Evaluation of Fault-Tolerance Latency from Real-Time Application's Perspectives[1]

Hagbae Kim and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109–2122
e-mail: {khbap,kgshin}@eecs.umich.edu; 313-763-0391(voice); 313-763-4617(fax)

Abstract

The *Fault-Tolerance Latency* (FTL) defined as the time required by all sequential steps taken to recover from an error is important to the design and evaluation of fault-tolerant computers used in safety-critical real-time control systems. To meet timing constraints or avoid *dynamic failure*, the latency of any fault-handling policy — that consists of several stages like error detection, fault location and recovery — must not be larger than the *Application Required Latency* (ARL), which depends upon the controlled process under consideration and its operating environment.

We evaluate the FTL while considering various fault-tolerance mechanisms and use the evaluated FTL to check if a fault-handling policy can meet the timing constraint, FTL $\leq$ ARL, for a given real-time application. The FTL is dependent on the underlying fault-handling mechanisms as well as fault behaviors during the application of temporal-redundancy recovery such as instruction retry or program rollback. We investigate all possible fault-handling scenarios and represent FTL with several random and deterministic variables that model the fault behaviors and/or the capability and performance of fault-handling mechanisms. We also present a simple example to demonstrate the application of the evaluated FTL in real-time systems, where an appropriate fault-handling policy is selected to meet the timing requirement with the minimum degree of spatial redundancy.

Index Terms — Fault-tolerance latency, real-time control systems, fault-tolerant controller computers, time/space and static/dynamic redundancy, hard deadline, dynamic failure

[1]The work reported was supported in part by a Texas Instruments Grant, the Office of Naval Research under Grant N00014–91–J–1115 and by the NASA under Grant NAG–1–1120. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the view of the funding agencies.

1 Introduction

Most real-time control systems have been realized with digital computers due mainly to the increasing capability and popularity of digital computers and both the high-performance and stringent-reliability requirements of real-time applications such as aircraft, nuclear reactors, and utility monitoring & control. Since failure of the controller computers in such applications may lead to catastrophe, e.g., loss of human lives or economic disaster, these computers must be equipped with appropriate fault-tolerance mechanisms which guarantee the safe operation of the system even in the presence of controller-component failures. A real-time controller computer may fail because of not responding fast enough or because of massive H/W or S/W component failures. In other words, a real-time computing system is required to deliver the expected services in a timely manner even in the presence of component failures. Thus, the requirement of a controller's fault-tolerance must be considered simultaneously with the timing constraints of the corresponding controlled process.

Fault-tolerance is achieved via temporal and/or spatial redundancy, and hence, its design methodologies are characterized by the tradeoff between these two types of redundancy. Most design criteria in non real-time systems deal with optimization of spatial redundancy, whereas in real-time systems time is so valuable to trade space for time. A fault-tolerance policy should be selected and implemented to recover completely from faults/failures within the time limit (deadline) of the underlying controlled process. We define *Fault-Tolerance Latency* (*FTL*) as the total time spent on such sequential fault-handling stages as error detection, fault location, system reconfiguration, and recovery of the contaminated application program.

Most work on fault-tolerance used simple models for FTL, which was also represented in [17] as the sum of Mean Time To Detection (MTTD) and Mean Time To Repair (MTTR). Reliability or dependability models assumed the recovery time to have a certain probability distribution; if the recovery time follows an exponential (or general) distribution, the transition from error state to normal state is represented by the mean rate in a Markov model (or a semi-Markov model). In [12], recovery procedures were represented by instantaneous probabilities which measure the effectiveness of fault-/error- handling mechanisms while ignoring the time spent on the recovery procedures due to the stiffness existing between fault occurrence and recovery. The authors of [6] derived a distribution of system-recovery times by using a truncated normal distribution and a displaced exponential distribution, which captures general short periods of normal recovery and special long durations of rare abnormal recovery. This work was based on the recovery time data collected from various (experimental) sources.

Note, however, that none of the foregoing approaches have treated the recovery process as consisting of several sequential stages such as fault detection/isolation, system reconfiguration, and recovery of contaminated computations; instead, they treated the recovery process as one event lumping all the sequential stages to (i) derive a simple expression for the recovery-time distribution to be solved analytically or (ii) implement the general models of error-/failure- handling with simulations.

In [5, 10], the experimental data/statistical methods (i.e., sampling and parameterestimation methods) for characterizing the times of fault detection, system reconfiguration, and computation recovery were discussed based on hardware fault injections in the Fault-Tolerant Multiple Processor (FTMP). In [1, 13], the recovery times were estimated for a pooled-spare and N-modular redundant systems. The effects of various fault-tolerance features on FTL were described there. However, the results were given in a specific application context applying spatial redundancy only, and assumed that the time required for each stage of fault/error recovery is approximated to be in a deterministic range.

In this paper, we propose to evaluate FTL analytically, covering most, if not all, practical fault-tolerance mechanisms based on the tradeoff between temporal and spatial redundancy. We first investigate the times required for all individual fault-/error- handling stages. Then, we tailor these results appropriately to represent all possible fault-/error- handling scenarios or policies. (A policy/scenario is composed of sequential fault-/error- handling stages.) Our analysis is based on the assumption that the latencies of fault-handling stages are stochastic depending upon the random characteristics of fault/error detection (or a random error latency) and fault behaviors; the active duration of a fault affects significantly the success/failure of a spatial-redundancy method (i.e., instruction retry or program rollback). Our results — that focus on a sequence of error-/failure- handling stages — can also be used in those well-developed reliability or dependability models [2, 4, 7].

In Section 2, general fault-tolerance features are described by classifying fault-tolerance mechanisms and considering the tradeoff between temporal and spatial redundancy. Section 3 examines the effects on the FTL of individual fault-handling stages from the occurrence of an error to its recovery, and combines these results to evaluate the FTL of a general fault-handling policy covering all possible fault-handling stages. In Section 4 we argue for the importance of FTL information to the design and validation of fault-tolerant controller computers. We present there an example that selects an appropriate fault-handling policy based on the FTL information. The paper concludes with some remarks in Section 5.

2 Generic Fault-Tolerance Features

Computer system failures occur due to errors, which are deviations from the program-specified behaviors. An error is the manifestation of a fault resulting from component defects, environmental interferences, operator or design mistakes. It is desirable to select an appropriate policy so as to continue the program-specified functions even in the presence of faults.

Fault-tolerance is achieved via spatial and/or temporal redundancy, i.e., systematic and balanced selection of protective redundancy among hardware (additional components), software (special programs), and time (repetition of operations). Thus, design methodologies for fault-tolerant computers are characterized by the tradeoff between spatial and temporal redundancy. Using these two types of redundancy, a fault-tolerant computer must go through as many as ten stages in response to the occurrence of an error, including fault location, fault confinement, fault masking, retry, rollback, diagnosis, recovery, restart, repair and reintegration. The design of fault-tolerant computers involves selection of an appropriate failure-handling policy that combines some or all of these stages.

Spatial redundancy is classified into two categories: static and dynamic. Static redundancy, also known as masking redundancy, can mask erroneous results without any delay as long as a majority of participant modules (processors or other H/W components) are nonfaulty. However, the associated spatial cost is high, e.g., three (four) modules are required to mask a non-Byzantine (Byzantine) failure in a TMR (QMR) system. The time overhead of managing redundant modules — for example, voting and synchronization — is also considerable for static redundancy. Dynamic redundancy is implemented with two sequential actions: fault/error detection and recovery of the contaminated computation. In distributed systems, upon detection of an error it is necessary to locate the faulty module before replacing it with a nonfaulty spare. Although this approach may be more flexible and less expensive than static redundancy, its cost may still be high due to the possibility of hastily eliminating modules with transient faults[2] and it may also increase the recovery time because of its dependence on time-consuming fault-handling stages such as fault diagnosis, system reconfiguration, and resumption of execution.

To overcome the above disadvantages, temporal redundancy can be used by simply repeating or acknowledging machine operations at various levels: micro-operation/single instruction (retry), program segment (rollback), or the entire program (restart). In fact, one of these recovery schemes is also needed to resume program execution in case of dynamic redundancy. This temporal-redundancy method requires high coverage of fault/error detection so as to invoke the recovery action quickly. (The same is also required for dynamic redundancy.) The main advantages of using temporal redundancy are not only its low spatial cost but also its low recovery time for transient faults. However, the time spent for this method would have been wasted in case of permanent or long-lasting transient faults, which may increase the probability of dynamic failure.

The relations between the temporal and spatial redundancy required (and the associated redundancy-management overhead) are shown in Fig. 1 for several fault-tolerance mechanisms. In case of time-critical applications, an appropriate fault-tolerance mechanism can be found from the top left of Fig. 1, i.e., paying a small amount of temporal redundancy at the cost of spatial redundancy like N-modular redundancy. When the timing constraint imposed by the controlled process is not tight, we can save the cost of spatial redundancy by increasing temporal redundancy (i.e., a larger time for retry, rollback, or restart recovery), which enhances the system's ability in recovering from more transient faults before the faulty modules are replaced with spares. Increasing temporal redundancy, however, increases the possibility of missing task deadlines or dynamic failure.

3 Evaulation of Fault-Tolerance Latency

The recovery process that begins from the occurrence of an error consists of several stages, some of which depend on each other, and the FTL is defined as the time spent for

[2]Note that more than 90% of faults are known to be non-permanent; as few as 2% of field failures are caused by permanent faults [11].

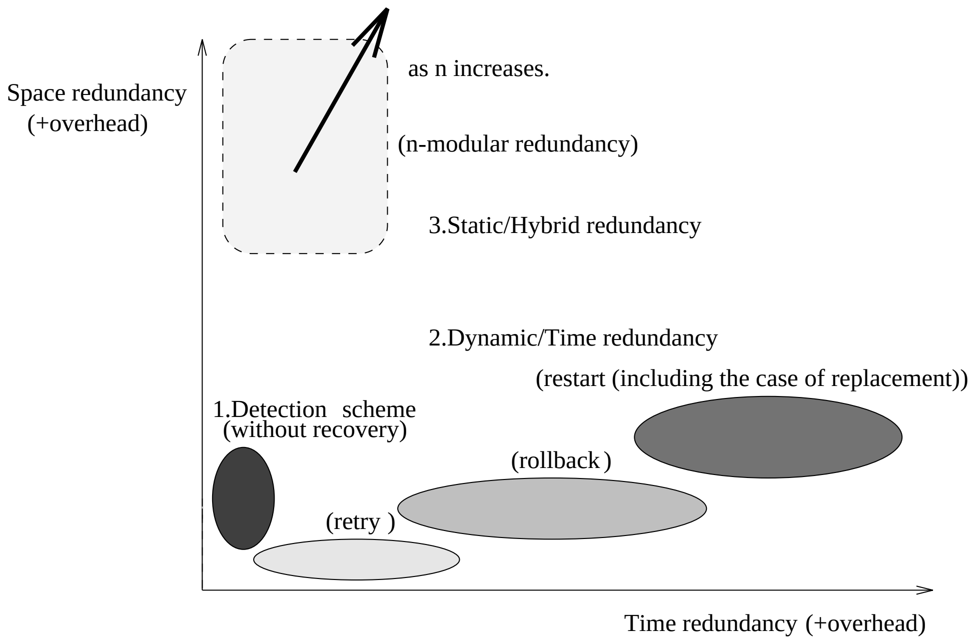

as n increases.
Space redundancy (+overhead)
(n-modular redundancy)
3.Static/Hybrid redundancy
2.Dynamic/Time redundancy
(restart (including the case of replacement))
1.Detection scheme (without recovery)
(rollback)
(retry)
Time redundancy (+overhead)

Figure 1: Tradeoff between temporal and spatial redundancy for various fault-tolerance mechanisms.

the entire recovery process. Thus, all the stages necessary to handle faults/failures upon occurrence of an error should be studied and their effects on the FTL must be analyzed.

In a specific application context, the recovery times were estimated in [1, 13] by decomposing the fault recovery process into stages and analyzing the effects of various fault-tolerance features on the FTL. We also use a similar approach to the problem of evaluating the FTL, but for more general fault-tolerance strategies. For completeness, these approaches are summarized below.

3.1 FTLs of a Pooled-Spares System

In [13], the FTL was estimated for a pooled-spares system implemented in the Dynamic Reconfiguration Demonstration System (DRDS) Program.[3] Phase 1 of the DRDS Program is reported to have shown potential benefits of the dynamic run-time reconfiguration implied by the pooled-spares approach, such as increased functional availability and flexibility, higher reliability, and less complexity than classical N-modular redundancy.

This work includes the analysis of FTL and has indicated that the pooled-spares approach can be used for many contemporary applications. The FTL was estimated for the demonstration system and the near-future (5–10 years from today) systems under the as-

[3]The DRDS Program is being developed to prove the feasibility of pooled spares for next generation weapon systems by Texas Instruments (TI) Incorporated under a contract from the Naval Air Warfare Center, Indianapolis, IN.

sumption that each fault-handling stage — fault detection/isolation, reconfiguration, and recovery — requires time within a deterministic range. In other words, (i) an upper bound of detection/isolation time was determined by using a fail-fast approach with health messages and the system is assumed to use a combination of continuous Built-In-Test (BIT), periodic BIT, and application-level detection/isolation methods, (ii) the reconfiguration times were actually measured on the demonstration system with cold backups for a specific load of size 8K 16-bit words, and (iii) recovery of the application code was performed via application checkpointing and rollback, and the required time was approximated to be a single checkpoint period. The expected reduction of FTL in the next 5–10 years was also estimated by considering such improvement factors as throughput and memory capacities.

In another paper dealing with the pooled-spares system [1], various fault-tolerance techniques covering both software and hardware issues were addressed by focusing on their latencies. This work also analyzed the FTL in the pooled-spares system based on the results of the DRDS Program in [13], and included part of the fault-masking method of N-modular redundancy. Possible fault detection/isolation, reconfiguration, recovery, and fault-masking features were considered to examine their effects on the FTL while considering the CPU speed and the relative rates of fault occurrences in individual components such as memory, I/O, buses, and processors.

Memory or data-path parity checking, error-correcting memory, checksum, reasonableness checks, health messages, data-type checks, watchdog timer, and periodic BIT were given as candidate fault detection/isolation mechanisms, while cold, warm, and hot spares were considered for classifying the system reconfiguration with the estimated data of the download and initialization delays depending upon the program size, bus speed, and CPU speed. Several characteristics of checkpointing such as consistency, independence, programmer-transparency, and conversation were introduced with the methods of software re-execution and rollback, where the time required for this recovery procedure was assumed to be a single checkpoint period as in [13].

Finally, the examples to select appropriate fault-handling policies to meet the given FTL requirement were presented for the cases of using cold, warm, and hot spares, illustrating how the analysis of FTL is applied.

3.2 The FTL of General Fault-Tolerance Mechanisms

Fig. 2 depicts all possible scenarios from an error occurrence to its recovery, covering static/dynamic redundancy, temporal-redundancy methods, and combinations thereof. Each *path* represents one fault-handling scenario which may occur as a result of selecting a fault-handling policy corresponding to the path and the success/fail result of the selected method, depending upon the fault behavior when a temporal-redundancy method is applied. For example, an unsuccessful retry implies another error detection, which may trigger a second retry or rollback or restart. As shown in Fig. 2, fault-handing processes are classified by fault/error detection and recovery mechanisms.

First, we divide the fault-handing process into several stages, and evaluate the time spent on each individual stage of Fig. 2. When a temporal-redundancy method such as retry or

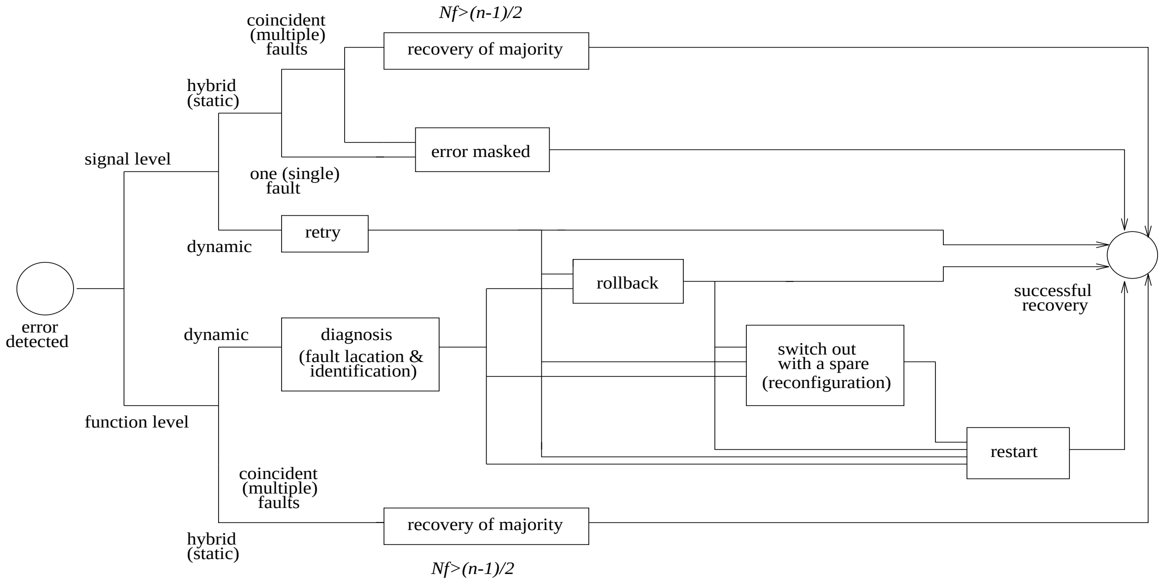


Figure 2: All possible failure-handling scenarios.

rollback is used upon detection of an error, we need such stages as fault diagnosis, system reconfiguration, and resumption of execution only after the temporal-redundancy method became unsuccessful in recovering from the fault. Whether the temporal-redundancy method is successful or not depends upon the policy used and the underlying fault behavior. Thus, we have to represent the effects of certain stages on the FTL in probabilistic terms.

3.2.1 Fault/Error Detection Process

The time interval between the occurrence of a fault and the detection of an error caused by the fault is divided into two parts by the time of error occurrence: *fault latency* for the time interval from the fault occurrence to the error generation, and *error latency* for the time interval from the error generation to the error detection. The distribution of fault latency was estimated in [16] by using the Gamma and Weibull distributions. Since the FTL begins with the occurrence of an error, we are mainly interested in the error latency which depends upon the active duration of a fault and the underlying detection mechanism. Error detection mechanisms are classified into (i) signal-level detection mechanisms, (ii) function-level detection mechanisms, and (iii) periodic diagnostics.

Let t_{el} and $F_{el}(t)$ be the error latency and its cumulative probability distribution, respectively. Several well-known *pdf*s, such as Weibull, Gamma, and lognormal distributions, were examined in [5] to model the error latency. If Δt_i is the mean execution time of an instruction, $F_{el}(\Delta t_i) \approx 1$ for a high-coverage signal-level detection mechanism. For a function-level detection mechanism, t_{el} depends on the detection mechanism used and the executing task. The coverage of the function-level detection is generally lower than that of the signal-level detection (and significantly less than one), and its error latency is thus larger than the signal-level detection's. We will therefore use the mean error latency of the function-level detection which is much larger than the signal-level detection's for the examples presented in Section 4. Although the periodic diagnostic whose coverage depends on both its period and duration is also a popular method to locate faults, only the first two types of detection mechanisms are considered in our analysis, because the results thus obtained can be extended to the case of periodic diagnostic.

3.2.2 Fault Masking (Static/Hybrid Redundancy)

This method filters out the effects of faulty modules as long as the number of faulty modules is not larger than $\frac{n-1}{2}$ for n-modular redundancy. The method induces the time overhead of redundancy management such as synchronization and voting/interactive consistency techniques even in the absence of faults, which increases with the degree of redundancy [9]. Although the time required for this type of recovery is almost zero it induces high spatial costs; when the number of modules available is limited, this method is not as reliable as the dynamic-redundancy method [13] and must be equipped with separate detection and recovery mechanisms for ultra-reliable systems [3].

A hazardous environment, like the one resulting from EMI, will affect the entire system and induce coincident, or common-source, faults in the multiple modules of an n-modular redundant system. If the number of faulty modules is larger than $\frac{n-1}{2}$ in such a harsh environment, then the recovery time, which also depends upon the adopted temporal- or spatial- redundancy method, is no longer negligible. We do not treat such a case because of the similarity of its recovery process to the case of dynamic redundancy.

3.2.3 Fault Diagnosis

When a function-level detection mechanism is used, upon detection of an error it is necessary to locate the faulty module[4] and/or to determine certain fault behaviors. Let t_d and p_d be the time spent for fault diagnosis and the probability of locating the faulty module (i.e., diagnostic coverage). Then, there is a tradeoff between t_d and p_d, which is usually difficult to quantify. The accuracy of diagnosis, which increases with the diagnosis time, affects greatly the results of the subsequent recovery and hence the FTL. Note that the time, t_d, taken for diagnosis is likely to be deterministic, because it is usually programmed *a priori*. We assume that t_d is sufficiently large to locate faulty modules, that is, $p_d \approx 1$.

3.2.4 System Reconfiguration

When a fault is located and identified as a permanent fault, the faulty module must be isolated from the rest of the system by replacing it with a spare module or switching it

[4]in distributed systems

off without replacement (thus allowing for graceful degradation). This process is necessary for both dynamic and hybrid redundancy. Specific hardware like the Configuration Control Unit (CCU) in FTMP [8], may be dedicated to handling system reconfiguration. This process (of using cold spares) generally consists of (i) switching power and bus connections, (ii) running built-in-test (BIT) on the selected spare module, (iii) loading programs and data, (iv) initializing the software. When warm spares are used, steps (i) and (ii) are not needed. The time taken for this process is also likely to be deterministic, which depends upon program size, system throughput, processor speed, and bus bandwidth. Let t_r be the time spent for system reconfiguration. We assume that t_r lies in a deterministic interval, $t_{r1} \leq t_r \leq t_{r2}$, where t_{r1} and t_{r2} are determined by the type of reconfiguration and several other factors described above. In fact, these values can be determined experimentally as was done in [1, 13].

3.2.5 Retry

This is the simplest recovery method using temporal redundancy, which repeats the execution of a micro-operation or instruction. To be effective, this method requires immediate error detection, i.e., almost perfect coverage of a signal-level detection mechanism yielding an error latency smaller than the execution time of a micro-operation or an instruction ($F_{el}(\Delta t_i) \approx 1$). The *retry period*, which is defined as a continuous-time interval or the number of re-executions, is the maximum allowable time for retry. In other words, a retry must be terminated when the retry period expires, regardless whether it is successful or not. Let t_{rp}, t_a, and $F_a(t)$ be the retry period, the active duration of a fault, and the probability distribution of the active duration, respectively. The result of a retry depends upon t_{rp} and t_a. When the retry is successful, the time it took is certainly smaller than the retry period and is equal to the fault duration, t_a. However, it is equal to the retry period, t_{rp}, when the retry became unsuccessful, and an alternative recovery method will be followed, thus increasing the FTL.

3.2.6 Rollback with Checkpoints

The inquality $t_{el} > \Delta t_i$ is allowed in this method, i.e., $F_{el}(\Delta t_i) < 1$. When an error is detected by a signal- or function- level detection mechanism, this method rolls back past the contaminated part of a program following a system reconfiguration in case of dynamic redundancy. It is invoked as the first step of recovery after an unsuccessful retry. The time taken for the rollback process is dependent upon the error latency, the inter-checkpoint interval, the number of checkpoints maintained, and the way checkpoints are selected for rollback. Let Δt_c and N_c be the inter-checkpoint interval and the maximum number of checkpoints necessary for rollback recovery, respectively. For simplicity we assume that the inter-checkpoint intervals are simply equidistant. (It is not difficult to extend our method to the case of non equidistant checkpoints, though the notation will become more complex.) When the rollback is successful, the time taken to restore the contaminated segment of a program is larger than the error latency but smaller than the error latency plus one inter-checkpoint interval; that is, equal to $\lceil \frac{t_{el}}{\Delta t_c} \rceil \Delta t_c$, where $\lceil x \rceil$ is the smallest integer larger than

x. If the fault is active during the entire period of rollback or the contaminated part is larger than the re-executed part of the program, the rollback recovery will fail and the corresponding "wasted" time is equal to $N_c \Delta t_c$.

3.2.7 Restart

If too much of the program is contaminated by an error due to a long error latency, its execution is repeated from the beginning. The time (computation loss) taken for the restart process depends upon (i) the time to detect an error and (ii) the types of restart (i.e., hot, warm, and cold restarts) following a system reconfiguration. We use t_e and $F_e(t)$ to denote the error-detection time measured from the beginning of the program execution and the probability distribution of error occurrences in a program (determined by the *pdf* of fault occurrence), respectively.

3.2.8 Combination of Failure-Handling Stages

As mentioned earlier, all failure-handling scenarios are described by the paths from error detection to the corresponding recovery in Fig. 2. It is clear that a fault-handling policy depends on several mutually exclusive events, where one event represents a scenario and its occurrence depends upon fault behaviors and the policy parameters. The probability of the occurrence of each event can thus be calculated by using the *pdf* of fault active duration (F_a) and the policy parameters such as Δt_c, N_c, or t_{rp}. The FTL of a certain fault-handling policy is thus obtained by using the probabilities of all possible events/scenarios and the times spent for those events/scenarios. Note that the time spent for each scenario is obtained by adding the times spent for all fault-handling stages on the path representing the scenario. Likewise we can obtain the probability distribution of a fault-handling policy (F_l) as:

$$F_l(t) = \sum_{i=1}^{n} F_l(t|S_i)P(S_i), \tag{3.1}$$

where S_i indicates the i-th scenario of a fault-handling policy, and $P(S_i)$ and n are the probability of the occurrence of S_i and the number of all possible scenarios in the selected fault-handling policy, respectively. Eq. (3.1) describes $F_l(t)$ as a weighted sum of conditional distribution functions. Each S_i's conditional distribution function is computed by convolving the probability distribution functions of the times spent for all the stages on the corresponding path or fault-handling policy. The times spent for all possible fault-handling stages are described as deterministic values or random variables with certain probability distribution functions. We will investigate each fault-handling stage individually.

Now, we characterize the fault-handling process into four policies according to the types of error-detection mechanisms and recovery methods combined with temporal and spatial redundancy; (i) restart after reconfiguration, (ii) rollback, (iii) retry, and (iv) retry then rollback. These cover all possible dynamic- and/or temporal- redundancy methods. Specifically, we can describe the fault-handling policies as follows. (Note that the number of all possible scenarios in each fault-handling policy is equal to n.)

- Policy 1 ($n = 2$): S_1 = successful restart after diagnosis and reconfiguration, S_2 = unsuccessful restart due to incorrect diagnosis then repeat.
- Policy 2 ($n = 2$): S_1 = successful rollback after diagnosis, S_2 = unsuccessful rollback then restart after diagnosis and reconfiguration.
- Policy 3 ($n = 2$): S_1 = successful retry, S_2 = unsuccessful retry then restart after reconfiguration.
- Policy 4 ($n = 3$): S_1 = successful retry, S_2 = unsuccessful retry and successful rollback, S_3 = unsuccessful retry and unsuccessful rollback then restart after reconfiguration.

While signal-level detection mechanisms can capture the faulty module immediately upon occurrence of an error and can thus invoke retry in Policies 3 and 4, function-level detection mechanisms — that cause a nonzero error latency and thus require the diagnosis process to locate the faulty module — may be used for Policies 1 and 2. The probabilities of scenario occurrences and the (conditional) distribution functions of the above scenarios are derived by using the variables defined earlier for individual fault-handling stages.

For simplicity, we do not consider the occurrence of an error/failure due to a second fault during the recovery from the first fault. If we need to consider the effects of such an error/failure, we cannot derive a closed-form distribution function of FTL, but can instead derive the moments of FTL by using recursive equations, which can then be used to derive the distribution function of FTL numerically.

Policy 1 is a simple form of dynamic redundancy, i.e., to restart the task from the beginning after identifying and replacing the faulty module with a nonfaulty spare. The first scenario is a successful restart with correct diagnosis. Thus, the probability of its occurrence is equal to that of successful diagnosis (p_d), and the time (t_l) spent on this scenario becomes:

$$t_l = t_{el} + t_d + t_r + t_e,$$

where t_d and t_r are deterministic variables, and t_{el} is a random variable with the distribution function, F_{el}. t_e is also a random variable with a certain conditional distribution function given that an error had occurred during the execution of a task, i.e., $F_e(t|\mathit{an\ error\ occurred})$. Let $t = t_l - t_d - t_r$, then:

$$\begin{aligned} P(S_1) &= p_d, \\ F_l(t|S_1) &= F_{el}(t) * F_e(t|\mathit{an\ error\ occurred}). \end{aligned} \tag{3.2}$$

Similarly, $P(S_2)$ and $F_l(t|S_2)$ are derived for the second scenario. Since an unsuccessful restart (of the second scenario) wastes more time than the first scenario by the amount of the incorrect–diagnosis time plus the (error) latency for a second error detection due to this incorrect diagnosis, t_l is changed to:

$$t_l = t_{el} + t_d + t_{el} + t_d + t_r + t_e = 2t_{el} + 2t_d + t_r + t_e.$$

Let $t = t_l - 2t_d - t_r$, then:

$$\begin{aligned} P(S_2) &= 1 - p_d, \\ F_l(t|S_2) &= F_{el}(\frac{t}{2}) * F_e(t|\textit{an error occurred}). \end{aligned} \tag{3.3}$$

Policies 2 and 3 use rollback with diagnosis and retry upon error detection, respectively. Reconfiguration is also called for if the temporal-redundancy approach became unsuccessful. Since the first scenario of Policy 2 is a successful rollback, the probability of its occurrence depends on the probability of successful diagnosis (p_d), the parameters of rollback (Δt_c and N_c), the error latency (t_{el}), and the fault active duration (t_a). For a successful rollback, (i) a faulty module must be identified with correct diagnosis, (ii) t_{el} must be smaller than $N_c\Delta t_c$, which is the maximum allowable time for rollback, and (iii) the fault must disappear within $N_c\Delta t_c$. Let p_t be the percentage of transient faults, then:

$$P(S_1) = p_t p_d F_a(N_c\Delta t_c) F_{el}(N_c\Delta t_c). \tag{3.4}$$

The time spent for this case is simply obtained as:

$$t_l = t_{el} + t_d + \lfloor \frac{t_{el}}{\Delta t_c} \rfloor \Delta t_c.$$

Let $t = t_l - t_d$, then:

$$F_l(t|S_1) = F_{el}(t) * F_m(\frac{t}{\Delta t_c}), \tag{3.5}$$

where F_m is a cumulative probability mass function for $m = \lfloor \frac{t_{el}}{\Delta t_c} \rfloor$. The probability of the second scenario being exclusive of the first one is equal to $1 - P(S_1)$:

$$P(S_2) = 1 - p_t p_d F_a(N_c\Delta t_c) F_{el}(N_c\Delta t_c). \tag{3.6}$$

The time spent for this scenario is increased to:

$$t_l = t_{el} + t_d + N_c\Delta t_c + t_d + t_r + t_e = t_{el} + 2t_d + N_c\Delta t_c + t_r + t_e.$$

Let $t = t_l - 2t_d - N_c\Delta t_c - t_r$, then:

$$F_l(t|S_2) = F_{el}(t) * F_e(t|\textit{an error occurred}). \tag{3.7}$$

For Policy 3 which does not require fault diagnosis due to the assumed immediate and correct detection of errors with signal-level detection mechanisms, the probabilities of scenario occurrences and distribution functions are derived similarly to Policy 2. For a successful retry, the error must be detected before contaminating the result of executing the instruction that will be retried (Δt_i) and the fault must become inactive within t_{rp}, if the time spent is $t_{el} + t_a$. Thus,

$$\begin{aligned} P(S_1) &= p_t F_a(t_{rp}) F_{el}(\Delta t_i), \\ F_l(t|S_1) &= F_{el}(t) * F_a(t). \end{aligned} \tag{3.8}$$

When a retry is unsuccessful, the time spent for this becomes:

$$t_l = t_{el} + t_{rp} + t_r + t_e.$$

Let $t = t_l - t_{rp} - t_r$, then:

$$\begin{aligned} P(S_2) &= 1 - p_t F_a(t_{rp}) F_{el}(\Delta t_i), \\ F_l(t|S_2) &= F_{el}(t) * F_e(t|\textit{an error occurred}). \end{aligned} \tag{3.9}$$

Policy 4 has three scenarios whose probabilities and distribution functions are obtained by combining those of Policies 2 and 3. The first scenario is a successful retry, for which $P(S_1)$ and $F_l(t|S_1)$ are equal to those of the first scenario in Policy 3 (i.e., Eq. (3.8)). The second scenario is a successful rollback following an unsuccessful retry. Thus, $P(S_2)$ and $F_l(t|S_2)$ can be obtained by modifying Eqs. (3.4) and (3.5) to include the effects of an unsuccessful retry. Let $t = t_l - t_{rp}$, then:

$$\begin{aligned} P(S_2) &= p_t[F_a(N_c\Delta t_c)F_{el}(N_c\Delta t_c) - F_a(t_{rp})F_{el}(\Delta t_i)], \\ F_l(t|S_2) &= F_{el}(t) * F_m(\frac{t}{\Delta t_c}). \end{aligned} \tag{3.10}$$

The third scenario is to restart with reconfiguration following an unsuccessful retry then rollback when the time spent is $t_l = t_{el} + t_{rp} + N_c\Delta t_c + t_r + t_e$. Thus, if $t = t_l - t_{rp} - N_c\Delta t_c - t_r$ then:

$$\begin{aligned} P(S_3) &= 1 - p_t F_a(N_c\Delta t_c) F_{el}(N_c\Delta t_c), \\ F_l(t|S_3) &= F_{el}(t) * F_e(t|\textit{an error occurred}). \end{aligned} \tag{3.11}$$

With these derived probabilities and conditional distribution functions, we can compute the probability distribution of FTL for each policy from Eq. (3.1).

4 Application of Fault-Tolerance Latency

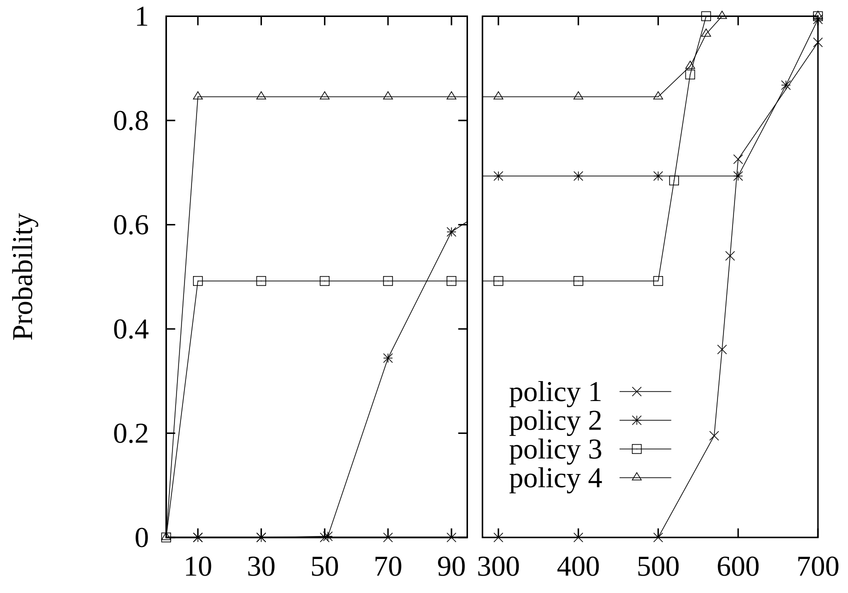
A real-time control system is composed of a controlled process/plant, a controller computers, and an environment, all of which work synergistically. A control system does not generally fail instantaneously upon occurrence of a controller failure. Instead, for a certain duration the system stays in a safe/stable region or in the admissible state space even without updating the control input from the controller computer. However, a serious degradation of system performance or catastrophe called a *dynamic failure* (or system failure), occurs if the duration of missing the update (or incorrect update) of the control input due to malfunctioning of the controller computer exceeds a certain limit called the *hard deadline* [15]. The hard deadline represents system inertia/resilience against a dynamic failure, which can be derived experimentally or analytically using the state dynamic equations of the controlled process, the information on fault behaviors involving environmental characteristics (such as electro-magnetic interferences), and the control algorithms programmed in the controller computers [14].

When an error/failure occurs in a controller computer, the error must be recovered within a certain period, called the *Application Required Latency* (ARL) [13], in order to avoid a dynamic failure. Roark *et al.* [13] presented several empirical examples of ARL for flight control, missile guidance, air data system, automatic tracking and recognition applications. It is important to note that one can derive the ARL analytically using the hard-deadline information [14], because the sum of ARL and the minimum time to execute the remaining control task to generate a correct control input is equal to the system's hard deadline. Using this information about the ARL/deadline of the controlled process and the FTL of the controller computer, one can select (or design) an appropriate fault-handling policy by making a tradeoff between temporal and spatial redundancy while satisfying the strict timing constraint, FTL$\leq$ARL. One can also estimate the system's ability of meeting the timing constraint in the presence of controller-computer failures, which is characterized by the probability of no dynamic failure using the evaluated ARL and FTL.

We now present an example to demonstrate the usefulness of the evaluated FTL. Consider a pooled-spares system which consists of multiple modules connected to a backplane. The system consists of power supplies, input/output modules, and a set of identical data processing modules, a subset of which are assigned to processing tasks. The remaining modules can be used as spares in case of an error/failure. Let the basic time unit be one millisecond, and let the task execution time in the absence of error/failure and the mean execution time of one instruction be given as $T = 50$ $(= 0.05sec)$ and $\Delta t_i = 0.002$ $(= 2\mu sec)$, respectively. The error latency is assumed to follow an exponential distribution with mean 12 and 0.002 for function- and signal- level detection mechanisms, respectively. The fault occurrence and duration are also governed by exponential distributions, where the mean value of active duration is 0.5, and the percentage of transient faults (p_t) is about 0.9. Then, given that an error occurred during T, the occurrence time, t_e, is uniformly distributed over T. The diagnosis time, t_d, is 50, which is assumed to yield coverage $p_d = 0.95$. When cold spares are used, it is assumed to take 500 units of time for system reconfiguration. We also assume that this value can be reduced to 100 by using warm spares. When applying rollback recovery, we set $\Delta t_c = 5$ and $N_c = 4$, whereas the retry period, t_{rp}, is set to 1. Under these conditions, the probability distribution functions of FTL are evaluated for the four representative policies by using the method developed in Section 3. These functions are plotted in Fig. 3.

From the above evaluations of FTL, one can conclude that Policies 1 and 2 are acceptable only when $t_{hd} > 14T(= 700 = 0.7sec)$. While the FTL of Policy 1 is distributed around $12T(= 600)$ with a small variance, Policy 2 has a wide range bounded by $14T$, indicating that Policy 2 is less likely than Policy 1 to violate the timing constraint, FTL$\leq$ARL, under the above chosen conditions. Policies 3 and 4 that use retry have better distributions as compared to Policies 1 and 2, which satisfy the constraint $t_{hd} > 12T$. However, to be effective, retry usually requires dedicated hardware and immediate error detection. (Note that the mean error latency of Policies 3 and 4 in Fig. 3 is 0.002.)

Fig. 4 plots the FTL while varying the policy parameters. We adopt a more accurate diagnosis process with $p_d = 0.97$, and change rollback and retry policies to $N_c = 5$ and $t_{rp} = 2$, respectively. The error-detection mechanisms are also improved to decrease the error latencies to 10 and 0.001 for both function- and signal- level mechanisms. In this case,



Fault-Tolerance Latency (FTL) [$msec$]

Figure 3: Probability distribution functions (PDF) of FTL with cold spares.

the mean values of FTL become smaller, but the upper bounds of FTL are not changed. Thus, we can draw the same conclusion as Fig. 3 in selecting an appropriate fault-handling policy.

We also consider different fault parameters: p_t and the mean value of active duration are changed to 0.95 and 0.25, respectively. Since the temporal-redundancy approaches get better under such conditions, the FTLs of Policies 2, 3, and 4 cluster small values, as depicted in Fig. 5. However, one cannot still neglect some possible FTLs larger than $10T$, albeit with small probabilities.

When the hard deadline t_{hd} is tight like $t_{hd} \leq 10T (= 500)$, no policy can meet the timing constraint, FTL$\leq$ARL. It is shown in Figs. 4 and 5 that the FTL does not change significantly even if the policy and/or fault parameters are changed. Considering the fact that reconfiguration is the most time-consuming among all the fault-handling stages, we use warm spares to reduce t_r, which skews significantly the probability distribution functions of the FTLs to the right, as shown in Fig. 6 where all parameters but t_r are the same as those in Fig. 3. In that case, Policies 3 and 4 are suitable for systems with $t_{hd} > 4T (= 200)$.

If the timing constraint is tighter, e.g., $t_{hd} \leq 4T$, we can conclude that static redundancy (or hot spares) must be used at the expense of spatial redundancy, since no policy using dynamic or temporal redundancy can satisfy the stringent constraint.

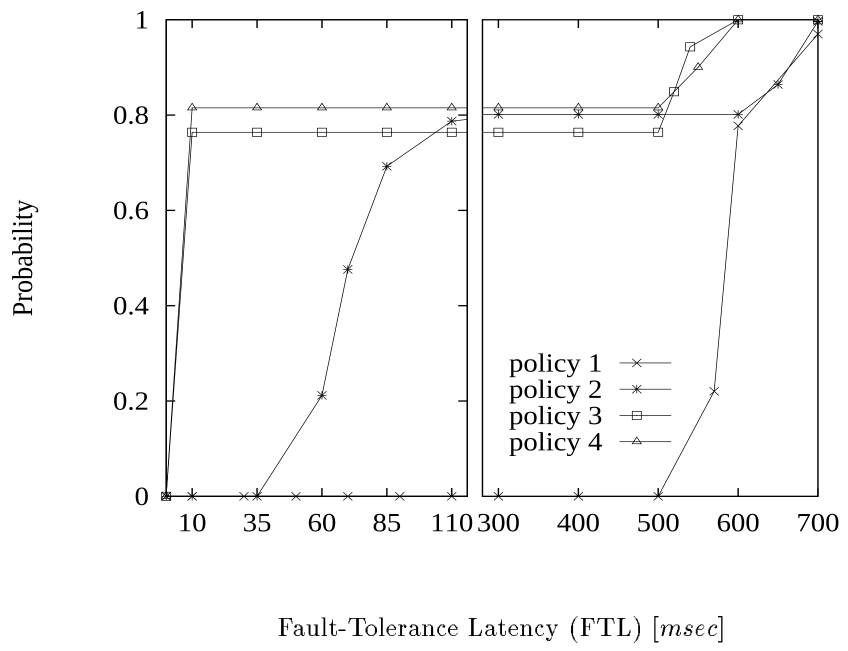


Figure 4: PDFs of the FTLs with policy parameters different from those of Fig. 3: $p_d = 0.97$, $N_c = 5$, and $t_{rp} = 2$.

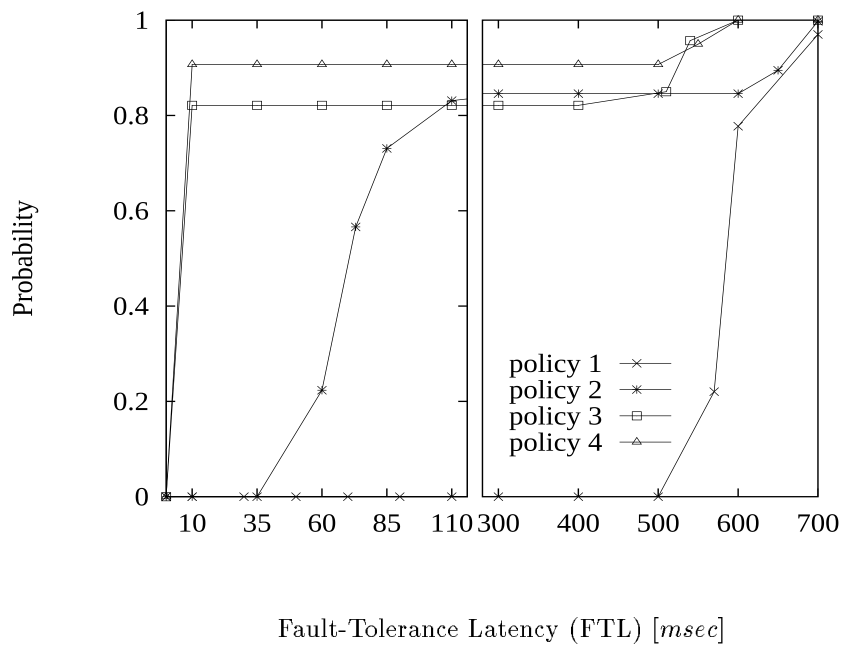


Figure 5: PDFs of the FTL under fault environments different from those of Fig. 3: $p_t = 0.95$ and $E(t_a)$ (the mean active duration)= 0.25.

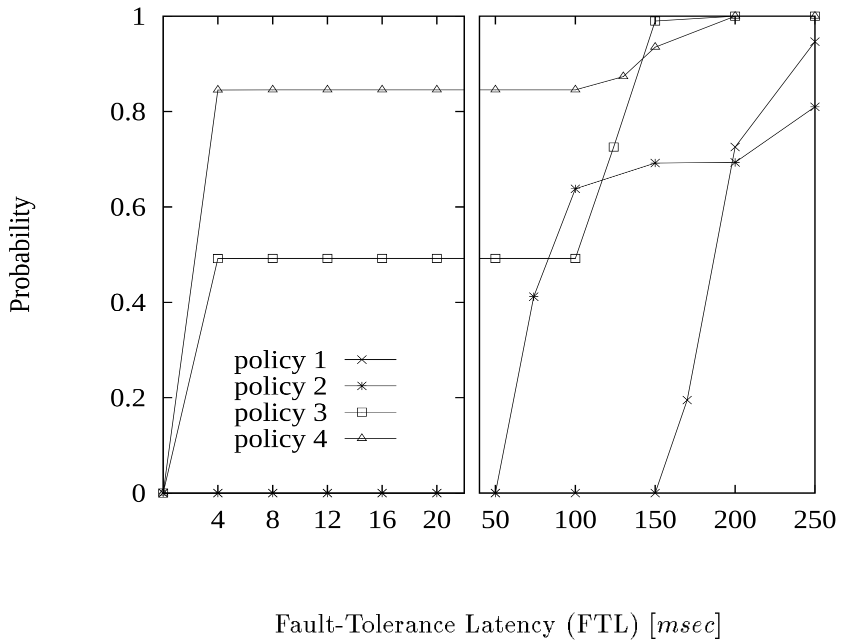


Figure 6: PDFs of the FTLs with a reconfiguration strategy different from that of Fig. 3: warm spares.

5 Conclusion

In this paper, we evaluated the FTL for general failure-handling policies that combine temporal and spatial redundancy. We investigated all the individual fault-handling stages from error detection to its complete recovery, and used some deterministic and random variables to model the times spent for these stages. This is in sharp contrast to the previous work that evaluated fault-recovery times with simple models or deterministic data collected from experiments. As shown in a simple example — although the parameters used in the example are chosen arbitrarily, their choice would not change the conclusion we have drawn — the evaluated FTL is a key to the selection of an appropriate fault-handling policy, especially for real-time controller computers.

Although we assumed the latencies of individual stages to be available by measuring or modeling relevant variables, it is in reality quite difficult to obtain their accurate values, some of which depend on each other and also on applications. We are currently investigating ways to evaluate or model the times spent on the individual stages while considering the system architecture, task type, and the implementation of each stage.

There are also several interesting related problems worth further investigation, including:

- Although it is not always required (because of, for example, a successful temporal-redundancy method like retry), system reconfiguration is generally the most time-consuming stage of fault-/error- handling. We are currently analyzing the latency of system reconfiguration, examining all of its features: (i) cold, warm, and hot spares, (ii) active or passive reconfiguration (and graceful degradation), (iii) complexity of switching and initiated built-in-testing of a spare, (iv) the tradeoff between computational capacity and system reliability associated with each reconfiguration strategy.

- It is important to derive, if possible, a closed-form *pdf* expression for the time spent for each individual stage. In case an exact closed-form *pdf* is not obtainable, an approximate expression may be used to determine an appropriate fault-/error- handling policy.
- The FTL strongly depends upon fault coverage, which was assumed to be a constant determined by the diagnosis stage. However, fault coverage is in reality not simple to determine due mainly to the effects of many coupled testing/detecting methods. The task type and the failure occurrence rate also affect fault coverage. When periodic diagnoses are used, there is a tradeoff between accuracy (fault coverage) and time (frequency and diagnosis time). It is important to study all the factors determining fault coverage and analyze its effects on the FTL.

Acknowledgement

The authors would like to thank Chuck Roark and his colleagues of the TI Microelectronic Center, Plano Texas, for the quantitative data of the reconfiguration time.

References

[1] P. Barton, "Fault latency white paper," Technical report, Texas Instruments, Microelectronics Department, Plano, TX, January 1993.

[2] R. W. Butler and A. L. White, "SURE reliability analysis," *NASA Technical Paper*, March 1990.

[3] P. K. Chande, A. K. Ramani, and P. C. Sharma, "Modular TMR multiprocessor system," *IEEE Trans. on Industrial Electronics*, vol. 36, no. 1, pp. 34–41, February 1989.

[4] J. Dugan, K. Trivedi, M. Smotherman, and R. Geist, "The hybrid automated reliability prediction," *AIAA Journal of Guidance, Control and Dynamics*, pp. 319–331, May 1986.

[5] G. B. Finelli, "Characterization of fault recovery through fault injection on FTMP," *IEEE Trans. on Reliability*, vol. R-36, no. 2, pp. 164–170, June 1987.

[6] R. M. Geist, M. Smotherman, and R. Talley, "Modeling recovery time distributions in ultrareliable fault–tolerant systems," in *Digest of Papers, FTCS-20*, pp. 499–504, June 1990.

[7] R. M. Geist and K. S. Trivedi, "Ultrahigh reliability prediction for fault–tolerant computer systems," *IEEE Trans. on Computer.*, vol. C-32, no. 12, pp. 1118–1127, December 1983.

[8] A. L. Hopkins Jr., T. B. Smith III, and J. H. Lala, "FTMP–a highly reliable fault-tolerant multiprocessor for aircraft," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1221–1239, October 1978.

[9] C. M. Krishna, K. G. Shin, and R. W. Butler, "Synchronization and fault-masking in redundant real-time systems," in *Digest of Papers, FTCS-14*, pp. 152–157, June 1984.

[10] J. H. Lala, "Fault detection, isolation and configuration in FTMP: Methods and experimental results," in *Proc. 5th IEEE/AIAA Digital Avionics Systems Conf.*, pp. 21.3.1–21.3.9, 1983.

[11] S. R. McConnel, D. P. Siewiorek, and M. M. Tsao, "The measurement and analysis of transient errors in digital computer systems," in *Digest of Papers, FTCS-9*, pp. 67–70, June 1979.

[12] J. McGough, M. Smotherman, and K. S. Trivedi, "The conservativeness of reliability estimates based on instantaneous coverage," *IEEE Trans. on Computers*, vol. C–34, no. 7, pp. 602–608, July 1985.

[13] C. Roark, D. Paul, D. Struble, D. Kohalmi, and J. Newport, "Pooled spares and dynamic reconfiguration," in *Proceedings of NAECON'93*, pp. 173–179, May 1993.

[14] K. G. Shin and H. Kim, "Derivation and application of hard deadlines for real–time control systems," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 22, no. 6, pp. 1403–1413, Nov./Dec. 1992.

[15] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "A unified method for evaliating real–time computer controller and its application," *IEEE Trans. on Automat. Contr.*, vol. AC–30, no. 4, pp. 357–366, April 1985.

[16] K. G. Shin and Y.-H. Lee, "Measurement and application of fault latency," *IEEE Trans. on Computers*, vol. C–35, no. 4, pp. 370–375, April 1986.

[17] D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design*, Digital Equipment Corporation, Bedford, MA, 1982.