effect" [KUHL86] among cooperating processes, affecting secondary communications (messages sent between other processes, that were caused by or would cause a primary communication). Complications arise because the secondary communications could have taken place at any time, not just since the last checkpoint.

In certain architectures and applications, parallel processor systems complicate fault tolerance [RENN86] because:

(1) One of the purposes of parallel processing is to apply many processors to a problem in order to solve it more quickly than can be done using fewer processors or only one processor. An optimum realization of this goal requires that fault tolerance be achieved through standby spare processors to replace those that fail, rather than distributing the workload among the remaining non-faulty processors. Thus, the level of processor redundancy is significantly higher in these types of systems.

(2) The topology of the parallel processing systems may be important in the solution of the problem. Therefore, not only must a processor be replaced with a standby spare, but the spare must also be inserted (by some reconfiguration mechanism) into the same location in the structure with the same communications topology in the system. For tightly coupled systems (i.e., shared memory, usually via a common bus) the topology is not significant. In other parallel processor configurations such as trees, n-cubes, and grids, which may contain hundreds or thousands of processors, topology can be a significant part of the solution design.

4.2.2 Recovery in tightly-coupled systems. Tightly-coupled systems can use the same backup schemes as loosely-coupled systems. They can, however, employ other techniques which allow instantaneous recovery without the overhead of checkpointing and backtracking after faults. One implementation approach is to provide a duplicate backup system running in lock-step with the system that may become faulty. The output from a faulty system is then replaced by that from the duplicate system; operation continues immediately with little or no time delay [SERL84, ZORP85].

In another approach, three or more identical components are used to perform the same operation, and majority voting is used to derive a correct result when there are errors. More simultaneous errors can be tolerated if more replicated units participate in the voting.

Comparison of outputs and voting can be performed directly by hardware, or by the use of software. Using hardware, the result of each instruction can be compared at the board level or at the outputs of chip level components such as processors, arithmetic units, memory chips, etc. If the error is corrected by combinatorial circuits without significant delay, operation continues unperturbed. A drawback to the hardware approach is that it requires cycle-by-cycle synchronization among the hardware components being compared, so a slight delay may be added to the execution time of each instruction. The hardware employed also represents an added expense above the design cost of a standard system. These drawbacks are balanced by the capability for instantaneous recovery.

4.2.3 Isolating faulty devices. Removing a device within a fault-tolerant system from active use may be accomplished by any of several methods. The faulty device must no longer affect system operation, but it may be desirable to have the ability to switch the device back in later for testing or for normal use. The simplest approach is for the device in question to be sent a signal which will place it in a WAIT state, after which it will not participate in system activities until it receives a reactivation signal. The most obvious difficulty with this technique is that the faulty device, being faulty, may not properly shut down. One may therefore be inclined to place a switch between each device and the rest of the system, in order to have more positive control. The switches, however, may also fail to shut off, may shut off when they are not supposed to, and add to complexity and propagation delays within the system. In order to be worthwhile for fault recovery, any such switches used must be highly reliable in comparison to the devices to which they are connected. It may be desirable to apply the known physical properties of electronic switches in the design of redundant switch systems that will meet this requirement.

For many fault-tolerant systems, the switching out of a faulty device or the switching in of a recertified or replacement device makes it necessary for the system to be reconfigured, meaning that changes are made in the logical or physical interconnections among the devices. Processors or memories may be assigned new addresses, and interconnecting signal paths may be rerouted by control of switching centers. When a device is introduced, any of a number of protocols of various complexities may be used to initialize the device and inform it of its new name or address. The major system components may all keep records of the names and the configuration of all other devices, and these records must be updated. When a device is removed, the corresponding entry in these records must be deleted or replaced, and the other configuration information changed appropriately.

When the hardware has been deemed sufficiently operational, efforts may begin to recover the functionality of the program that was being executed when the fault was detected. In lockstep machines, the appropriate machine state is still present in the working processors, and if there is a sufficient number of processors, operation can continue immediately. In machines which employ checkpointing, the most recent fault-free state can be loaded into the appropriate processors, and operations can resume.

4.2.4 I/O recovery techniques. I/O recovery procedures differ from the corresponding internal communication and memory recovery procedures in that much of the recovery process must be carried out by transfers through the I/O channels, in a context of negotiation among the communicating devices. The source of the fault must be determined, and corrective action appropriate to the devices and operations involved initiated. For a simple data transfer, a reasonable response would be the deletion and retransmission of the faulty data. A correction signal to a mechanical controlling device would have to be chosen according to the exact nature of the actions involved, since a correct but late signal might be worse than no signal at all. A fault resulting in damage to the software controlling structures for I/O transactions might necessitate the complete removal and reinitialization of these structures.

4.3 Evaluation of Fault Recovery

There are a number of measurements which can be useful in the evaluation of the recovery mechanisms of a fault-tolerant system. The parameters measured fall generally into the categories of ability to recover from faults and performance degradation incurred during and after the recovery process. Interpretation of these parameters is closely tied to the needs of the user and the design goals of the system being tested. As is the case for investigation of fault detection capabilities, measurement is most simply done when induced errors or induced notifications of errors are used, because of time constraints and controllability of the experiment.

For most systems, it is not practical to attempt to produce a specific rating that will describe the ability of the system to recover from any possible combination of faults, given the size of the system state space and the many ways in which faults can appear. It may be possible, however, to characterize the ability to recover from the general types of faults that are most likely to appear. Knowledge of the type of fault recovery mechanism used can help considerably in this process. As an example, a system without fault recovery will be unable to recover from any of a large set of faults. A fault recovery system is then added to deal with faults that seem most likely to appear and cause failures. However, there are usually certain types of faults that can defeat the purpose of the recovery mechanism and cause system failure. To counteract these faults, the designer may implement additional measures to deal with the most common and the most troublesome of these faults. This process may be reiterated until the designer feels that the fault coverage is adequate to satisfy the needs of the users. Knowing the characteristics of the recovery mechanism, one can make a reasonable estimate of the types of faults that will be easily handled, and the types which should receive special attention in the measurement process to determine whether they are adequately covered. These faults can possibly be induced, and the results observed.

Faults which can be especially troublesome for fault recovery systems include those which occur during the recovery procedure, and those which directly affect the operation of the fault detection and recovery mechanisms. To have a chance of counteracting these faults, fault detection must continue during the recovery process. Detection and recovery devices must have some sort of redundancy, and the ability to bring themselves to recovery. These improved mechanisms are in turn subject to still more subtle faults. In addition to the limits imposed by the developer in choosing an adequate level of fault coverage, the nature of the system may be such that beyond a certain point, additional complexity actually decreases the reliability of the system.

In investigating faults from which recovery may not be possible, some form of data logging by the measurement equipment can be very useful, since the system being measured may crash and be unavailable to help in the diagnosis. Any sequence of faults which leads to system failure should be recorded if possible, and analyzed later to obtain an estimate of the probability that this or related faults will appear during normal operation. It would similarly be useful to record faults which bring about an unusually long recovery time.

There are several ways in which performance degradation associated with the fault recovery process may be evaluated. One way is to measure the output of the system being tested as a function of time, while the detection and recovery process is underway. Depending on the number of processors, etc., this parameter may drop slightly, or drop all the way to zero during the recovery process. Another method is to measure the intervals between outputs, and to take the difference between an expected interval time and the observed time to be the recovery service time. A more difficult but theoretically more accurate approach is to keep records of the resources used and the intervals in which they are used. This measure prevents idle time on the system from being considered part of the recovery time.

Since recovery time can be affected by the types of faults detected, the choice of faults induced during a test period can significantly affect the observed level of performance of the recovery system. It is therefore important in making such measurements to make sure that the set of faults induced is of a type that will lead to useful results, whether the parameter of interest is "worst-case" recovery time or "typical" recovery time. Section 3 includes a more complete discussion on the methods of inducing faults.

5. Measurement of Fault Recovery

Upon detection of a fault condition, the fault-tolerant system will undertake a series of actions designed to limit the resultant effects, and restore the system to partial or full operation. The nature of the expected field of application determines whether the recovery scheme places emphasis on full recovery, or quick but possibly partial recovery. In the extreme case, where no interruption can be tolerated, instantaneous fault suppression hardware recovery techniques must be employed.

The recovery parameters of interest include useful system throughput for each process, response time, availability of each data set, ability of the system to handle additional faults, and loss of input data. Some users, particularly system designers, may need to know details of the utilization of system resources as well. Each of these parameters should be expressed as a function of time during the recovery period.

One should recognize that the measurement result may be that the supposedly fault-tolerant system *never* fully recovers from some particular faults. Some functions may be completely lost until repairs are made. Some data may vanish. The importance of these losses depends on the application. In systems continuously processing a stream of data, the loss may be of minor importance since old data could become uselessly stale by the time it could be recovered. These applications would not attempt to recover the lost input data.

The measurement of throughput and use of system resources can be accomplished using techniques such as those mentioned in reports by Roberts [ROB86] and Mink, et al. [MIN86]. A necessary adjunct to this measurement is the software running on the system under test.

The measurements proposed here are mostly directed toward the *reduction* in system performance caused by a fault. They do not, in general, concern themselves (except for 5.1.1) with the reduction in performance one must suffer in order to be prepared for a fault, should one occur. In the case of software recovery systems, the execution cost of checkpointing can be measured. In systems where hardware redundancy is used to suppress the effects of faults, it is generally impossible to measure the performance which could have been attained if this extra hardware had been used to add the capacity to handle additional instruction or data streams.

It will be particularly difficult to measure the availability of each data set *during* recovery. Special test code or careful synchronization of the time of fault injection or simulation with the progress of test routine execution will be necessary in this evaluation.

In most situations detailed measurement of resource utilization should only be attempted under steady-state operating conditions: before a fault or after recovery. Measurements *during* recovery will have to be limited to more gross parameters such as throughput or response-time or results-per-unit-time, unless the application justifies substantial expenditures for testing.

The results obtained when testing recovery performance may be greatly influenced by the type and location of the fault(s), and the functions that the fault-tolerant machine is performing (i.e. the software which is currently running). The results will thus be multidimensional.

Because the concerns and techniques of measurement of fault recovery are often similar to those of measurement of fault detection, this section repeats a certain amount of the material from section three.

As in detection of faults, recovery can be classified as being based on either **hardware** or **software** techniques. The common combinations are hardware detection, hardware recovery; software detection, software recovery; and sometimes hardware detection, software recovery. The fourth possibility - software detection, hardware recovery - is essentially unknown in practice.

5.1 Systems Using Software Recovery

Whether the faults are detected by hardware or software techniques, software recovery begins when the fault is detected. Thus elapsed times should be measured from the internal notification of fault detection (if possible), not from when the fault is injected into the system under test. Unavailability time is the sum of fault detection time and fault recovery time.

The interaction between the system hardware, system software, and the use to which the user's code puts them, is vital to the success and performance of the software fault recovery approach. The system will have to be tested with software which uses its fault-tolerant features suitably. The test software may have to be specifically designed for fault tolerance. In general this will be difficult and time-consuming. Creation of special test software is necessary to evaluate the system's fault-tolerant features divorced from user application code, in accordance with the general concept of synthetic benchmarking. On the other hand, this approach may be undesirable *because* it will not make the same use of the fault-tolerant features as any particular eventual application code.

5.1.1 Overhead in normal operation. There is a reduction in performance of fault-free fault-tolerant systems (with software recovery schemes) caused by checkpointing. At points chosen by the applications programmer, or automatically chosen by the compiler or operating system, critical information about each process must be transferred to each backup copy of the process and/or to "safe" places in the memory hierarchy. This overhead is clearly a function of the process and machine state which must be transferred. In any given implementation there will likely be a fixed portion which is characteristic of the machine and its software system, and a variable portion which is a function of the size of the application data set which must also be transferred.

Test method: "Interesting" sections of application code and special test code should be instrumented to allow timing of the code both with and without checkpointing code. Resource utilization can also be observed. The effect on both the "sending" process(or) and the "receiving" process(or) must be measured. The overhead of each checkpoint should be resolved into its basic and data-dependent parts. The special test code should provide for a range of data set sizes to allow measurement of this dependency. Particular attention needs to be placed on measurement of possible bottlenecks in interprocessor communication paths. No faults are induced or simulated for this test.

Apparatus Required: Measurement apparatus must be provided to quantify the throughput of the system, by process. At a minimum, this would use the system's timing service to measure execution times. Because of the serious overhead of this service in most systems, it is recommended that use be made of a time-stamping execution trace monitor such as that described by Mink, et al [MIN86]. For more detailed information about resource utilization bottlenecks caused by the checkpointing process, one could use a resource monitor such as that under construction at NBS [CAR86].

Limitations of test: This measurement may be impossible if checkpointing is an automatic function of the compiler or operating system. In these cases the most one can hope for is that an option of operation without checkpointing is provided. Then one can measure overall operation with and without checkpointing.

5.1.2 Faults in intermodule data transmission.
Description of fault: Faults may occur in the communication paths between processors. These faults are described in more detail in Section 3.2.1 of this report. The result of the fault is that the system must take some recovery action to correct the erroneous data or obtain correct data by using an alternate communication path. If the error is transient, retry will suffice. The system must either ignore the erroneous data, or have some recovery scheme to undo any effects of its use.

Test method: The best technique is to actually inject faults in the paths by the techniques described in Section 3.2.1. A lesser test method is to simulate the fault-detected signal, but this prevents measurement of the success in eliminating the errors caused by the fault. After the fault is induced or its detection simulated, throughput of the system is measured as a function of time for each process. Carefully contrived test routines can assist in revealing loss of data.

Special apparatus: The special apparatus mentioned in 3.2.1 is needed to inject errors, or to simulate a hardware signal indicating detection of a fault. This apparatus is not needed if the fault detection is simulated in software. Throughput and resource utilization are measured as in Section 5.1.1, above.

Limitations of test: Measurement of data loss, especially *during* recovery, is difficult. Measurement of the degree of success in preventing use or recovering from use of erroneous data is difficult.

5.1.3 Faults in addressing. This type of fault requires essentially the same testing approach as the above fault type. The most significant difference is that the likelihood of data or program damage in widely different areas of storage is much greater, increasing the need to exhaustively test for these effects through choice of test routines.

5.1.4 Faults in processor registers.
Description of fault: Processor registers are small, dedicated-purpose memory domains internal to a processor. They may develop either permanent or transient faults, as described in Sections 3.3.1 and 3.3.2.

Test method: Faults can be injected into the registers by the techniques used in Sections 3.3.1 and 3.3.2. A lesser test method is to simulate the fault-detected signal, but this prevents measurement of the success in eliminating the errors caused by the fault. After the fault is induced or its detection simulated, throughput of the system is measured as a function of time for each process. Again, success in discovering data loss is tied to careful design of test routines.

Special apparatus: The special apparatus mentioned in 3.3.1 and 3.3.2 is needed to inject errors, or to simulate a hardware signal indicating detection of a fault. This apparatus is not needed if the fault detection is simulated in software. Throughput and resource utilization are measured as in Section 5.1.1, above.

Limitations of test: Measurement of data loss, especially *during* recovery, is difficult. Measurement of the degree of success in preventing use or recovering from use of erroneous data is difficult.

5.1.5 Faults in main memory.

Description of fault: As described more fully in Section 3.3.3, the major memory assemblies of a computer system are prone to a number of types of errors. Recovery from these errors will at least temporarily reduce system performance. Some of the memory may become temporarily or permanently unavailable. This may result in the loss of some data (to some processes), or a reduction in system performance.

Test method: Faults can be injected into the memory system by the techniques used in Sections 3.3.3. A lesser test method is to simulate the fault-detected signal, but this prevents measurement of the success in eliminating the errors caused by the fault. After the fault is induced or its detection simulated, throughput of the system is measured as a function of time for each process. It is especially important to measure data loss or corruption. Using special test routines, evaluate any changes in the availability of memory to processes.

Special apparatus: The special apparatus mentioned in 3.3.3 is needed to inject errors, or to simulate a hardware signal indicating detection of a fault. Only part of the apparatus is needed if the fault detection is simulated in software. Throughput and resource utilization are measured as in Section 5.1.1, above.

Limitations of test: Measurement of data loss or corruption, especially *during* recovery, is very difficult.

5.1.6 Faults in cache memory.

Description of fault: Temporary copies of values from main memory are stored in the cache for fast access. The memory controller looks in the cache before starting an access of main memory. Because of the close relation between cache and the memory access controller, problems with the cache can seriously affect all memory accesses. A cache controller that always reports a "miss" will slow the processor considerably, and the controller will waste time trying to update the cache. A cache controller that falsely reports a "hit" or stores a value incorrectly will cause an incorrect value to be received by the processor. Caches are subject to the data errors common to any type of memory.

Test method: Faults can be injected into the cache memory system by the techniques used in Section 3.3.4. A lesser test method is to simulate the fault-detected signal, but this prevents measurement of the success in eliminating the errors caused by the fault. After the fault is induced or its detection simulated, throughput of the system is measured as a function of time for each process. It is important to attempt to measure the degree of data loss or corruption caused by the cache fault. The failure of cache memory will cause additional traffic on the path to main memory; utilization of this resource should certainly be measured.

Special apparatus: The special apparatus mentioned in 3.3.4 is needed to inject errors, or to simulate a hardware signal indicating detection of a fault. This apparatus is not needed if the fault detection is simulated in software. Throughput and resource utilization are measured as in Section 5.1.1, above. Special test routines must be used to evaluate data loss and corruption.

Limitations of test: Measurement of data loss or corruption, especially *during* recovery, is difficult. It is hard to predict just *what* data will be affected.

5.1.7 Faults within processors.

Description of fault: The central processors of a system are where most data *transformations* take place. Most current processors are extremely complex, and are often implemented as VLSI microprocessors with up to several hundred thousand logic elements. The number of signals available for interface with outside circuitry is almost always less than 200. (Simpler processors generally have far fewer pins or numerous processors on one integrated circuit.) Means to allow monitoring of the inner operations of the processor are usually neglected. Processors built from bit slices or small- and medium-scale components have more of their "internal" signals available for probing.

Test method: Faults can be injected into the processors or at their terminals by the techniques used in Section 3.4.1. A lesser test method is to simulate the fault-detected signal, but this prevents measurement of the success in eliminating the errors caused by the fault. Since this test involves data transformation elements, where detection of errors is difficult, the test method should pay special attention to the correctness of the processor outputs. The measurement equipment should carry out these fault detection procedures much more extensively than the system under test uses in its own processor fault detection. This allows more effective detection of the induced faults, as a reference for comparison to the fault detection of the tested processors. The measurement system may have to contain an error-free replication of the processor being tested for use in the comparison.

After the fault is induced or its detection simulated, throughput of the system is measured as a function of time for each process. It is important to attempt to measure the degree of data loss or corruption caused by the processor fault.

Special apparatus: The special apparatus mentioned in 3.4.1 is needed to inject errors, or to simulate a hardware signal indicating detection of a fault. The equipment mentioned above is needed to closely follow the correctness of processor output. This apparatus is not needed if the fault detection is simulated in software, though the test is inferior. Throughput and resource utilization are measured as in Section 5.1.1, above. Special test routines must be used to evaluate data loss and corruption.

Limitations of test: Measurement of data loss or corruption, especially *during* recovery, is difficult.

5.1.8 Faults within coprocessors and controllers.

Description of fault: Coprocessors and controllers are subservient to the general processors. Communications with the general processors may go through special buses, or through the normal processor and address data lines.

Receiver/transmitters for communications and disk controllers are included in many systems. Operation in parallel with operation of the general processor is a major incentive for the use of I/O controllers. These devices allow the system to handle slow communications without the need for continual supervision and consequent slowdown of the general processor. These processors may perform tasks which are logically part of the central processor or may perform some other task such as input or output processing.

Test method: The test methods of Section 3.4.2 are appropriate. There are generally connections from coprocessors and controllers and the general processors with which they are associated which may be accessible to probes from the measurement equipment. This may make it possible for the measurement equipment to pick up signals and directly observe faults and indications of faults. Emulators and simulators may be used, as for general processors. In addition to measurement of throughput as a function of time, special attention must be paid to discovery of missing or corrupted data.

Special apparatus: Probes into system, additional coprocessors/emulators, error signal drivers.

5.1.9 Faults at the (processor) board level.

Description of fault: Evaluation of fault recovery at the processor-board level differs from evaluation of faults within a microprocessor. The designer is more likely to have included hardware which is desirable to aid in detection of faults and fault recovery. Fault detection may involve the coordination of a set of boards. For instance, a triply redundant system, may be designed with the processors on separate boards to allow replacement without system shut-down. On the other hand, the triply redundant processor may be placed on a single board and considered as a single highly-reliable system element.

A board-level design will have hundreds of signal lines to which probes may be attached. Availability of points for observation of faults and fault notification and for injection of errors are likely to be much better at board level than at processor level. (If what is considered the "processor" takes up the entire board, then the board-level descriptions apply.)

While commercial mass-produced (micro)processors are likely to be very well and accurately analyzed and documented, this is much less likely to be the case for a board-level processor. There are also more likely to be errors in design, which can result in unexpected responses to faults.

If there are several processors per board, either coupled together or independent of one another, measurement of the interactions between the processors can be important in searching for side effects.

Test method: The test methods of Section 3.4.3 are appropriate. Measurement of throughput, data loss and data corruption versus time are important.

Special apparatus: Probes into system, error signal drivers. As described in Section 3.3.3. All of the normal throughput and error detection apparatus will be needed.

Limitations of test: The chief limitation is the extreme complexity of operation of a typical processor board as a whole, and the degree to which those conducting the tests must come to understand it before certain measurements may be made.

5.1.10 Computer faults in loosely-coupled systems.
Description of fault: In a loosely-coupled system each processor may be multiply redundant and thus be a highly reliable component, may include internal two-way redundancy for fault detection, or may be nonredundant and the system may merely rely on failure to produce output or "I'm alive" messages in a timely manner as an indication of failure.

In these systems the messages that pass between the component computers are very important in the operation of the system, and equipment to measure the recovery from faults must pay attention to these communications. Evaluation of communication faults is described in the section on data transmission.

Reduction of performance arises from the fact that the links generally can not transfer data as rapidly as can be done between tightly-coupled processors. In order to maintain good system performance, the designer will probably seek to make the interprocessor state comparisons and checkpointing as infrequent as possible while meeting the requirements for recoverability. Loose coupling of processors increases the likelihood that some of the function or information in the system will be completely lost due to a fault.

Test method: The test methods of Section 3.5.1 form the basis of this measurement. Fault recovery procedures are likely to be largely software-based, and may be observable if probes can be attached to the internal signal lines of the component computers. Special attention must be paid to loss or corruption of data. There is likely to be a dramatic reduction in performance on at least some of the processes in the system during recovery. One should pay special attention that output which occurred after the last checkpoint, but before the fault occurred, is not repeated during the recovery process.

Special apparatus: Probes into system, and error signal drivers, as described in Section 3.5.1. All of the normal throughput and error detection apparatus will be needed.

Limitations of test: The messages between processors may not be readily interpreted by the measurement equipment. The data may require another processor for interpretation.

5.2 Hardware Recovery from Faults

Hardware recovery techniques are generally applied using redundant units that are tightly coupled. The outputs of these units are continually or frequently compared, with the immediate suppression of errors through switching or voting. This approach substitutes the expense of additional hardware for the more complex and time-consuming software recovery process. If the fault-tolerant system relies entirely on hardware recovery techniques, the software may the identical to that which would run on a conventional computer system. For testing, one still needs special test software which will exercise the system so as to allow detection of imperfect fault recovery.

The tests of Sections 5.1.2 through 5.1.9 can be applied to systems with hardware fault recovery. One would expect that there would be **no** detectable performance or data-availability reduction resulting from the fault, but detailed resource measurements could give interesting insight on system functioning. Some systems may choose to handle the "first" fault by hardware redundancy methods, but resort to software recovery techniques if additional faults take place before sufficient faulty components have been replaced to permit hardware recovery from additional faults. As in any fault-tolerant system, one possible measurement result is that the system is *not* fault-tolerant.

6. Summary

The performance of fault-tolerant computer systems, in the absence of faults, can be measured using the hardware techniques discussed earlier by Roberts[ROB86]. Our interest here has been the *degradation* of performance (throughput, response time, data loss, etc.) which results from hardware faults. This degradation is generally a function of time after the fault occurs. Recovery may be full or partial. In most architectures, the *detection* of the fault is critical to fault tolerance. Fault containment and recovery cannot begin until the fault has been detected. Thus we have separated measurements designed to evaluate the *detection* of faults from the techniques which evaluate the overall ability of the of the system to continue useful output during and after recovery from the fault(s).

The test methods employed involve the artificial injection of errors into the system under test so that the nature of the fault (or at least its cause), and the time of its onset can be well known to the experimenter. Injection of faults is vital in order to have a fault rate which is high enough to allow evaluation to be accomplished in a reasonable time. Large scale integrated circuits limit the detail at which systems can be probed by the injection of faults, unless fault-injection provisions are made in the integrated circuits themselves. This restriction is not considered to be serious, since the *result* of the internal fault can usually be simulated by carefully planned forced modification of signals at the package pins.

By considering faults at the scale of large system elements, the scale at which faults will be detected, diagnosed and isolated by the recovery system, highly reliable results can be obtained at an expense which can be tolerated for important applications. No evaluation of a fault-tolerant system is likely to be exhaustive if attempted at the minor component level. The full range of combinations of faults in minor components cannot be induced in realistic detail and most evaluators cannot afford the equipment and time required to experimentally investigate all the combinations.

7. References

[AVIZ84] Avizienis, A. and Kelly, P. J. "Fault-Tolerance by Design Diversity: Concepts and Experiments", IEEE Computer, Vol. 17, No. 8, August, 1984, pp 67-80.

[CAR86] Carpenter, R. J., private communications, National Bureau of Standards, Parallel Processing Group, March and November, 1986.

[KUHL86] Kuhl, J. G. and Reddy, S. M., "Fault-Tolerance Considerations in Large, Multiple-processor Systems", IEEE Computer, Vol. 19, No. 3, March, 1986, pp 56-67.

[MARK85] Mark, P. B. "The Sequoia Computer: A Fault-Tolerant Tightly-coupled Multiprocessor Architecture", The 12th Annual International Symposium on Computer Architecture, Boston, Mass., June, 1985, pg. 232.

[MIN86] Mink, A., et al. "Simple Multiprocessor Performance Measurement Techniques and Preliminary Examples of Their Use", NBSIR 86-3416, National Bureau of Standards, July, 1986.

[NEST85] Nestle, E. and Inselberg, A. "The Synapse N+1 System: Architectural Characteristics and Performance Data of a Tightly-coupled Multiprocessor System", The 12th Annual International Symposium on Computer Architecture, Boston, Mass., June, 1985, pp 233-239.

[RENN86] Rennels, D. A. "On Implementing Fault-tolerance in Binary Hypercubes", Digest, 16th International Symposium on Fault Tolerant Computing Systems, Vienna, Austria, July, 1986, pp 344-349.

[ROB86] Roberts, John W., *Performance Measurement Techniques for Multi-Processor Computers*, NBSIR 85-3296, National Bureau of Standards, Gaithersburg, MD, February, 1986.

[SERL84] Serlin, O. "Fault-tolerant Systems in Commercial Applications", IEEE Computer, Vol. 17, No. 8, August, 1984, pp 19-30.

[SIEW84] Siewiorek, D. P., "Architecture of Fault-Tolerant Computers", IEEE Computer, Vol. 17, No. 8, August, 1984, pp 9-18.

[SIEW86] Siewiorek, D. P., "Fault Tolerance in Multiprocessor Systems", tutorial, 1986 International Conference on Parallel Processing, St. Charles, Illinois.

[ZORP85] Zorpette, G., "Computers that are 'Never' Down", IEEE Spectrum, April, 1985, pp 46-54.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 87-3568	2. Performing Organ. Report No.	3. Publication Date MAY 1987

4. TITLE AND SUBTITLE

On the
Measurement of Fault-Tolerant Parallel Processors

5. AUTHOR(S)

John W. Roberts, Alan Mink, Robert J. Carpenter

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

10. SUPPLEMENTARY NOTES

[ ] Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

Computer systems that continue to operate correctly in the presence of faults are vital for many important applications. A number of measurement techniques can be used to determine how well computers detect and recover from faults. Both time to recover and degree of recovery can be measured.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

computers; fault detection effectiveness; fault recovery effectiveness; fault tolerant; performance measurement

13. AVAILABILITY

[X] Unlimited
[ ] For Official Distribution. Do Not Release to NTIS
[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

49

15. Price

$11.95